

Linked Data Patterns

**A pattern catalogue for modelling,
publishing, and consuming Linked Data**

Leigh Dodds

Ian Davis

Linked Data Patterns: A pattern catalogue for modelling, publishing, and consuming Linked Data

Leigh Dodds
Ian Davis

Publication date 2011-08-19

Abstract

This book lives at <http://patterns.dataincubator.org>. Check that website for the latest version.

The book is also available as both a PDF [<http://patterns.dataincubator.org/book/linked-data-patterns.pdf>] and EPUB [<http://patterns.dataincubator.org/book/linked-data-patterns.epub>] download.

This work is licenced under the Creative Commons Attribution 2.0 UK: England & Wales License. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/2.0/uk/>.

Thanks to members of the Linked Data mailing list [<http://lists.w3.org/Archives/Public/public-lod/>] for their feedback and input, and Sean Hannan [<http://twitter.com/MrDys/>] for contributing some CSS to style the online book.

Table of Contents

| | |
|-----------------------------------|----|
| 1. Introduction | 1 |
| Overview | 1 |
| 2. Identifier Patterns | 4 |
| Hierarchical URIs | 4 |
| Literal Keys | 5 |
| Natural Keys | 5 |
| Patterned URIs | 6 |
| Proxy URIs | 7 |
| Shared Keys | 8 |
| URL Slug | 9 |
| 3. Modelling Patterns | 11 |
| Custom Datatype | 11 |
| Index Resources | 12 |
| Label Everything | 13 |
| Link Not Label | 14 |
| Multi-Lingual Literal | 16 |
| N-Ary Relation | 17 |
| Ordered List | 18 |
| Ordering Relation | 19 |
| Preferred Label | 20 |
| Qualified Relation | 21 |
| Reified Statement | 23 |
| Repeated Property | 24 |
| Topic Relation | 25 |
| Typed Literal | 26 |
| 4. Publishing Patterns | 28 |
| Annotation | 28 |
| Autodiscovery | 29 |
| Document Type | 29 |
| Edit Trail | 31 |
| Embedded Metadata | 31 |
| Equivalence Links | 32 |
| Link Base | 34 |
| Materialize Inferences | 35 |
| Named Graphs | 36 |
| Primary Topic Autodiscovery | 37 |
| Progressive Enrichment | 38 |
| See Also | 38 |
| 5. Application Patterns | 40 |
| Assertion Query | 40 |
| Blackboard | 41 |
| Bounded Description | 42 |
| Composite Descriptions | 43 |
| Follow Your Nose | 44 |
| Missing Isn't Broken | 45 |
| Parallel Loading | 46 |
| Parallel Retrieval | 47 |
| Resource Caching | 48 |
| Schema Annotation | 48 |
| Smushing | 50 |
| Transformation Query | 52 |

Chapter 1. Introduction

Abstract

There are many ways to help spread the adoption of a technology, and to share skills and experience amongst a community of practitioners. Different approaches work well for communicating different kinds of knowledge. And we all individually have a preferred means of acquiring new skills, or getting help with a specific problem. Reference manuals, tutorials, recipes, best practice guides and experience reports all have their role. As do training courses, mentoring, pair programming and code reviews.

This book attempts to add to the steadily growing canon of reference documentation relating to Linked Data. Linked Data is a means of publishing "web-native" data using standards like HTTP, URIs and RDF. The book adopts a tried and tested means of communicating knowledge and experience in software development: the design pattern. The book is organized as a pattern catalogue that covers a range of different areas from the design of web scale identifiers through to application development patterns. The intent is to create a ready reference that will be useful for both the beginner and the experienced practitioner alike. It's also intended to grow and mature in line with the practitioner community.

Overview

Why A Pattern Catalogue?

Design patterns have a number of benefits:

- Patterns have a well-defined structure that encourages focus on the essential knowledge being communicated. This makes them accessible and easy to consume.
- Patterns encourage discussion of related and complementary approaches. Design decisions are rarely clear cut. A focus on specifics is useful for understanding trade-offs
- Patterns provide a name for a particular design decision or solution. Collectively they build a lexicon that encourages clearer communication between practitioners

The authors have successfully applied design patterns in their software development activities. The approach seemed well suited to teasing out some of the experiences and lessons they have learnt through working with Semantic Web technologies; the rigour of a pattern approach also helped the authoring.

...And Why a Book? Why Not a Blog or a Wiki?

This is something that we wrestled with for a long time. Our goal is that this book should ultimately reflect the collective experience of the Linked Data community and we want to encourage participation and feedback. You can use the dataincubator.org [http://dataincubator.org] mailing list to discuss the book and debate its contents. We're also hoping to include your submissions, corrections and edits in the future.

But while we want this book to grow as a participatory activity we (particularly Leigh) felt that an editorial layer would be a useful addition to this process. Helping firm up the naming, communication and organisation of the pattern catalogue as it develops.

We also encourage the community, if they find a design pattern approach to be useful, to publish and share their own patterns using whatever mechanism feels right for them. A thousand flowers, etc. The OntologyDesignPatterns.org [http://OntologyDesignPatterns.org] wiki provides one forum for helping to contribute to this effort.

What's Not Covered?

This book isn't a primer on RDF or OWL. There are already plenty of good sources of introductory material on the technologies discussed here. The book makes the assumptions that you have some basic understanding of RDF, RDF Schema and possibly OWL. The examples are given in Turtle syntax, so you should be familiar with that syntax too.

If you're looking for a deeper introduction to modelling with RDF Schema and OWL then you should read *Semantic Web for the Working Ontologist* [<http://workingontologist.org/>]. It's a great book that will give you a thorough understanding of how to apply the technologies. We're hoping that this work is in some sense a companion piece.

How the Catalogue Is Organized

The catalogue has been broken down into a number of separate chapters. Each chapter collects together patterns that have a common theme.

- Chapter 2, *Identifier Patterns*
- Chapter 3, *Modelling Patterns*
- Chapter 4, *Publishing Patterns*
- Chapter 5, *Application Patterns*

The catalogue also includes a few patterns that arguably aren't patterns at all, they're similar features of the RDF model; Typed Literal for example. We decided to include these for the sake of helping to document best practices. There are plenty of examples and material on some of these basic features but they're often overlooked by both experienced and new practitioners. So we've opted to document these as patterns to help draw attention to them.

Examples

The pattern catalogue includes a number of examples. Snippets of code, data, or queries that help to illustrate a specific pattern. Code examples are shown in this font.

Where we should examples of RDF data, we have used the Turtle [<http://www.w3.org/TeamSubmission/turtle/>] syntax for RDF because its more concise and readable than RDF/XML. We have preferred to use prefixed names for standard RDF properties and classes, but do not include the declarations of these prefixes in the examples.

Unless otherwise stated, assume that when you're looking at some Turtle we've declared the following prefixes:

```
@prefix ex: <http://www.example.org/>
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix dcterms: <http://purl.org/dc/terms/> .
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix skos: <http://www.w3.org/2004/02/skos/core#> .
```

@prefix xsd: <<http://www.w3.org/2001/XMLSchema#>> .

Chapter 2. Identifier Patterns

Abstract

This chapter introduces a collection of patterns that all relate to the generation, assignment and creation of web-scale identifiers (URIs) for Linked Data.

Hierarchical URIs

How should URIs be assigned to a group of resources that form a natural hierarchy?

Context

It is often the case that a collection of resources may form a natural hierarchy. E.g. the chapters within a book, or the departments within an organization. Reflecting this strict hierarchy within the URI structure makes those URIs more hackable allowing users/developers to "navigate" up the hierarchy by pruning the URI.

Solution

Where a natural hierarchy exists between a set of resources use Patterned URIs that conform to the following pattern:

```
:collection/:item/:sub-collection/:item
```

E.g. in a system which is publishing data about individual books and their chapters, we might use the following identifier for chapter 1 of a specific book:

```
/books/12345/chapters/1
```

The `/chapters` URI will naturally reflect to the collection of all chapters within a specific book. The `/books` URI maps to the collection of all books within a system, etc.

Example(s)

The discogs dataset in dataincubator uses hierarchical uris of the form:

```
http://discogs.dataincubator.org/release/22530/track/1-01
```

Discussion

This technique is best suited to scenarios where the items in the sub-collections (chapters) are always associated with a single parent item. Other relationships might exist, e.g. the chapter may be included in another but the chapter is always associated with at least one book: they do not exist in isolation. In circumstances where this doesn't hold true, then it is best to just use simple Patterned URIs.

The same applies to circumstances where the hierarchy may change over time.

Related

- Patterned URIs

Literal Keys

How do we publish non-global identifiers in RDF?

Context

The Natural Keys pattern encourages the creation of URIs from existing non-global identifiers. While this provides a way to begin identifying a resource so that we can describe it in RDF, it does not address the issue of how to publish these existing identifiers. Nor does it address situations where natural keys change over time, e.g. the move from ISBN-10 to ISBN-13 in the publishing world.

Solution

Create a custom property, as a sub-class of the `dc:identifier` property for relating the existing literal key value with the resource.

Example(s)

The nasa dataset in dataincubator uses Patterned URIs based on the NSSDC international designator, but includes these as literal values associated with each spacecraft using a custom property.

Discussion

While hackable URIs are a useful short-cut they don't address all common circumstances. For example different departments within an organization may have different non-global identifiers for a resource; or the process and format for those identifiers may change over time. The ability to algorithmically derive a URI is useful but limiting in a global sense as knowledge of the algorithm has to be published separately to the data.

By publishing the original "raw" identifier as a literal property of the resource we allow systems to look-up the URI for the associated resource using a simple SPARQL query. If multiple identifiers have been created for a resource, or additional identifiers assigned over time, then these can be added as additional repeated properties.

For systems that may need to bridge between the Linked Data and non-Linked Data views of the world, e.g. integrating with legacy applications and databases that do not store the URI, then the ability to find the identifier for the resource provides a useful integration step.

Related

- Patterned URIs
- Natural Keys

Natural Keys

How can we create unique URIs from data that already has unique identifiers?

Context

It is often the case that a group of resources already have a unique identifier. This might be a local system identifier derived from, e.g. a database key; or a global non-URI identifier, e.g. an ISBN.

Solution

Mint URIs that are algorithmically derived from the existing non-URI identifier. This can be as simple as concatenating the existing identifier or key with a suitable base URI. The existing identifier may need to be URL encoded before creating the URI. It is common to combine this technique with Patterned URIs.

Example(s)

The BBC programmes website uses URIs that are derived from its existing "programme id" or pid.

Discussion

Where resources are already associated with existing keys, it is likely that the creation and management of those identifiers will already be supported by a specific technology or process. There is a need to be able to create global URI based identifiers for these resources without creating unnecessary additional overheads in creating entirely new identifiers and/or mapping between URIs and existing keys.

