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 2012-05-31

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.

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

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 practioner 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 reflective 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 debates 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 6, *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, for clarity, have not always included the declarations of these prefixes in the examples. This allows the example to focus on the particular usage being demonstrated.

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#> .
```

In the Data Management Patterns section of the book a number of the examples use the TRiG [http://www4.wiwiss.fu-berlin.de/bizer/trig/] syntax, which is an extension of Turtle that supports Named Graphs.

Chapter 2. Identifier Patterns

Abstract

The single most important part of the Linked Data approach is the adoption of web-scale identifiers (URIs) to identify things of interest: people, events, places, statistical observations, colours. Anything that we want to publish data about on the web needs to have a URI, allowing it to be referenced, browsed and linked using existing web tools. The existing tools of the web of documents are already designed to work well with things that have URIs. We can "like" them, discuss them, and refer to them in documents.

In RDF we capture data as statements about resources. RDF allows resources to have global identifiers or to be un-named "blank nodes". While blank nodes may offer flexibility for some use cases, in a Linked Data context blank nodes limit our ability to collaboratively annotate data. A blank node cannot be the target of a link and we can't annotate it with new information from new sources. As one of the biggest benefits of the Linked Data approach is that "anyone can say anything anywhere", use of blank nodes undermines some of the advantages we can gain from wide adoption of the RDF model. Even within the closed world of a single application dataset, use of blank nodes can quickly become limiting when integrating new data.

Successful publishing of Linked Data requires the careful selection of good, clean, stable URIs for the resources in a dataset. This means that the most important first step in any Linked Data project is deciding on an appropriate identifier scheme: the conventions for how URIs will be assigned to resources. This is not to say that conventions won't evolve or be extended over time, but some upfront thought about identifiers is always beneficial.

This chapter introduces a collection of design patterns that promote some principled ways to assign identifiers within a dataset. All of these patterns are in wide use today and so are tried and tested in the field.

Many of these patterns are also prevalent in modern web frameworks and are more generally applicable to URL design for web applications.

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 predictable, 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

Rebased URI

How can we construct one URI based on another?

Context

Sometimes when generating a Patterned URI the key that we have for as URL Slug is not a simple literal value, but instead another URI. For example this can occur when generating a new Named Graph URI for a resource, or when defining a service URL for a URI Resolver.

Solution

Rewrite the original URI to use a new, predictable base URI using on of the options described below.

Example(s)

An application needs to generate a new Named Graph URI for a resource URI of `http://example.org/document/1`. The application uses a regular expression to rewrite the original URI to a new base, e.g. `http://graphs.example.org/document/1`.

Discussion

URL rewriting is a common feature of all web servers and most web frameworks. Rewriting is normally carried out using regular expressions to match and replace portions of the original URI with some standard replacement text.

URL rewriting is used in several Linked Data services in order to create new URLs. This is typically to support URI resolution for remote (RDF) resources.

Several different approaches seem to be in use. The following examples all show a rewrite for this URI: `http://example.org/document/1`. Each example notes an example service that uses the approach:

- *Simple Prefixing* (URIBurnder): `http://service.example.com/resolve/http://example.org/document/1`. The original URI is simply appended to a new base URL. Has the advantage of working with any protocol.
- *Prefixing, No Protocol* (Triplr): `http://service.example.com/resolve/example.org/document/1`. The original URI is simply appended to a new base URL after first removing the protocol (e.g. "http://"). Server will need to assume the `http` protocol if de-referencing the URI or reversing the rewrite.
- *Prefixing, With Delimiter* (Callimachus): `http://service.example.com/resolve;http://example.org/document/1`. The original URI is simply appended to a new base URL which ends with a semi-colon.
- *Prefixing, As Parameter* (SPARQL Graph Protocol): `http://service.example.com/resolve?uri=http://example.org/document/1`. The original URI is simply appended to a new base URL as a query string parameter.
- *Rewritten Authority*: `http://data.example.org/document/1`. The authority component of the URI is rewritten to create a new base URI. This approach might lead to URI clashes unless the input URIs have a predictable structure.