By deriving the URI from the natural key for the identifier we avoid the need to create a new process for assigning identifiers and largely eliminate the need to have a mapping between the two identification systems.

Related

- Patterned URIs

Patterned URIs

How can we create more hackable, human-readable URIs?

Context

Clean, clear URIs can be easier to remember and easier for developers to work with. This is especially true if the URIs can be algorithmically constructed or follow a common pattern. This allows URIs to be constructed or hacked in order to create new entry points into the dataset, e.g. determining the URI for a collection of similar resources based on knowledge of a single example URI.

Solution

Create URIs that follow a simple naming pattern. For applications generating Linked Data one technique for building patterned URIs is to use the pluralized class name as part of the URI pattern.

For example if an application will be publishing data about book resources, which are modelled as the `rdf:type ex:Book`. One might construct URIs of the form:

```
/books/12345
```

Where `/books` is the base part of the URI indicating "the collection of books", and the `12345` is an identifier for an individual book.

If multiple classes share a common base class, then it is also possible to use the name of the common base class, rather than generating separate URIs for each derived type

Example(s)

The BBC website uses /programmes to group together URIs that relate to series, brands and episodes, all of which are subclasses of the `rdf:type po:Programme`

Discussion

There are clear benefits from having human-readable, hackable URIs. This solution achieves that by ensuring the same naming scheme that applies to the underlying data also applies to the URIs. This provides a clear relation between the URI and the type of thing that it describes.

Related

- Hierarchical URIs
- Natural Keys

Proxy URIs

How do we deal with lack of standard identifiers for third-party resources?

Context

While it is a goal to reuse identifiers across datasets wherever possible, this is often difficult to achieve in practice. An authority for some specific data, e.g. ISO, may not have assigned unique URIs to resources in their dataset. Datasets also appear online at different times, making reuse difficult until more authoritative data appears and convergence happens on common identifiers. In these circumstances, how should identifiers be created for these third-party resources.

Solution

Treat third-party resources identically to those in your own data and assign them URIs within your domain.

Example(s)

There is still no agreed standard way of generating URIs for Internet Media Types. IANA have adopted RDF for publishing descriptions of registered media types. A data set containing descriptions of images may therefore use locally minted URIs for those media types:

```
ex:anImage a foaf:Image;  
  dc:format <http://www.example.org/media-types/image/jpeg>
```

Discussion

A publisher should focus on their immediate goal of opening up their data, ensuring that the published data is internally consistent and has identifiers for all key concepts. If existing identifiers exist then these should be reused. Where they don't then new locally minted URIs should be created from Shared Keys.

Once the data has been published, some alignment can take place within a community to achieve agreement on standard URIs for shared identifiers. One approach for achieving this alignment is to publish Equivalence Links.

Related

- Shared Keys
- Equivalence Links

Shared Keys

How do we simplify the inter-linking of datasets?

Context

It is common to carry out inter-linking of datasets as a separate activity following the initial modelling and data conversion exercises have been completed. How can the final stage of inter-linking be simplified?

It is also common that within a specific domain there will be a set of non-web identifiers that are standardised across different applications and publishers. How can the inter-linking of those datasets be simplified, encouraging convergence on web identifiers?

Solution

Create Patterned URIs by applying the Natural Keys pattern, but prefer public, standard identifiers rather than internal system specific codes.

Example(s)

The BBC have created URIs for artists that are algorithmically related to the MusicBrainz URIs using a common Shared Key. MusicBrainz URIs are Patterned URIs built from a "MusicBrainz ID", e.g. a74b1b7f-71a5-4011-9441-d0b5e4122711. The MusicBrainz and BBC URIs are shown below:

```
http://www.bbc.co.uk/music/artists/a74b1b7f-71a5-4011-9441-d0b5e4122711  
http://musicbrainz.org/artist/a74b1b7f-71a5-4011-9441-d0b5e4122711
```

Discussion

Predictable URIs structures make it easy for application developers and toolkits to build URIs from simple templates. By using URIs that are constructed from public identifiers, that already have scope outside of the immediate application, it increases the ease with which inter-linking can take place. For example, the pattern may avoid the need to look-up URIs in a SPARQL endpoint, allowing a developer to simplify use a URI template.

The Shared Keys pattern is best suited to situations where the shared identifiers are stable and rarely change.

Related

- Patterned URIs>

- Natural Keys

URL Slug

How can we create URLs from arbitrary text or keywords?

Context

When generating a URI for an identifier we may not always have a simple numeric Natural Key for a resource. In some cases, e.g. for documents, keywords, categories, we may only have a title or name.

Solution

Apply the Patterned URI pattern to create a common root URI and generate a simple URL "slug" by from the available text or keywords

There are several potential algorithms for normalising a string to create a URL slug. A typical algorithm would be:

- Lowercase the string
- Remove any special characters and punctuation that might require encoding in the URL
- Replace spaces with a dash

The original text is then preserved by using it to label the resource.

Example(s)

For example if we are generating a URI for a category called "Heavy Metal" we might generate a URI as follows

```
#Generate a patterned URI with a simple URL slug from "Heavy Metal"  
<http:www.example.org/category/heavy-metal>  
  rdfs:label "Heavy Metal"
```

Discussion

Clean URLs are easier to work with. If a dataset (or any website) has an inconsistent policy about the casing and encoding used in URLs then it can be more difficult to generate links to that dataset.

The use of URL slugs is prevalent in a number of different web frameworks already, e.g. blogging platforms. So this approach is already well-deployed.

Normalization of text strings can be problematic if it results in the same URL slug being generated for different resources. In some systems names and titles may be unique so this might not be an issue. Using patterned URIs to divide the URI space into different "sections" for different types of resources can help avoid unwanted collisions. Obviously in some cases collisions might be useful, e.g. to ensure that user input, which might vary in casing, is normalized into a standard form.

Related

- [Patterned URI](#)
- [Natural Key](#)

Further Reading

- [Slug \(web publishing\) on Wikipedia](#)

Chapter 3. Modelling Patterns

Abstract

This chapter introduces a collection of patterns that all relate to RDF modelling

Custom Datatype

A data model contains structured values that don't correspond to one of the pre-existing XML Schema datatypes

Context

Some applications may have pre-defined custom datatypes for describing the structure of literal values. It is useful to preserve this type information when publishing it as RDF in order to allow applications to handle them correctly.

Solution

Create a URI to identify the custom datatype and use that URI when creating Typed Literals

Example(s)

```
#Define a URI for different forms of shipping codes
<http:www.example.org/datatype/FedexShippingCode>
  rdfs:label "Fedex Shipping Code".
<http:www.example.org/datatype/UPSShippingCode>
  rdfs:label "UPS Shipping Code".

#Indicate the form of shipping code with a Typed Literal
_:delivery1
  ex:shippingCode "1234-568"^^<http:www.example.org/datatype/FedexShippingCode>.

_:delivery1
  ex:shippingCode "abcd/zx"^^<http:www.example.org/datatype/UPSShippingCode>.
```

Discussion

RDF does not place any limits on what datatypes might be associated with a literal. The model recommends using a standard type library, like XML Schema, wherever possible but the only real constraint is that datatypes should be identified by a URI. Creating a new URI for an application or domain specific datatype allows the type system to be easily extended. The datatype URI can be annotated with additional RDF statements, e.g. a label or description, to describe its usage. With sufficient detail on the data type, reasoners may be better equipped to draw new knowledge from a data model or highlight inconsistencies.

The key limitation to custom datatypes is that SPARQL query engines will not understand how to compare values of that type. Some SPARQL query processors may provide support for understanding the range of types it understands.

Note that a typed literal with a custom datatype can also be modelled as a sub-property. Continuing the above example, instead of defining a new type for each shipping code, the model could have instead been structured as:

```
<http://www.example.org/def/post/fedexShippingCode>
  a rdfs:Property;
  rdfs:subPropertyOf <http://www.example.org/def/post/shippingCode>
  rdfs:label "Fedex Shipping Code".
```

```
#Use the derived property:
_:delivery1
  ex:fedexShippingCode "1234-568".
```

,

The advantages of this alternate approaches are:

- Simpler querying of data by allowing the use of triple patterns, rather than FILTERs for extracting data of a common type.
- More explicit, fine-grained semantics

But there are several disadvantages to this approach:

- Increases the number of very similar properties, requiring redundancy of data or reasoning to allow applications to treat them generically
- Doesn't address the core requirement to indicate the lexical form of a structured value

With this in mind the general recommendation is to:

- Use a custom datatype to label particular types of structured value that share a common lexical form. These values may be associated with a broad range of different properties. Processing applications may want to implement a common set type conversions or display options for the values.
- Use a sub-property in all other cases

Note that these options aren't always mutually exclusive. It might be useful in some scenarios to have an explicit property for associating a general class of code, identifier, or other Literal Key with a resource, but also assign a specific datatype to the identifier as a cue for applications and reasoners

Related

- Typed Literal
- Literal Key

Index Resources

How can ordered collections or indexes be published as RDF?

Context

It is often the case that a web application will provide its users with a number of alternative ways to navigate. This might be an A-Z list of resources, or a list ordered based on creation date or "interestingness". These collections are useful resources in their own right that should be published as Linked Data.

Solution

Create resources for each "index" and associate the index with the items that it contains. An `rdf:List` or `rdf:Seq` is best used here as these structured include the notion of ordering that is an essential aspect of an index.

Example(s)

The BBC /music website publishes an A-Z list of artists. But this does not use a sequence to indicate ordering.

Discussion

By creating separate resources and links between these indexes and their contents we avoid the need to publish the algorithms that are used to generate the indexes, we can instead allow machines to navigate the collection just as a human user would do. This approach is particularly useful to ensure that data can be crawled by a semantic web search engine.

If a collection may be unmanageably large, then it can be decomposed into smaller groupings. E.g. a large A-Z index may be decomposed into smaller collections: A-C, D-F, etc.

By using an `rdf:List` or `rdf:Seq` to order the association between the index and its contents we retain the notion of ordering. Where the index is a permanent collection, with all contents known at the time of publishing, then an `rdf:List` is the best structure. Otherwise use an `rdf:Seq`. These RDF data structures can easily be generated and maintained using an asynchronous indexing system, avoiding the need to rebuild all the indexes whenever the underlying data changes. However for a large number of different indexes or variations the time spent building these indexes can be expensive.

Related

- Hierarchical URIs
- Composite Descriptions

Label Everything

How can we ensure that every resource has a basic human-readable name?

Context

A dataset may have a number of different entities, some of which are simple, e.g. people or organizations, whereas others are more conceptual or complex, e.g. an observation made at a particular point in time, under specific conditions. It may not always be clear to a developer, or a user exploring a graph in a browser, what a particular resource represents

Solution

Ensure that every resource in a dataset has an `rdfs:label` property

Example(s)

```
ex:Book
  rdfs:label "War and Peace".

ex:WeatherObservation
  rdfs:label "Rainfall measurement from Weather Station 1 recorded by Bob on 17th
  ex:rainfall 50;
  ex:date "2011-08-17"^^xsd:date
  ex:location ex:WeatherStation1;
  ex:experimenter ex:Bob.
```

Discussion

The `rdfs:label` property is a useful generic property for labelling any type of resource. By using this generic property to label any resource we can ensure that applications can easily discover a useful default label for a specific resource using a well-known property. This is particularly useful for supporting browsing of a dataset, as a browser can look for a default label. Developers can also use the label to assist in debugging queries or exploring a dataset.

Client applications may not always wish to use a provided label instead preferring to construct them based on other criteria. The Preferred Label pattern recommends using the `skos:prefLabel` property to communicate to clients a preferred label specified by the data publisher.

In some cases both a `rdfs:label` and a `skos:prefLabel` (or other specific labelling property such as `dcterms:title`) might be provided for the same resource. The content of the labels may differ reflecting the slightly different semantics of each property, e.g the `rdfs:label` might be longer or more descriptive than a shorter `skos:prefLabel`. If both label properties are provided with the same content, then this is an example of the Materialize Inference pattern: `skos:prefLabel` is a specialization of `rdfs:label`.

Related

- Preferred Label

Link Not Label

How do we model a dataset to maximise benefits of a graph based model?

Context

Most datasets are centred around a few core resources types that are central to that dataset. For example a social network dataset would be centred on people, groups and organizations, whereas a publishing dataset would be centred on authors, publications and publishers.

However every dataset typically has some additional resource types that are less "central". E.g areas or topics of interest, spoken or print languages, publication formats, etc. Often these resources are overlooked during modelling and are often only represented as simple literal values. This can make it less efficient to query a dataset, e.g. to group resources based on shared characteristics (e.g. everyone with same interests, all hardback books). It can also limit the annotate these aspects of a dataset, e.g. in order to add equivalence links.

Many common modelling approaches or industry standard models often re-inforce a less expressive modelling approach. For example many publishing and library standards, while encouraging use of controlled terms and authority files, focus largely on publications as the only important entity in a data model, leaving subject categories and authors as little more than labels associated with a work.

Solution

Ensure that all resources in a dataset are modelled as resources rather than simple literal values. Relate resources together to create a richer graph model. Use the literal values as the labels of the new resources.

A good approach is to look for any controlled vocabularies, keywords, tags, or annotations and dimensions in a dataset and model them as resources. Even structured literal values like dates might be more usefully modelled as resources.

Example(s)

Example of potential resources that might get overlooked:

- Languages
- Country codes
- Tags, categories, or subject headings
- Gender
- Genres
- Formats

A simple example:

#Rather than this:

```
<http://www.example.org/book/1>
  dc:format "Hardback";
  dc:lang "en"
  dc:subject "RDF", "SPARQL".
```

#Use a richer model:

```
<http://www.example.org/book/1>
  dcterms:format <http://example.org/formats/hardback>;
  dcterms:lang <http://www.lingvoj.org/lingvo/en>;
  dcterms:subject <http://example.org/category/rdf>;
  dcterms:subject <http://example.org/category/sparql>.
```

```
<http://example.org/formats/hardback>
```

```
rdfs:label "Hardback".

<http://example.org/category/rdf>
  rdfs:label "RDF".

<http://example.org/category/sparql>
  rdfs:label "SPARQL".

#Categories could later be related to other sources
<http://example.org/category/rdf>
  owl:sameAs <http://id.loc.gov/authorities/sh2003010124#concept>;
  owl:sameAs <http://rdf.freebase.com/ns/authority.us.gov.loc.sh.sh2003010124>.
```

Discussion

Creating a rich graph of relationships within a dataset will yield more flexibility and value from adopting Linked Data.

For example, as RDF triple stores are optimized for storing and querying relationships and graph patterns, creating resources for common dimensions in a dataset will allow for faster and more flexible querying. By representing these dimensions as resources in their own right, then they can be more easily annotated, e.g. to qualify them with additional data, or relate them to external sources.

In many cases existing resources in publically available datasets can be used instead of creating new URIs. By using resources, and reusing identifiers, it becomes easier to correlate and traverse different datasets. For example it becomes possible to draw in external data to enrich an existing application, e.g. an author profile or related works in another collection.

Related

- Annotation
- Equivalence Links

Further Reading

- Why Resources in Linked Data are Good

Multi-Lingual Literal

How can internationalized text be expressed in RDF?

Context

It is increasingly common for data to contain internationalized text, e.g. translated titles for a document. This alternate language text needs to be associated with the relevant resource in a clearly identifiable way

Solution

Multi-lingual versions of a literal property can be expressed as a simple Repeated Property with the addition of a language code to distinguish between the alternate versions.

Example(s)

```
_:greeting a ex:LoginGreeting; skos:prefLabel "Hello!"; skos:prefLabel "Hola!"@es.
```

Discussion

RDF allows a literal value to be associated with a language code. This code indicates the language in which the value of the literal has been expressed. RDF uses the ISO standard language codes, including regional variants. This capability lowers the bar to internationalizing data that is published as RDF. All serializations support this functionality and the SPARQL query language provides several functions for matching and manipulating language codes.

Related

- Repeated Property

N-Ary Relation

How can a complex relation, involving several resources, be modelled as RDF?

Context

An RDF triple expresses a relationship between at most two resources. However some relationships are not binary and involve the equal participation of several different resources. This is most common in the case of events, where the event is a relation between several resources, e.g. a Purchase is an n-ary relationship between a Person, a Product, and a Seller. All of those participants are key to the relationship.

Solution

Create a class for the relationship, and create instances of that resource to relate together the other resources that are involved in the relation.

Example(s)

```
ex:bob a foaf:Person.
ex:mary a foaf:Person.
ex:conferenceCentre a ex:Building.
ex:legoinc a foaf:Organization.

_:event1 a ex:Conference;
  dc:title "Lego Hack Day";
  ex:organizer ex:mary;
  ex:attendee ex:bob;
  ex:sponsor ex:legoinc;
  ex:location ex:conferenceCentre.
```

Discussion

The N-ary Relation pattern is similar to the Qualified Relation pattern as both involve the same basic solution: modelling a relationship as a resource rather than a property. They differ in their context. In the Qualified Relation pattern the desire is to annotate a relationship between two resources, whereas in the N-ary Relation pattern the goal is to represent a complex relation between several resources.

Related

- Qualified Relation

Further Reading

- Defining N-ary Relations on the Semantic Web [<http://www.w3.org/TR/swbp-n-aryRelations/>]

Ordered List

How do we specify an ordering over a collection of resources?

Context

Sometimes ordering is an important aspect of some collection of data. E.g. the list of authors associated with an academic paper, or the placings on a scoreboard. RDF offers a several different ways to capture ordering across some portion of a graph.

Solution

Use an `rdf:List` to describe a ordered list of resources.

Example(s)

```
<http://www.example.com/docs/1> ex:authors (
  <http://www.example.com/author/joe>
  <http://www.example.com/author/bob>
).
```

```
<http://www.example.com/author/joe>
  foaf:name "Joe".
```

```
<http://www.example.com/author/bob>
  foaf:name "Bob".
```

Discussion

RDF offers several modelling options for defining collections of resources. Formally these are the RDF Containers (Sequence, Bag, and Alternates) and the RDF Collections (List). For this purposes of this pattern an RDF Sequence and an RDF List are very similar: both describe an ordered list of resources.

Semantically the two structures differ in that a sequence is open ended (i.e. other members may exist, but aren't itemized) while a list is closed (i.e. the members are complete).

In practice though there is no real difference between the structures as data cannot be easily merged into an existing sequence, e.g. to append values. Both also suffer from being poorly handled in SPARQL 1.0: there is no way to query or construct an arbitrary sized list or sequence without extensions; SPARQL 1.1 property paths will remedy the querying aspects.

Related

- Repeated Property

Ordering Relation

How can we specify an ordering relationship between two or more resources?

Context

Ordering is important in many data models. Often that ordering can be implicit in the properties of resources, e.g. publication dates of Articles. In some cases the ordering needs to be more explicit, describing a fixed ordering relationship between the resources, e.g. a sequence of events, or stops on a delivery route.

Solution

Create ordering properties, e.g. "previous" and "next", for specifying the relationship between the ordered resources.

Example(s)

The following example describes the bus route, consisting of a number of bus stops. A stop may be preceded by a previous stop and may be followed by another stop. Custom relations have been defined to identify the previous and next relationships between the bus stops.

```
ex:bus10 a ex:Bus;
  ex:route ex:stop1.

ex:stop1 a ex:BusStop;
  ex:number 1;
  ex:nextStop ex:stop2.

ex:stop2 a ex:BusStop;
  ex:number 2;
  ex:previousStop ex:stop1;
  ex:nextStop ex:stop2.

ex:stop3 a ex:BusStop;
  ex:number 3;
  ex:previousStop ex:stop2.
```

Discussion

RDF provides several ways to implicitly and explicitly defining ordering in a dataset. Resources often have literal properties that implicitly define an ordering relationship between resources. E.g. time stamps on events, publication dates, or in the above example the numbers of each of the bus stops. Application code can use a query to extract this implicit ordering from the data. However its otherwise not defined in the model.

Another, more explicit way to define order is by using an RDF collection to create an Ordered List. Using an Ordered List, the above example could have been expressed as:

```
ex:bus10 a ex:Bus;  
  ex:route ( ex:stop1 ex:stop2 ex:stop3 ).
```

Lists are useful when there is no explicit relationship between the resources in the ordered collection. The above example has lost the fact that Bus Stop 2 is preceded by Bus Stop 1, other than their arrangement in a list. An application could still discover this relation using a query, but this can be difficult, and is certainly less explicit than the original example.

An ordering relation provides an explicit relationship between the ordered resources; in effect the ordering relations define a linked list that connects the resources together. Ordering relations are also simple to query and manipulate. For example, it would be very simple to adjust the ordered relation approach to modelling a bus route to add an extra stop, whereas the list based approach is more difficult as the entire list needs to be re-built requiring more changes to the graph.

The trade-offs between using an Ordering Relation versus an Ordered List depends on the specific data model. Continuing the above example, if the ordering of stops is generally fixed, then it is appropriate to using an Ordering Relation. However, if new bus routes are constructed by selecting from the available bus stops then an Ordered List may be more appropriate as there is no longer a fixed relationship between the stops.