Related

- URI resolver

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

Creating a model of some domain for publishing data as Linked Data is, fundamentally, no different to any other form of modelling exercise: we need to understand the core entities and their attributes, including their relationships to one another. RDF has a core model that is based on the decared old Entity-Attribute-Value (EAV) approach that underpins many different information management approaches and technologies. What is different and new for many people is the context in which that modelling is carried out.

When we are creating an Entity-Relationship (ER) or Object-Oriented model we are typically doing so in order to support the needs of a particular application. We have some idea about what data the application needs to store and we tailor the model accordingly. We also typically optimise the model for the needs of that application. In particular, when translating a logical ER into a physical database schema, we often need to introduce new entities, e.g. to capture many-many relationships in order to be able to express the model within the restrictions imposed by a relational database.

In a Linked Data context we approach a modelling exercise in a different way.

Firstly we may be able to avoid the exercise completely. Using RDF Schema and OWL it is possible to share vocabularies (or ontologies, or schemas, whatever your prefered term) on the web, just as we share data. This means that communities of interest can collaborate around a data model and then extend it where necessary. With semantic web technologies we can better crowd-source and share our data models.

It is often the case that a combination of existing vocabularies will adequately cover a domain. Where there is a short-fall we can define small extensions, e.g. new properties or types, that extend the model for our purposes.

The second key difference is that we focus on modelling the domain itself and set aside the immediate needs of the application. By focusing on the model rather than the application, we are more likely to be able to extend and enrich the model at a later date. Thinking about the entities and their relationships, rather than our application code, results in a stronger model. And one that may be more likely to be of use to others.

Of course a model must ultimately support the kinds of data we want to capture or manipulate in our application. The ability to populate and query a model is a good test of its fit for purpose, but application requirements alone shouldn't guide our modelling.

This is not the same as saying that all modelling must be done up front. Modelling for Linked Data can be agile and iterative. We should just take care to think about the likely areas of extension, as the Link Not Label pattern illustrates.

Thirdly, and finally, the other important way in which RDF modelling differs from other approaches is that there is no separate physical data model: the logical model is exactly how our data is stored. An RDF triple store is "schema-free" in the sense that we don't have to define the physical layout (e.g. tables, columns) of our database. Any RDF triple store can store any RDF data expressed using any RDF vocabulary.

This makes RDF very amenable for rapid, iterative application development. Importantly it also means that everyone involved in a project team is working off the same domain model; the same mental model of how the resources in a dataset are described and how they relate to one another.

This chapter introduces a number of RDF modelling patterns that are useful in a variety of different use cases. Several of the patterns are really illustrations of how to use particular features of RDF, e.g. annotation of literals with data types or languages. We've included these within the overall patterns collection as we feel that presenting them as solutions to specific questions, e.g. "How can internationalized text be expressed in RDF?" may help address what are frequently asked questions by new users.

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`.

The importance of applying labels to Linked Data, as well as evidence for the poor adoption of this practice, is given in a paper called "Labels in the Web of Data" by Basil Ell, Denny Vrande#i#, and Elena Simperl.

Related

- Preferred Label

Further Reading

- Labels in the Web of Data (Paper)
- Labels in the Web of Data (Video Lecture + Slides)

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 over-looked 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 are 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 preceeded 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 date. 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 preceeded 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 reuired 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

There is an increasingly wide variety of organisations who are publishing Linked Open Data. The growth of the "Linked Data Cloud" is a staple of all Linked Data presentations and charts the early success of the community in boot-strapping an interconnected set of datasets across the web.

There is also a wide variety of ways in which Linked Data is being published, including:

- simple statically generated RDF and HTML files
- RDFa embedded in web application pages
- As an extension to the principled design of a web application that supports a rich variety of views and data formats