Related

- Ordered List

Further Reading

- Defining N-ary Relations on the Semantic Web [<http://www.w3.org/TR/swbp-n-aryRelations/#pattern2>]

Preferred Label

How can a simple unambiguous label be provided for a resource?

Context

There are many different RDF properties that can be used for expressing a label, including generic properties such as `rdfs:label` and more domain specific labels such as `foaf:name`. This presents a variety of choices to the data provider, and can be confusing for application authors.

Solution

Use `skos:prefLabel` to publish the preferred display label for a resource

Example(s)

A library system publishes Linked Data about authors. The preferred mechanism for specifying author names is using a normalized form, e.g. Surname, First Name. This preferred label can be specified using `skos:prefLabel`, but the authors full name, and part names can be published using properties from FOAF. A simple Linked Data browser may use the preferred label, whereas an Address Book application browsing the same data may choose to assemble display labels according to user defined preferences, e.g. First Name, Surname.

Discussion

The `skos:prefLabel` property, whilst defined in the SKOS ontology, is a general purpose property that identifies the preferred label for a resource. By standardising on this property for labelling data providers and consumers can converge on a common mechanism for expressing labels of resources.

Having a single preferred property for display labels encourages convergence but doesn't preclude the use of more specific properties for other purposes. For example the literal value of the `skos:prefLabel` property can be formatted for display, leaving other properties, e.g. `rdfs:label`, `foaf:name`, etc to express alternatives or variations in labels or naming. A client that is aware of the meaning of specific predicates may choose to build labels using alternate logic, but having a label unambiguously specified simplifies application development.

Qualified Relation

How can we describe or qualify a relationship between two resources?

Context

In some cases relationships need to be qualified or annotated in some way. There are a number of different use cases that required this capability. E.g. to indicate the date when a specific relationship was made, or to indicate its source, or perhaps associate it with a probability or other similar qualifier. RDF only allows binary relations between resources, so how can these additional qualified relations be expressed?

Solution

Create a class for the relationship and create instances of that resource to relate together the resources that are involved in the relation. The relationship resource can then be annotated to qualify the relation further.

Example(s)

Marriage is a relationship that could be modelled as a simple relationship between two people. But that simplistic approach doesn't let us capture the date that the marriage started (or ended). Modelling the marriage as a relationship allows the relationship to be annotated:

```
eg:bob a foaf:Person.  
eg:mary a foaf:Person.
```

```
_:bobMaryMarriage a ex:Marriage;  
  ex:partner eg:bob;  
  ex:partner eg:mary;  
  ex:date "2009-04-01"^^xsd:date.
```

A diagnosis can be viewed as a relationship between a person and a disease. A diagnosis needs to be qualified with a probability. By creating a class to model the diagnosis explicitly, as well as additional properties for relating the diagnosis to a patient and a disease, it becomes possible to annotate the relationship with qualifying properties:

```
eg:bob a foaf:Person.  
  
eg:measles a ex:Disease.  
  
_:bobDiagnosis a ex:Diagnosis;  
  ex:patient eg:bob;  
  ex:disease eg:meases;  
  ex:probability "high";  
  ex:diagnostician ex:drhouse.
```

Discussion

Modelling relationships as resources works around the limitations of simple binary predicates. Creating a resource for the relationship allows much more flexibility in qualifying or describing the relationships between resources. Any number of additional properties may be used to annotate the relation. When the relationship is between two resources we refer to it as a qualified relation, when it is between several resources, each of which are equally involved in the relationship then we have an N-Ary Relation.

If modelling relationships as classes is useful, then why not use this pattern for all non-trivial relationships in a model? The main reason is that it causes explosion in the number of terms in a vocabulary, e.g. each predicate is replaced with two predicates and a class. A vocabulary can quickly become unwieldy, so the value of the extra modelling structure needs to be justified with clear requirements for needing the extra complexity. As described here, the primary reason is to qualify the relation.

The only alternative to this approach would be to have another independent property whose value is intended to be interpreted alongside one or more other properties of the same resource, e.g.:

```
eg:bob a foaf:Person.  
eg:mary a foaf:Person.  
  
eg:bob ex:partner eg:mary.  
eg:bob ex:weddingDay "2009-04-01"^^xsd:date.  
eg:mary ex:weddingDay "2009-04-01"^^xsd:date.
```

In the above alternative the marriage relationship between two people is expressed using the `ex:partner` relation and the date of the wedding with the separate `ex:weddingDay` property. On the surface this seems simpler but loses flexibility and clarity. Firstly there could be a large number of additional properties associated with the marriage relation, all of these would need to be added to the model, and applications would need to know that this collection of properties were all related in some way (i.e. were about a marriage). It also fails to model divorces and re-marriages. Adding a relation resource deals with this better as we have greater clarity in the model about the relationship and its specific properties.

There are a number of cases where adding resources into a data model in this way can aid expressivity, understanding when and when not to apply the pattern is an important part of RDF modelling.

Related

- N-Ary Relation

Reified Statement

How can we make statements about statements?

Context

The foundation of the RDF model is the triple. This simple structure can be combined to create complex descriptions that can support any kind of data model. But in some circumstances there may be a need to annotate an individual triple, e.g. to indicate when the statement was made, or who by. How can this be achieved?

Solution

Use a technique known as reification and model the triple as a resource, with properties referring to its subject, predicate and object.

Example(s)

```
_:ex rdf:type rdf:Statement;  
  rdf:subject <http://www.example.com/book/1>;  
  rdf:predicate <http://xmlns.com/foaf/0.1/maker>;  
  rdf:object <http://www.example.com/author/joe>;  
  dc:created "2010-02-13".
```

Discussion

Reification is a much maligned technique but has its role to play in RDF modelling. Understanding its limitations allows it to be used where appropriate and avoided when other techniques like Named Graphs are a better fit.

RDF triples cannot be directly annotated. Reification is a modelling approach to dealing with this that involves changing the structure of the model. Each triple that needs to be annotated is instead modelled as

a resource in its own right; an instance of the `rdf:Statement` class. The statement resource then has subject, predicate, and object properties that describe the original triple; additional properties can then be added to annotate the statement.

The obvious limitation of this approach is that the data model no longer contains a simple set of triples, instead it contains *descriptions of triples*. The triples themselves have not been explicitly asserted. This makes a reified model harder to query, more difficult to manipulate, and at least three or four times larger in terms of number of triples.

In contrast to reification the Named Graph pattern offers a data management approach to annotating triples or sets of triples, by associating them with a URI which can itself be the subject of additional assertions.

So where is reification best used? Current practice seems to suggest that reified statements are best used as a technique for describing changes to the structure of a graph, e.g. statements that have been added or removed, along with additional properties to indicate the time of the change, etc. In other circumstances it is best used with a reasoner or rule engine that can support surfacing the original triples, thereby simplifying querying. But this still comes with a cost of increased storage as well as potentially performance issues.

Related

- Named Graphs

Repeated Property

How can properties that have multiple values be expressed?

Context

A resource may have multiple values for a specific property. E.g. a set of keywords associated with a research paper. RDF offers several options for modelling these multi-valued relations. Sometimes these multi-valued relations have an explicit ordering that should be preserved in the data, e.g. the order of authors. In other cases an explicit ordering is not required

Solution

Simply repeat the property of the resource multiple times.

Example(s)

For example a research paper may have multiple keywords. The ordering of these keywords is not important, so using the Dublin Core subject property, we can express this multi-valued relation as:

```
_:doc
  dc:subject "RDF" ;
  dc:subject "Semantic Web".
```

Discussion

Repeating properties is the simplest approach to handling multi-valued relations. The alternatives all have their downsides. One alternative would be to use a structured value for the literal, e.g:

```
_:doc dc:subject "RDF, Semantic Web".
```

But structured values limit the ability for applications to query for specific data items, e.g. documents that have a specific keyword. With a structured value a regular expression would be required to test for the values in the literal.

Another alternative would have been to use an RDF Container (Sequence, Bag, or Alt) or Collection (RDF List). Some of these structures imply ordering (Sequence, List) while others (Bag, Alt) don't imply an ordering over their members. This means that the keywords could have been expressed as:

```
_:doc dc:subject ( "RDF", "Semantic Web" )
```

There are several downsides to using Containers and Collections. Firstly they can be difficult to use as they can be hard to query. Secondly, when combining data from different sources the Containers and Collections won't be merged together to create a single structure; the lists will remain separate.

On the whole, the simplest way to express multi-valued relations is to use a simple Repeated Property. Collections and Containers are best used only when ordering is significant.

Related

- Ordered List

Topic Relation

How can a web page or document be associated with a resource?

Context

There is often a need to associate a resource with a page or document that provides further, human-readable content about the resource. This information may often be present in a content management system or dataset as "further reading" links. How can these relations be published as RDF, clearly identifying the relationship between the document and the resource being described?

Solution

Use the `foaf:topic` and/or `foaf:primaryTopic` relationships to associate a resource with a page(s) in which it is featured or discussed

Example(s)

In a dataset describing Paris, some additional further reading links may be available:

```
<http://www.example.org/place/Paris> a ex:Place;  
  skos:prefLabel "Paris"@fr.
```

```
<http://travel.example.com/blog/a-paris-guide>
  foaf:primaryTopic <http://www.example.org/place/Paris>.

<http://travel.example.com/blog/top-europe-cities>
  foaf:topic <http://www.example.org/place/Paris>.
```

Discussion

There will always be a mix of unstructured documents and structured data on the Semantic Web. There are many kinds of relations that could be expressed between a document and a resource, but a very common relation is one of categorisation, e.g. identifying when a document or page has a specific focus or theme. The FOAF topic terms, `foaf:topic` and `foaf:primaryTopic` provide a means to link a document directly with a resource which features as a topic in that document.

Other vocabularies provide similar related properties that may be more useful in some scenarios. For example Dublin Core defines the `dc:subject` property, but this is best used to relate a document to an entry in a controlled vocabulary, e.g. a subject heading, rather than a "real world" resource.

Because the Topic Relation pattern involves making statements about other resources on the web, it is an application of the Annotation pattern.

Related

- Annotation

Typed Literal

How can a datatype be associated with an RDF literal?

Context

Data items are generally of a specific type, e.g. a date, floating point value, or decimal. This type information is important for consuming applications as it provides flexibility when querying, formatting, displaying and converting data.

Solution

Always associate a data type with an RDF literal that contains a structured value.

Example(s)

TODO

Discussion

The RDF allows a literal value to be associated with a data type. RDF itself does not have a built-in type system, it defers to the XML Schema datatypes to define a useful common set of data types and their legal lexical values. By using this facility data publishers can ensure that consumers can more easily manipulate and process the published data. Use of standard data types encourages interoperability between systems. It

also supports internationalization of data as client applications can more easily process the value to present it for display in a specific locale

In some cases XML Schema does not define an existing data type. It is therefore common practice to define a Custom Datatype

Related

- Custom Datatype

Chapter 4. Publishing Patterns

Abstract

This chapter introduces a collection of patterns that all relate to the publishing of Linked Data. In some cases these patterns provide guidance on additional aspects of modelling, in others they describe approaches to surfacing data on the web.

Annotation

How can data about third-party resources be published as part of a dataset?

Context

Datasets are rarely completely self-contained. They will often contain facts or data about third-party resources. These may be entities from other systems or existing web documents that are part of an external system or website. It should be possible to surface this information alongside the data that originates in the dataset.

Solution

Just publish RDF documents containing the statements about the external resources

Example(s)

Linked Data available from `http://www.example.org/author/john` when retrieved might contain the following data which contains annotations about two additional resources:

```
<http://www.example.org/author/john> a foaf:Person.  
  
<http://wiki.example.net/page/UbuntuTips>  
  dc:title "Ubuntu Tips";  
  dc:creator <http://www.example.org/author/john>.  
  
<http://publisher.example.org/authors/1234>  
  owl:sameAs <http://www.example.org/author/john>.
```

Discussion

With RDF Anyone can say Anything Anywhere, there are no restrictions about who may make statements about a resource; although clearly a processing application might want to pick its sources carefully.

It is entirely consistent with the Linked Data principles to make statements about third-party resources. While in some cases it is useful to use the Proxy URIs pattern, in many cases simply making statements about external resources, whether these are entities (e.g. a person) or documents (e.g. a web page) is sufficient.

Related

- Proxy URIs
- Topic Relation

Autodiscovery

How can people find the underlying linked data for a given web page?

Context

Currently the Web is predominately built from interlinked HTML pages. Linking directly to linked data documents from HTML presents the risk of confusion for non-technical audiences. However the publisher requires that the underlying data be discoverable by linked data aware tools and indexable by search engines.

Solution

When publishing a web page derived from linked data include a link element in the head of the web page pointing to the original data.

```
<link rel="meta" type="application/rdf+xml" title="Raw Data"
href="http://example.com/data.rdf" />
```

Example(s)

The FOAF Vocabulary recommends linking a homepage to an equivalent FOAF profile using the link element.

The Semantic Radar Firefox plugin uses the autodiscovery pattern to detect the presence of linked data related to the web page the user is viewing.

Discussion

Until web browsers become fully linked data aware it may not be satisfactory to link directly to linked data pages from the body of an HTML page. HTML provides the link element to allow publishers to include links to information and resources that may be relevant in addition to the main content of the page. Web browsers may choose to display these links in their user interface. Web search engines can use these links to discover additional information that may make their search more relevant for the user.

Related

- Primary Topic Autodiscovery

Document Type

How can some context be provided about a set of RDF triples published to the web?

Context

While the web of Linked Data is, in its broadest sense, a set of RDF triples, there are often circumstances in which it is useful to describe a smaller grouping of triples. RDF statements are published to the web as documents conforming to a particular syntax, e.g. RDF/XML, Turtle, or XHTML+RDFa. These documents may be directly inter-linked using See Also relations. To enable user agents to select between links it is useful to indicate the type of document which a link is referencing.

Solution

Define a document type describing a conceptual or physical grouping of triples. Indicate where a specific document is of a particular type, including a Topic Relation such as `foaf:primaryTopic` to relate the document to the resource(s) it is describing.

Example(s)

```
#document type
foaf:PersonalProfileDocument a foaf:Document.

#specific instance of document, with indication of its topic
<http://www.example.org/doc/john> a foaf:PersonalProfileDocument;
    foaf:primaryTopic <http://www.example.org/doc/john#me>.

<http://www.example.org/doc/john#me> a foaf:Person;
    foaf:name "John".
```

Discussion

XML is a document format and XML schemas describe the valid structure of documents. In contrast RDF is defined in terms of sets of triples and schemas are used to support inferencing and description of data structures. It is often useful to describe specific collections of triples. For example within a triple store it is often useful to group triples into Named Graphs. These collections can be usefully annotated in various ways, e.g. indicating their provenance, creation date, origin, etc.

Collections of triples may be published to the web using a variety of syntaxes and mechanisms. It can be useful to partition data into a number of different documents, e.g. to simplify the publishing process or usefully present data to user agents. By annotating the documents to indicate their type we can usefully allow user agents to select specific collections that are more likely to contain information of interest. This can help target crawler behaviour or prioritise documents for de-referencing.

Using document types does not imply that a user agent can make assumptions about the structure or format of the data that will be retrievable. The document may contain information about any number of different resources, or use any RDF syntax.

Two well deployed examples of document typing in use today are RSS 1.0 and FOAF. RSS 1.0 is defined as both an XML and an RDF vocabulary and as such has a strict definition of document that aligns with its use in an XML context. FOAF however is an RDF vocabulary, but has still found it useful to define the notion of a `foaf:PersonalProfileDocument` which indicates that a document primarily describes a particular person (but may include additional data).

The Document Type pattern is most commonly used in conjunction with the See Also and Annotation patterns. It could also usefully be applied when referencing a Link Base, allowing a user agent to more easily discover Equivalence Links related to a specific resource(s).

Related

- Named Graphs
- See Also

Edit Trail

How can people be encouraged to improve or fix up open data?

Context

Community maintained or managed information sources often underpin a particular set of data. However by publishing the data in a machine-readable form the data may become dissociated from the community tools or forum that is used to create and maintain the data. Without a clear association from the data to the tools, users may be discouraged from using due to a perception of poor quality, when it is often within their means to fix up or improve the data.

Solution

Use the `ex:editform` property to associate a resource with a form, or other entry point into an application that can be used to correct the raw data.

Example(s)

Associating a dbpedia resource page with the wikipedia page from which its data is derived.

Discussion

The "Edit This Page" concept is well embedded into a number of community maintained websites, and particularly those based on a wiki platform. A convention has grown up on those sites that involves providing a machine-readable link in the HTML page that allows a user agent to provide the user with a way to auto-discover links to the associated editing form.

The `ex:editform` property applies this concept to Linked Data resources. In this case the association is from the resource to editing form, allowing a more direct link from to be made from an information resource to a data management tool that can be used to edit its properties.

By associating the property directly with the resource, as opposed to the related page of which it is a primary topic, we aim to ensure that even the link remains even if only a fragment of the dataset is re-published.

Embedded Metadata

How do we add structured data to an existing document or file?

Context

There are two related aspects to this pattern

Firstly, many web sites that are using frameworks or content management systems that are either difficult to customize or are operated by organizations that have little or no in-house technical staff. How can these sites be updated, with a minimum of effort, to support publishing of structured data?

Secondly, documents or files may be shared and copied across the web. This can result in a broken connection between the document and it's metadata which might only be available from the source website. How can the metadata be made available to anyone discovering a copy or mirror of the document?

Solution

Embed the structured data directly in the document itself rather than, or in addition to, publishing that data separately.

The most common scenario here is instrumenting an existing webpage to add some "semantic markup" that will enable a client to extract data directly from the HTML source. This typically involves changing just the templates used to generate the web pages. By changing a small number of resources, it becomes possible to quickly and easily publish data about a large number of resources.

A less common scenario involves embedding data within a different document format. Typically this relies on using an existing extension mechanism that has been defined for that format. A tool may then inspect the file directly to discover the metadata or a link to its location.

Example(s)

At the time of writing there are a number of competing proposals for embedding metadata in XHTML/HTML documents, including RDFa, microdata and microformats. RDFa can also be used to embed metadata in other XML formats

Options vary for embedding metadata in other formats, but Adobe XMP provides an option that can be used in a variety of formats.

Discussion

Embedding metadata into existing resources, rather than requiring changes to how content and data is published to the web is often much easier to achieve. At the time of writing there are a number of competing approaches for embedding metadata in XHTML/HTML. These typically offer the same basic features but vary in how much they require specific markup extensions in the original document.

While it can be very easy to quickly instrument a website with embedded metadata, the requirement to introduce all data into the same template means that it can become awkward to manage the competing concerns: e.g. clean, well-structured markup for styling and presentation, versus rich, detailed, semantic markup for machines. For more complex use cases it may be better to simply publish the metadata separately, or embedded only a subset of the data with links to additional resources.

Related

- Primary Topic Autodiscovery
- See Also

Equivalence Links

How do we indicate that different URIs refer to the same resource or concept?

Context

Linked Data is published in a de-centralised way with multiple people and organisations collectively publishing data about the same resources. While it is a goal to reuse identifiers wherever possible this may not always be achievable. If data has been published in this way, how can links be built between datasets in order to identify that distinct URIs refer to the same resource?

Solution

Use `owl:sameAs`, `skos:exactMatch`, or similar specialised relation, to indicate that two URIs are equivalent.

Example(s)

```
#Statement from Data Set 1
ex:bob a foaf:Person.
```

```
#Statement from Data Set 2, with equivalence
ex:otherBob a foaf:Person.
ex:otherBob owl:sameAs ex:bob.
```

Discussion

Distributed publishing is a fact of life on the web. Semantic web technologies have built-in support for handling distributed publishing of both data and schemas by standardising specific relations for bringing together disparate datasources. The most important of these facilities is the `owl:sameAs` relation, which indicates that two URIs are equivalent. According to the semantics defined in the OWL standard, the two URIs are synonyms of each other and all statements made about one of those URIs is also considered to be true of the other.

With support from a reasoner the `owl:sameAs` relation allows a semantic web application to query and interact with a single view across any number of datasets. This allows data to be integrated, without requiring any merging or re-processing of the data. It also avoids the need for up-front convergence on standard identifier schemes.

However `owl:sameAs` is only one possible equivalence relation that could be stated between resources. Other vocabularies may choose to define additional relationships that have less strong semantics associated with them. For example SKOS defines two additional properties, `skos:closeMatch` and `skos:exactMatch`, that can be used to state equivalences between concepts in a thesaurus. The relations allow for more fuzzy notions of equivalence and have weaker semantics: `skos:exactMatch` declares two concepts to be the same but doesn't imply that all statements about one concept are also true of another.

OWL also defines some additional specialised equivalence relations for relating together classes (`owl:equivalentClass`) and properties (`owl:equivalentProperty`). These should be used instead of `owl:sameAs` when relating together terms in an ontology.

Related

- Proxy URIs

- Link Base

Link Base

How can outbound links from a dataset be managed separately from the core data?

Context

When publishing Linked Data it is common to undertake the inter-linking of newly published data with existing sources as a final step in the publishing process. Linking strategies range from manual methods through to automated link discovery. The volume of out-bound links may vary over time, e.g. as new links are discovered to existing sources, or new links are made to other new datasets. Over time, due to "semantic drift", it may be necessary to remove links to some external resource.