- As independent datasets dynamically generated over data stored in triple stores, relational databases or other data sources

In addition data might be published as both a primary or secondary sources, e.g. as an RDF conversion of a data dump available from another organisation. The ongoing challenge for the growth and adoption of Linked Data will be in simplifying getting more data online, e.g. continuous improvement to tools, as well as the introduction of more primary sources that are commitment to publishing high quality, regularly updated data.

Regardless of the source of the data or the means of its publication, there are a number of recurring patterns and frequently asked questions that relate to best practices around data publishing. This chapter documents a number of patterns relating to Linked Data publishing, and in particular how data can be made more discoverable, and over time enriched and inter-linked.

While many of these patterns may have been discovered through the publication of Linked Open Data, they are generally applicable to Linked Data publishing in other contexts, e.g. inside an enterprise.

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 relevent for the user.

Related

- Primary Topic Autodiscovery

Dataset Autodiscovery

How can an application discover the datasets (and any associated APIs) published by a website?

Context

A Linked Data application might apply the Follow Your Nose pattern to discover additional data about resources in an RDF graph. But the application may need to discovert additional metadata about the dataset to which the resource belongs, such as its license. The application may also need access to additional facilities, such as a SPARQL endpoint, in order to extract any additional data it requires.

Solution

Use the Vocabulary of Interlinked Datasets (VoiD) vocabulary to publish a description of a dataset. Make this description available from the standard location at the domain of the website, i.e. `/.well-known/void`.

Example(s)

An application discovers the URI `http://data.example.org/thing/1` in a dataset. De-referencing that URI will yield a description of that resource. In order to discover a SPARQL endpoint for that dataset, the application performs a GET request on `http://data.example.org/.well-known/void`. This URL returns a description of the dataset published at that domain.

Discussion

RFC 5785 defines a means for declaring well known URIs for a website. A registry of well known URIs is maintained by the IETF, and a registration has been made for `/.well-known/void`. As the VoiD specification indicates, this URI should return a description of all datasets available from the specified domain.

Providing such as description allows a consuming application to bootstrap its knowledge of available datasets, their licensing, and the means of interacting with them. This can greatly simplify working with unknown data sources. For example an application could discover a SPARQL endpoint or data dump that would allow it to more efficiently harvest some required data.

Related

- Autodiscovery

Further Reading

- VoiD Dataset Auto-discovery
- RFC5785 - Defining Well-Known URIs

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 ot 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 embedded 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 requring 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 its 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)

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

Unpublish

How do we temporarily or permanently remove some Linked Data from the web?

Context

It is sometimes necessary to remove a Linked Data set from the web, either in whole or in part. A dataset might be published by an organisation who can no longer commit to its long term availability. Or a dataset

might be transferred to a new authority. This applies to scenarios where a third-party has done a proof-of-concept conversion of a dataset that is later replaced by an official version.

In practical terms a dataset might also be temporarily unavailable for any number of technical reasons.

How can the temporary or permanent removal of some data be communicated? And, in cases where it has been replaced or superceded, how can the new authoritative copy be referenced.

Solution

Use an appropriate HTTP status code to indicate the temporary or permanent removal of a resource, or its migration to a new location.

Where a resource has moved to a new location, publish Equivalence Links between the old and the new resources.

Example(s)

A dataset has been published by a developer illustrating the benefits of a Linked Data approach to data publishing. The developer has used URIs based on a domain of `http://demo.example.net`. At a later date the original owner of the data decides to embrace Linked Data publishing. The new dataset will be published at `http://authority.example.org`.

The developer therefore reconfigures his web server to redirect all URIs for `http://demo.example.net` to return a `301` redirect to the new domain. Consuming applications are then able to determine that the data has been permanently moved to a new location.

The developer also creates a data dump that contains a series of RDF statements that indicate that all of the resources originally available from `http://demo.example.net` are `owl:sameAs` the new official URIs.

Discussion