The linking data within a dataset may therefore have different quality assurances associated with it, as well as a different rate of change to the core dataset.

Solution

Partition the data into two datasets: the core data and the linking data. Publish linking data as a separate set of documents. Use See Also links to tie the data together.

Example(s)

```
#Core Data
<http://www.example.org/places/Paris> a ex:Place;
  skos:prefLabel "Paris";
  rdfs:seeAlso <http://www.example.org/links/Paris>.

#Linking Data
<http://www.example.org/links/Paris>
  owl:sameAs <http://dbpedia.org/resource/Paris>.
```

Discussion

Partitioning links from core data has a number of advantages in terms of both how it is published and how it is consumed.

For the publisher, partitioning links allows the linking data to be revised and improved separately to the publication of the core reference data. This is particularly useful where the linking data is generated automatically using heuristics. The quality of the generated links may need to be refined and iterated over time. As new datasets are published, the linking data can get updated to include additional relationships. Managing the typically smaller subset of links distinctly from the core data, e.g. in a separate triple store, allows more flexibility in managing the data.

For a consumer, the partitioning of links into a separate dataset allows more choice in what data is to be consumed. Because the quality characteristics of linking data may vary considerably from that of the core data, the ability to selectively consume only the core data is an important one. If the linking data

is presented as part of the core dataset, then this requires the consumer to filter the data to remove the unneeded triples.

Related

- Equivalence Links

Materialize Inferences

How can data be published for use by clients with limited reasoning capabilities?

Context

Linked Data can be consumed by a wide variety of different client applications and libraries. Not all of these will have ready access to an RDFS or OWL reasoner, e.g. Javascript libraries running within a browser or mobile devices with limited processing power. How can a publisher provide access to data which can be inferred from the triples they are publishing?

Solution

Publish both the original and some inferred (materialized) triples within the Linked Data.

Discussion

Reasoners are not as widely deployed as client libraries for accessing RDF. Even as deployment spreads there will typically be processing or performance constraints that may limit the ability for a consuming application to perform reasoning over some retrieved data. By also publishing materialized triples a publisher can better support clients in consuming their data.

Most commonly materialization of the inferred triples would happen through application of a reasoner to the publishers data. However a limited amount of materialized data can easily be included in Linked Data views through simple static publishing of the extra relations. E.g. by adding extra "redundant" statements in a template.

There are a range of useful relationships that could be included when implementing this pattern:

- Typing, based on range and domains of properties and/or derived classes
- Super properties, based on property extensions
- Transitive relationships, e.g. `skos:broader` and `skos:narrower` relations
- Inverse properties

Materialization may also be targetted in some way, e.g. to address specific application needs, rather than publish the full set of inferred relations. The specific materializations chosen can be based on expectations of common uses for the data. For example the publisher of a SKOS vocabulary may publish transitive relations between SKOS concepts, but opt not to include additional properties (e.g. that every `skos:prefLabel` is also an `rdfs:label`)

A reasonable rule of thumb approach to materializations would be to always include additional type or property relations whenever that would help ground the data in more commonly used vocabularies. Inverse

and transitive relationships provide extra navigation options for Linked Data clients so are also worth considering.

The downside to publishing of materialized triples is that there is no way for the consuming system to differentiate between the original and the inferred data. This limits the ability for the client to access only the raw data, e.g. in order to apply some local inferencing rules. This is an important consideration as publishers and consumers may have very different requirements. Clearly materializing triples also places additional burdens on the publisher.

An alternative approach is to publish the materialized data in some other way, e.g. in a separate document(s) referenced by a See Also link.

Further Reading

- Creating Linked Data - Part V: Finishing Touches [<http://www.jenitennison.com/blog/node/139>] (see section on Derivable Data)

Named Graphs

How do we organize and annotate a set of RDF statements?

Context

The RDF model treats triples as a single collection of statements. For many scenarios it is useful to have some additional structure or grouping to the data. E.g. to identify all triples from a particular source, or created on a specific date. It is useful to be able to describe and annotate these collections just like any other resource.

Solution

Use URIs to identify collections of statements. Where necessary associate a triple with this URI to create a *quad*

Discussion

RDF triple stores have for some time allowed sets of statements to be grouped into identifiable sets. Using a URI for this identifier means that such named graphs can themselves be described using RDF. E.g. to say when they were created, how they were authored, etc. SPARQL 1.0 has, to some extent, standardised this capability, allowing queries to be made across all or selected named graphs. Syntaxes for serialising named graphs also exist, allowing quads to be exchanged between systems. These syntaxes include NQuads, TriX, and TriG.

By essentially extending the core RDF model from a triple to a quad, Named Graphs provide an extra degree of freedom in managing RDF datasets. This means that they are an increasingly important part of the RDF toolset, supporting a number of different application scenarios as well as a more manageable alternative to reification for handling versioning and provenance in datasets.

However there is no agreed meaning for this extra data element. The RDF data model describes sets of triples, but doesn't standardise any method of referring to or labelling a set. So in one application the graph identifier could be a simple label used to identify a set of triples, whereas in another it may identify the source URI from which some data was retrieved.

Some recorded uses for Named Graphs include:

- *Tracking provenance of RDF data* — here the extra URI is used to track the source of the data; especially useful for web crawling scenarios
- *Replication of RDF graphs* — triples are grouped into sets, labelled by a URI, that may then be separately exchanged and replicated
- *Managing RDF datasets* — here the set of triples may be an entire RDF dataset, e.g. all of dbpedia, or all of musicbrainz, making it easier to identify and query subsets within an aggregation
- *Versioning* — the URI identifies a set of triples, and that URI may be separately described, e.g. to capture the creation & modification dates of the triples in that set, who performed the change, etc.
- *Access Control* — by identifying sets of triples we can then record access control related metadata

Related

- Reified Statement

Primary Topic Autodiscovery

How can people identify the principal subject of a given web page?

Context

Often a web page is concerned with a particular physical object or other resource. To assist discovery and aggregation of pages about particular topics the publisher wants to indicate the URI of this resource.

Solution

When publishing a web page include a link element in the head of the web page pointing to the URI of the page's primary topic. Use a rel attribute value of "primarytopic".

```
<link rel="primarytopic" href="http://dbpedia.org/resource/London"/>
```

Example(s)

Associating a wikipedia page with the equivalent dbpedia resource.

Discussion

Many pages on the Web are explicitly about a single subject. Examples include Amazon product pages, Wikipedia entries, blogs and company home pages. Without an explicit link, content aggregators must resort to heuristic inference of the topic which is prone to classification error. Often the original publisher knows the specific topic and would like to provide this as a hint to aggregators and other content consumers.

Even when the page is about several topics there can be a single primary topic that can be linked to directly.

Related

- Autodiscovery

Progressive Enrichment

How can the quality of data or a data model be improved over time?

Context

At the time when a dataset is first published the initial data may be incomplete, e.g. because data from additional systems has not yet been published, or the initial dataset is a place-holder that is to be later annotated with additional data. Data models are also likely to evolve over time, e.g. to refine a model following usage experience or to converge on standard terms.

Solution

As the need arises, update a dataset to include additional annotations for existing or new resources.

Discussion

A key benefit of the semi-structured nature of RDF is the ability to easily merge new statements into an existing dataset. The new statements may be about entirely new resources or include additional facts about existing resources. There is no need to fully define a schema, or even fully populate a data model, up front. Data can be published and then refined and improved over time.

Progressive Enhancement is essentially a variant of the Annotation pattern within a single dataset. Whereas the Annotation pattern describes an approach to distributed publishing of data about a set of resources, Progressive Enhancement confines this to a particular dataset allowing the depth of detail or quality of the modelling to improve over time.

A common use of this pattern in Linked Data publishing is to update a dataset with additional Equivalence Links.

Progressive Enrichment is a key aspect of the Blackboard application pattern.

Related

- Annotation
- Equivalence Links

See Also

How can RDF documents be linked together to allow crawlers and user agents to navigate between them?

Context

Linked Data is typically discoverable by de-referencing URIs. Starting with a single URI a user agent can find more data by discovering other URIs returned by progressively retrieving descriptions of resources referred to in a dataset. However in some cases it is useful to provide additional links to other resources or documents. These links are not semantic relations per se, just hypertext links to other sources of RDF.

Solution

Use the `rdfs:seeAlso` property to link to additional RDF documents.

Example(s)

The Linked Data published by the BBC Wildlife Finder application includes data about ecozones. The data about an individual ecozone, e.g. the Nearctic Ecozone [http://www.bbc.co.uk/nature/ecozones/Nearctic_ecozone.rdf] refers to the habitats it contains and the species that live in that ecozone. A semantic web agent can therefore begin traversing the graph to find more related data. The RDF document returned from that URI also includes a `seeAlso` relationship to another document that lists all ecozones.

Discussion

The `rdfs:seeAlso` relationship is intended to support some hypertext links between RDF documents on the web. There are no explicit semantics for the property other than that a user agent might expect to find additional, relevant RDF data about a resource at the indicated location. Using this relationship allows documents to be linked together without requiring semantic relations to be specified between resources where none exists.

By ensuring that data from a Linked Data site is robustly linked together, it helps semantic web crawlers and user agents to traverse the site to find all relevant material. The `rdfs:seeAlso` relation is therefore well-suited for publishing simple directories of links for a crawler to follow.

The relation can also be used to refer to other documents on the web, e.g. published by third-parties, that may contain additional useful Annotation data.

Related

- Annotation

Chapter 5. Application Patterns

Abstract

This chapter introduces a collection of patterns that all relate to the building of applications over RDF and/or Linked Data

Assertion Query

How can a dataset be tested for known patterns?

Context

There are a number of circumstances in which it is useful to test for patterns in a dataset. The most common is likely to be discovery of whether there is data available about a specific resource or from a specific vocabulary. Another very common use of pattern detection is to validate a dataset to check that it conforms to a specific structure. The latter is very common when testing the data generated by an application, e.g. during development or ongoing maintenance, or to check data received from third-party systems.

Solution

Use a SPARQL ASK or CONSTRUCT query to probe the dataset for known patterns

Example(s)

The following will return true if the specified pattern is found:

```
#Is there any data about a specific resource?
ASK WHERE {
  <http://www.example.org/person/bob> ?p ?o.
}
```

Discussion

As described in the Transformation Query pattern, SPARQL provides a declarative syntax for certain kinds of operations on RDF graphs. The SPARQL ASK query form is intentionally provided to support making assertions against an RDF dataset. It is therefore useful in the role of testing the output of an application (e.g. for unit testing) or acceptance testing incoming data received from a third-party.

In some cases it is useful to run a number of queries against a dataset and generate suitable warning messages if any of the assertions succeed. Instead of using an ASK query, a CONSTRUCT query can be used instead, allowing a simple message to be generated as output. These messages can then be used to, e.g. produce a test report from a dataset:

```
CONSTRUCT {
  _:msg ex:message "Every person should have a name".
}
```

```
}  
WHERE {  
  ?person a foaf:Person;  
  OPTIONAL {  
    ?person foaf:name ?name.  
  }  
  FILTER (!bound(?name))  
}
```

Related

- Transformation Query

Further Reading

- The Schemarama [<http://isegserv.itd.rl.ac.uk/schemarama/>] testing framework uses the CONSTRUCT variant of this pattern.

Blackboard

How can the task of compiling or constructing a dataset be divided up into smaller tasks?

Context

Applications consuming Linked Data commonly need to compile a dataset by collecting data from a number of distributed sources, e.g. other Linked Data datasets, SPARQL endpoints, or converting data from other legacy systems. A monolithic approach to constructing an aggregated dataset can be slow, fragile, and complex to implement. Dividing up tasks into smaller units of work can help parallelize the compilation of the data.

Solution

Create a number of separate processes that are each specialised to a particular task. Each process runs independently from each of the others, allowing each to be as simple or as complex as necessary. The processes all co-ordinate in the Progressive Enrichment of a single dataset, but are not triggered in any particular sequence. Instead each process looks for specific patterns in the underlying dataset, triggering the processing of the data, the results are then written back to the dataset.

Example(s)

An application is written to monitor twitter for uses of a particular hash tag. The application provides a simple RDF view of each tweet, including its author, text, timestamp. The RDF is written into a shared RDF dataset that serves as the Blackboard for a number of other processes. Additional processes are then written that query the dataset to discover new tweets. These processes carry out discrete tasks. The results of the tasks may trigger further processing by other tasks, or may be used to directly drive specific application behaviour. The tasks may include steps such as:

- Looking for profile information about the author of each tweet
- Attempting to find a FOAF profile for each author

- Extracting hash tags from new tweets
- Extracting URLs from individual tweets
- Discovering metadata associated with URLs mentioned in tweets

Discussion

The Blackboard pattern is an existing design pattern that has been used in a number of existing systems, it works well when applied to the task of aggregating RDF and Linked Data due to the ease with which a dataset can be enriched over time.

The RDF dataset used as the "blackboard" shared by the independent processes may be short-lived, e.g. an in-memory datastore used to respond to a particular request or task, or permanent, e.g. an on-going aggregation of data on a particular topic or theme.

The decomposition of the data aggregation & conversion tasks into smaller units makes it easier to explore different approaches for implementing the desired behaviour, e.g. to explore alternate technologies or algorithms. The overall result of the processes co-operating to compile and enrich the dataset can be extremely complex but without requiring any overall co-ordination effort. Additional processes steps, e.g. to acquire data from additional sources, can easily be added without impacting on the overall system, making the architecture extensible.

Related

- Progressive Enrichment
- Parallel Loading

Bounded Description

How can we generate a useful default description of a resource without having to enumerate all the properties or relations that are of interest?

Context

Application using semi-structured data sources should be tolerant of discovering unexpected data or missing properties of resources. For applications to be able to achieve this, there needs to be an approach to generating useful default descriptions of resources that don't require enumerating every property of interest. This behaviour is particularly useful for Linked Data browsers or similar applications that can have little or no expectation as to with which datasets they may be interacting.

Solution

Extract a sub-graph, or "bounded description", from a dataset that contains all of relevant properties and relationships associated with a resource.

Discussion

Bounded Descriptions take advantage of the graph structure of RDF in order to define simple graph operations that can be applied to any node in the graph. The operations yield a useful sub-set of the properties associated with the resource based on how they relate to the resource, rather than the specific RDF predicates that have been used.

There are a number of different types of bounded description that are in common use:

- *Datatype Property Description* -- retrieve all properties of a resource whose values are literals
- *Object Property Description* -- retrieve all properties of a resource whose values are resources, typically eliminating blank nodes
- *Concise Bounded Description* -- effectively the above two descriptions, but recursively include all properties of any blank nodes present in object properties
- *Symmetric Concise Bounded Description* -- as above but include statements where the resource being described is the object, rather than the subject

Many different variations of these basic descriptions are possible, especially when additional filtering is done to include, for example, properties that are useful for labelling (in a user interface).

In practice many common web application use cases can easily be fulfilled with one or more bounded description queries. The ability to use general purpose queries to build a user interface or otherwise drive application behaviour increases cacheability of the results: an application may end up using a small number of relatively general purpose queries that apply to a number of use cases.

Bounded descriptions can be implemented using SPARQL CONSTRUCT queries. SPARQL DESCRIBE queries are implemented using a Bounded Description that is built-in to the specific SPARQL processor being used. The most common approach is to use a Concise Bounded Description.

Further Reading

- Bounded Descriptions in RDF [http://n2.talis.com/wiki/Bounded_Descriptions_in_RDF]
- Concise Bounded Description [<http://www.w3.org/Submission/CBD/>]

Composite Descriptions

How do we declare the underlying dataset for a page involving custom subsets or views of the data?

Context

When integrating data from heterogenous sources it is sometimes necessary to synthesise page URIs non-algorithmically from the underlying data. Alternatively views of data may be required that follow a clustering or structure that does not have a simple 1:1 correspondence with underlying data URIs.

Solution

Create data about your description pages and include foaf:topic and foaf:primaryTopic properties to link the page to the resources that it describes. When rendering these pages obtain the data describing the page then bring in descriptions of each resource referenced with foaf:topic and foaf:primaryTopic to build the base dataset for the page.

Example(s)

The BBC programme pages include information on a primary topic supplemented with additional data about other related topics. The data included on each page may vary depending on factors other than the type of resource being described.

Discussion

Most database driven pages on the Web involve more than one type of data and augment a base dataset with related information by using multiple database queries. In many cases it is possible for the publisher to anticipate these arrangements and describe them as linked data. This can reduce multiple queries to a single query describing the resource and any related resources. The resulting dataset can be passed directly to a templating system for rendering.

The topics associated with a page do not need to be closely related in the underlying data or even connected at all. The page description gathers together a group of resources according to the precise context specified by the publisher without reliance on particular relationships pre-existing in the data.

Changing the level of detail for classes of page or even of specified individual pages can be done simply by updating the description of those pages and allowing the templating system to work with the new dataset.

An additional benefit is that the page structure of the site can also be made queryable so it would be possible to discover which pages include information about a specific subject, thereby presenting the possibility of automatic cross-linking.

Follow Your Nose

How do we find additional relevant data from the web?

Context

When we retrieve some data from a URI, we are unlikely to have obtained all of the relevant data about that resource. Additional statements or useful extra context may be available from both the original source, as well as other third-party sources on the web.

Solution

Identify additional useful links within the available data, and then de-reference those URIs in turn to find the additional data.

Example(s)

The BBC Wildlife Finder application exposes data about biological species. By following links within the data we can find additional information about related species or its habitat from within the same dataset. By following links to dbpedia or other sources, we can find additional detail on the biology and distribution of the species.

Discussion

This pattern is at the core of the Linked Data approach. By giving resources URIs we make them part of the web, allowing a description of them to be retrieved by a simple HTTP GET request. By linking to other resources on the web, we allow applications to find more information by repeatedly following links to crawl the additional sources.

There are two main types of links that could be followed in an RDF graph:

- URIs of other resources -- See Also links to further documents, or URIs of other related resources. Follow these links to find more data.

- URIs of RDF terms -- links to definitions of properties, classes and other terms. Follow these links to find machine-readable descriptions of terms

An application is unlikely to want to blindly follow all links. For example applications will certainly want to place a limit on how many additional links it will want to fetch, e.g. one or two hops from the original resource. An application may also want to limit the data retrieved, e.g. by only following certain types of relationship or restricting the domains from which data will be retrieved. The former allows a more directed "crawl" to find related information, while the latter allows simple white/black-listing to only obtain data from trusted sources.

An application might also want to limit network traffic by performing Resource Caching. Parallel Retrieval can also improve performance

The retrieved data will often be parsed into one RDF graph that can then be queried or manipulated within the application. This "working set" might be cached as well as the original source descriptions, to allow for the fact that the same data may be repeatedly referenced.

Some additional processing may also be carried out on the retrieved data, e.g. to apply Smushing to combine all available data about a resource into a single description.

Related

- Missing Isn't Broken
- See Also
- Smushing
- Resource Caching
- Parallel Retrieval

Missing Isn't Broken

How do we handle the potentially messy or incomplete data we use from the web?

Context

In RDF anyone can say anything, anywhere. In other words anyone can make statements about a resource and publish that to the web for others to use. There is no requirement about how much data needs to be published: there are no validation rules that require a minimum amount of data. This means that data on the web may be of varying quality or of varying detail.

This variation is partly a factor of the flexibility of the model, but is really also a fact of life when dealing with any data or content found on the web: even within well-defined standards there may be varying levels of detail available.

How do we deal with this in our Linked Data applications?

Solution

Recognise that "missing isn't broken"

Example(s)

An application might chose to render as much of a FOAF profile (for example) as it can, even though individual profiles might be of varying details.

Discussion

This pattern is really just a restatement of Postel's Law: *Be conservative in what you send; be liberal in what you accept*. This advice is particularly applicable when dealing with any data or content obtained from the web. Applications ought to be tolerant of missing or invalid data and make best effort to process or render what is available

In a Linked Data context this advice is particularly applicable as the flexibility of the RDF model means that there is greater chance for variation in detail across data sources. Rather than rely on schema or document validation, as in XML or relational database systems, to identify and reject data, applications should be designed to be more tolerant.

Of course an application may require some minimum data in order to do anything useful with some data. Although if a data publisher has followed the Label Everything pattern then at a minimum a data browser, for example, may still be able to render the name of the resource.

Unlike other approaches, where data is found to be missing, Linked Data provides additional opportunities for finding more data by supplementing the available data with additional sources, E.g. by using the Follow Your Nose pattern.

Related

- Follow Your Nose
- Label Everything

Further Reading

- Missing Isn't Broken

Parallel Loading

How can we reduce loading times for a web-accessible triple store?

Context

It is quite common for triple stores to expose an HTTP based API to support data loading. E.g. via SPARQL 1.1 Update or the SPARQL 1.1. Uniform Protocol. It can be inefficient or difficult to POST very large datasets over HTTP, e.g. due to protocol time-outs, network errors, etc

Solution

Chunk the data to be loaded into smaller files and use a number of worker processes to submit data via parallel HTTP requests

Example(s)

Most good HTTP client libraries will support parallelisation of HTTP requests. E.g. PHP's `curl_multi` or Ruby's `typhoeus` library.

Discussion

Parallelization can improve any process. Because an RDF graph is a set of triples there is no ordering criteria for adding statements to a store. This means that it is usually possible to divide up an RDF data dump into a number of smaller files or chunks for loading via parallel POST requests.

This approach works best when the RDF data is made available as N-Triples, because the chunking can be done by simply splitting the file on line numbers. This isn't possible with RDF/XML or Turtle files that use prefixes or other syntax short-cuts.

The one caveat to this approach is if the data contains blank nodes. It is important that all statements about a single blank node are submitted in the same batch. Either avoid using bnodes, or split the file based on a Bounded Description of each resource.

Related

- Parallel Retrieval

Parallel Retrieval

How can we improve performance of an application dynamically retrieving Linked Data?

Context

An application that draws on data from the web may typically be retrieving a number of different resources. This is especially true if using the Follow Your Nose pattern to discover data

Solution

Use several workers to make parallel GET requests, with each work writing into a shared RDF graph

Example(s)

Most good HTTP client libraries will support parallelisation of HTTP requests. E.g. PHP's `curl_multi` or Ruby's `typhoeus` library.

Discussion

Parallelisation of HTTP requests can greatly reduce retrieval times, e.g. to time of the single longest GET request.

By combining this approach with Resource Caching of the individual responses, an application can maintain a local cache of the most requested data, which are then combined and parsed into a single RDF graph for driving application behaviour.

Parallelisation is particularly useful for AJAX based applications as browsers are particularly well optimized for making a large number of parallel HTTP requests.

Related

- Follow Your Nose
- Parallel Loading

Resource Caching

How can an application that relies on loading data be more tolerant of network failures and/or reduce use of bandwidth

Context

Linked Data applications will typically need to discover and load data and schemas from the web. A user may request that extra data is displayed from specific locations, and the loading of a new data source may trigger loading of additional schemas, e.g. to discover labels for properties and types, or to pass to a reasoner for inferring additional data and relationships. Some resources and vocabularies may be very commonly used, e.g. the RDF, RDF Schema and OWL vocabularies, while others may only be encountered during run-time.

Solution

Build a local cache of retrieved resources, refreshing the cache only when source data has changed.

Discussion

Retrieving resources from the web, like any other network access, is prone to failure. Repeated fetching of the same resources is waste-ful of bandwidth on the client and the server: a large number of clients can easily overload resources on a system serving up popular vocabularies.

Applications should cache remote resources wherever possible. The cache may be handled entirely in-memory but with sufficient permissions and access to the local file-system an application could also build a persistent cache. Desktop application may ship with a pre-seeded cache of commonly retrieved resources such as ontologies. Efficient use of HTTP request can ensure that cached versions need only be updated when the remote copy of the resource has been updated. Certain vocabularies, e.g. RDF Schema, will only change rarely, if at all. These could be cached for longer periods, if not permanently.

Ideally applications should provide configuration to support the user in managing the amount of local resources (memory or disk space) that can be used by the cache. Control over the location in which cached data will be stored is also useful.

Related

- Follow Your Nose
- Parallel Retrieval

Schema Annotation

How can application-specific processing rules be externalized?

Context

Data driven applications typically end up with built-in processing rules for handling particular types of data, e.g. validation constraints, preferences for specific properties or types, etc. These rules are often encapsulated in procedural code, making them difficult to change. Externalizing these rules as declarative configuration can make an application easier to customize. How can this be achieved with applications that consume Linked Data?

Solution

Externalize constraints using annotation properties that are used to drive processing rules or constraints by annotating classes and properties in a vocabulary

Example(s)

```
ex:RequiredProperty a rdfs:Property;
    rdfs:comment "must be completed on data entry form".

ex:IgnoredProperty a rdfs:Property;
    rdfs:comment "never shown when displaying data".

<http://xmlns.com/foaf/0.1/name>
    a ex:RequiredProperty.

<http://xmlns.com/foaf/0.1/dnaChecksum>
    a ex:IgnoredProperty.
```

Discussion

Simple annotations of classes and properties is a simple and easy way to externalize some common types of application configuration. RDF vocabularies are easily extended with additional properties, making them suitable for extension in this way. Using this approach applications can be very easily tailored to work with a range of different vocabularies.

Annotations may encode a wide range of configuration options including: display preferences, validation constraints, identifier assignment rules for classes, and local labelling for classes and properties. Annotation may even be used to tailor inferencing over specific vocabularies to allow for more local customisation and control over how inferencing is applied; for example a local schema annotation might declare that two classes were equivalent, or that a specific property is an inverse-functional-property, triggering data to be merged.

Schema annotations would typically form part of the overall application configuration and would be applied locally, rather than being published to the open web.

Related

- Annotation

Smushing

How do we merge data about resources that may not be consistently identified?

Context

It will often be the case that different data publishers have used different identifiers for the same resource. In some cases there may be direct Equivalence Links between resources. In others their equivalence might be inferred based on other data, e.g. common properties.

How can we merge statements made about these distinct resources into a single description?

Solution

Apply the technique of "smushing" to manipulate an RDF graph containing the descriptions of each of the resources. Broadly a smushing algorithm will consist of the following steps:

- Decide on the URI for the resource that will hold the final description, i.e. the *target resource* -- this could be one randomly selected from available URIs, or one from a local dataset
- Identify all *equivalent resources* -- i.e. by finding Equivalence Links such as `owl:sameAs` statements, or by property values that indicate that two resources are similar (e.g. Inverse Functional Properties, see below).
- Iterate over the *equivalent resources* and for each RDF statement for which it is the *subject*, assert a new statement with the same predicate and object but using the *target resource* as the subject
- Iterate over the *equivalent resources* and for each RDF statement for which it is the *object*, assert a new statement with the same subject and predicate but using the *target resource* as the object

The end result will be an modified RDF graph with all properties of the *equivalent resources* being "copied" to the *target resource*. In addition, any references to the *equivalent resources* will also be made to the target resource

By applying this to all resources in a graph, the available data can be normalized into a consistent set of descriptions based on a known set of resources. An application may then generate a Bounded Description of any resource and guarantee that it will include all available data

Example(s)

Assume we start with the following graph, which contains two equivalent resources, as defined by an `owl:sameAs` link.

```
<http://example.com/product/6>
  rdfs:label "Camera";
  owl:sameAs <http://example.org/cameras/10>.

<http://example.org/cameras/10>
  ex:manufacturer <http://example.org/company/5>.

<http://example.org/company/5>.
```

```
ex:manufactured <http://example.org/cameras/10>.
```

Assuming we want to collate all data around resources from `example.com`, we can apply smushing to create the following graph:

```
<http://example.com/product/6>
  rdfs:label "Camera";
  owl:sameAs <http://example.org/cameras/10>;
  ex:manufacturer <http://example.org/company/5>.

<http://example.org/cameras/10>
  ex:manufacturer <http://example.org/company/5>.

<http://example.org/company/5>.
  ex:manufactured <http://example.org/cameras/10>.
  ex:manufactured <http://example.com/product/6>.
```

We can also tidy up the graph to remove statements about the equivalent resources, leaving:

```
<http://example.com/product/6>
  rdfs:label "Camera";
  owl:sameAs <http://example.org/cameras/10>;
  ex:manufacturer <http://example.org/company/5>.

<http://example.org/company/5>.
  ex:manufactured <http://example.com/product/6>.
```

Discussion

Smushing is essentially a process of inference: by using available data we create new statements. Any OWL reasoner will carry out this kind of data merging automatically based on the available data and schema/ontology without the need for custom code. Applications that are using a triple store that applies inferencing by default will not need to use this approach. However for applications that don't need a full inferencing engine, or need only lightweight merging of data, then a custom smushing algorithm can achieve the same goal.

There are several different variations on algorithm described above. For example, applications might vary in how they nominate the *target resource*. Typically though this will be based on a preferred URI. Algorithms can also be divided into those that preserve the original statements, e.g. so that the *equivalent resources* remain in the source RDF graph, or whether their statements are removed from the graph to leave only a normalized description. Applications could also use Named Graphs to separately stored the "smushed" view of the data, preserving the original data in another graph or triple store.

As noted above there are also several ways to identify equivalent resources. Equivalence Links are an obvious approach. Other cues can also be used including the use of Inverse Functional Properties. An inverse functional property is simply a property whose value uniquely identifies a resource, such as Literal Keys.

An application is also free to apply its own rules about what constitutes "equivalence". For example an application may decide to merge together resources with similar property values, even if those properties are not declared as Inverse Functional Properties. This allows for local customization of smushing rules, but runs the risk of generating false positives. One way to apply these custom rules is to use local Schema Annotations to declare specific properties as being equivalent. This has the benefit of working with both custom code and OWL reasoners.

Smushing is often used to normalize an RDF graph resulting from a Follow Your Nose approach to data discovery

Related

- [Equivalence Links](#)
- [Follow Your Nose](#)

Further Reading

- [Smushing](#)
- [RDF Smushing](#)
- [Smushing Algorithms](#)

Transformation Query

How can we normalize or transform some RDF data so that it conforms to a preferred model?

Context

There are a broad range of different vocabularies in use on the Linked Data web. While there has been convergence on common vocabularies in a number of domains, there will always be some variations in how data is presented, e.g. using slightly different modelling styles and vocabularies. Schemas will also evolve over time and certain properties may be refined or replaced. How can a consuming application normalize these different models into a single preferred representation that matches expectations or requirements of the application code?

Solution

Use a SPARQL CONSTRUCT query, or collection of queries, to generate a normalized view of the data, saving the results back into the triple store.

Example(s)

The following query normalizes any one of three different naming or labelling properties into `rdfs:label` properties.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX vCard: <http://www.w3.org/2001/vcard-rdf/3.0#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
```

```
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
CONSTRUCT {
  ?s rdfs:label ?o.
}
WHERE {
  { ?s foaf:name ?o. }
  UNION
  { ?s vCard:FN ?o. }
  UNION
  { ?s skos:prefLabel ?o. }
}
```

Discussion

Writing application code to normalize data patterns can be tedious and difficult to maintain in the face of changes. SPARQL CONSTRUCT queries provide a way to generate arbitrary RDF graphs from existing data. They therefore provide a way to carry out transformations on RDF graph. These transformations can be used to infer new data based on existing patterns and this covers the generation of normalized data models. SPARQL therefore provides a declarative syntax for describing graph transformations. The increased expressivity of SPARQL 1.1 will allow more complex transformations to be specified.

An application may apply one or more Transformation Queries to its data source either during execution, e.g. to extract a graph of a known structure, or during assembly of the underlying dataset, e.g. as part of the Blackboard pattern.

Each transformation query may cover one specific normalization task. However, as shown in the example above, several operations can be combined using a UNION query. This allows the graph pattern of the query to match for a number of different variants, resulting in the generation of a single standard output graph.