Movement or removal of web resources is not specific to Linked Data, and so HTTP offers several status codes that are applicable to the circumstances described in this pattern. Using the correct HTTP status code is important to ensure that clients can differentiate between the different scenarios. An HTTP status code of `503` indicates that a resource is temporarily unavailable; `410` that a resource has been deleted; and `301` that a resource has been moved to a new location. Returning `404` for a resource that is only temporarily unavailable, or has been moved or deleted is bad practice.

Where data has been replaced, e.g. new URIs have been minted either at the same authority or a new one, then publishing RDF assertions that relate the two URIs together is also useful. An `owl:sameAs` statement will communicate that two URIs are equivalent and will ensure that any historical annotations associated with the URI can be united with any newly published data.

Lastly, in case of complete removal of a dataset, it is important to consider archiving scenarios. If licensing permits, then data publishers should provide a data dump of a complete dataset. Doing so will mean that consumers, or intermediary services, can host local caches of the data to support continued URI resolution (e.g. via a URI Resolver). This mitigates impacts on downstream consumers.

Related

- Equivalence Links

- URI Resolver

Chapter 5. Data Management Patterns

Abstract

While the statements in an RDF dataset describe a direct graph of connections between resources, the collection of triples itself has no structure: it is just a set of RDF statements. This lack of structure is not a problem for many simple RDF applications; the application code and behaviour is focused on exploring the connections in the graph. But for more complex systems that involve integrating data from many different sources it becomes useful to be able to partition a graph into a collection of smaller sub-graphs.

One reason for partitioning of the graph, is to support data extraction. Creating a useful view over one or more resources in a graph, e.g. to drive a user interface. There are a number of different partitioning mechanisms that can be used and these are covered in the Bounded Description pattern described in the next chapter.

A very important reason for wanting to partition a graph is to make data management simpler. By partitioning a graph according to its source or the kinds of statements it contains we can make it easier to organise and update a dataset. Managing smaller graphs gives more affordance to the data, allowing entire collections of statements to be manipulated more easily.

The means by which this affordance is created is by extending the core triple model of RDF to include an extra identifier. This allows us to identify collections of RDF triples, known as Named Graphs. The patterns captured in this chapter describe different approaches for managing RDF data using Named Graphs. The patterns cover different approaches for deciding on the scope of individual graphs, as well as how to annotate individual graphs, as well as ultimately re-assembling graphs back into a useful whole.

It should be apparent that Named Graphs is essentially a document-oriented approach to managing RDF data. Each document contains a collection of RDF statements. This means that we can benefit from thinking about good document design when determining the scope of each graph, as well as more general document management practices in deciding how to organise our data.

The beauty of the RDF model is that it is trivial to manage a triple store as a collection of documents (graphs) whilst still driving application logic from the overall web of connections described by the statements contained in those documents. An XML database might also offer facilities for managing collections of XML documents, but there is no standard way in which the content of those documents can be viewed or manipulated. In contrast the data merging model described by RDF provides a principled way to merge data across documents ((Union Graph).

This flexibility provides some powerful data management options for RDF applications.

Graph Annotation

How can we capture some metadata about a collection of triples?

Context

There are a number of scenarios where it is useful to capture some metadata about a collection of statements in a triplestore. For example we may want to capture:

- publication metadata, such as the date that the triples were asserted or last updated
- provenance metadata, e.g. who asserted those triples, or how they were generated

- access control data, e.g. which user(s) or role(s) can access those triples

The Named Graph pattern allows us to identify a set of triples, via a URI.But how do we then capture information about that graph?

Solution

Treat the Named Graph like any other resource in your dataset and make additional RDF statements about the graph itself. Those additional statements can themselves be stored in a further named graph.

Example(s)

A triple store contains a Named Graph that is used to store the results of transforming a local database into RDF. The Named Graph has been given the identifier of `http://app.example.org/graphs/my-database` to label the triples resulting from that conversion process. As the source is a live, regularly updated database it is useful to know when the RDF conversion was last executed and the data stored. It is also useful to know which version of the conversion software was used, to track potential bugs. This additional metadata could be captured as follows:

```
@prefix ex: <http://www.example.org/> .

#Named graph containing results of database conversion
<http://app.example.org/graphs/my-database> {
   ...triples from conversion...
}

#Metadata graph
<http://app.example.org/graphs> {
  #Description of a named graph in this dataset
  <http://app.example.org/graphs/my-database> dct:source <http://app.example.org/d
  <http://app.example.org/graphs/my-database> dct:created "2012-05-28"^^xsd:date.
  <http://app.example.org/graphs/my-database> foaf:generatorAgent <http://app.exam

  ...descriptions of other graphs...
}
```

In the above example there are two graphs in the dataset: the graph containing the data from the conversion and a second graph containing metadata about the first. An application that needed to identify the date that some triples were asserted can easily do this by querying the metadata graph.

Discussion

The majority of Named Graph uses cases require some additional context to be captured about the set of triples labelled with a graph URI. Describing named graphs in RDF, by using the graph URI as the subject of additional RDF statements, provides a simple way to capture additional metadata relevant to an application.

As shown in the above example, a specific "well-known" named graph in the dataset can be designated as the location in which these extra statements are recorded. However it is also possible to use multiple named graphs. In this case we might have one graph for the data and one "parallel" graph that captures the context. For example if we could adjust the above example as follows:

```
@prefix ex: <http://www.example.org/> .

#Named graph containing results of database conversion
<http://app.example.org/graphs/my-database> {
   ...triples from conversion...
}

#"Parallel" metadata graph
<http://app.example.org/metadata-graphs/my-database> {
  #Description of  named graph in this dataset
  <http://app.example.org/graphs/my-database> dct:source <http://app.example.org/d
  <http://app.example.org/graphs/my-database> dct:created "2012-05-28"^^xsd:date.
  <http://app.example.org/graphs/my-database> foaf:generatorAgent <http://app.exam
}
...other named graphs
}
```

Using a pair of named graphs per source can quickly lead to a very large number of graphs in a dataset. Some triple stores may not be optimized to deal with a very large number of graphs. However the approach does benefit from flexibility of adding and removing both source graphs and their metadata.

Care should be taken when choosing graph URIs. If an application uses the source document URL of some RDF as the named graph URI then this can lead to confusing statements as illustrated below:

```
@prefix ex: <http://www.example.org/> .

#Named graph containing results of HTTP crawl
<http://example.org/person/joe.rdf> {
  #statements contained in the document, about itself
  <http://example.org/person/joe.rdf> foaf:primaryTopic <http://example.org/person
  #date the FOAF document was created
  <http://example.org/person/joe.rdf> dct:created "2010-06-01".

  <http://example.org/person/joe> foaf:name "Joe Bloggs".
}

#Metadata graph, describing results of crawl
<http://app.example.org/graphs/> {
    #date the named graph was created
    <http://example.org/person/joe.rdf> dct:created "2012-05-28".
}
```

In the above example there are two `dct:created` properties associated with the resource `http://example.org/person/joe.rdf`. That document is a FOAF description which describes its primary topic and the date it was generated. The second date was added by a hypothetical web crawler that captured the date it stored that information in a named graph. It is possible for a SPARQL query applied to the union graph for this dataset to return conflicting information, suggesting that the use of the source URL as a graph identifier is a poor choice. A better alternative would be to add some indirection:

```
@prefix ex: <http://www.example.org/> .

#Named graph containing results of HTTP crawl
```

```
<http://app.example.org/graphs?source=http://example.org/person/joe.rdf> {
  #statements contained in the document, about itself
  <http://example.org/person/joe.rdf> foaf:primaryTopic <http://example.org/person
  #date the FOAF document was created
  <http://example.org/person/joe.rdf> dct:created "2010-06-01".

  <http://example.org/person/joe> foaf:name "Joe Bloggs".
}

#Metadata graph, describing results of crawl
<http://app.example.org/graphs> {
    #date the named graph was created
    <http://app.example.org/graphs?source=http://example.org/person/joe.rdf> dct:c
    #source of the graph
 <http://app.example.org/graphs?source=http://example.org/person/joe.rdf> dct:sour
}
```

In the revised example a new URI is associated with the results of the web crawl. The web crawler then uses this URI to record the required metadata. By adding an `dct:source` property (or similar) it is still possible to identify which named graph was derived from which source URL. As can be seen a Patterned URI is used to generate the graph identifier, giving a predictable structure to the store.

Related

- Named Graph
- Graph Per Resource
- Graph Per Source

Further Reading

- Named Graph (Wikipedia)

Graph Per Aspect

How can we avoid contention around updates to a single graph when applying the Graph Per Resource pattern?

Context

For some applications the entire description of a resource might be maintained by a single authority, e.g. the data might all derive from a single data conversion or be managed by a single editing interface. However in some applications data about a single resource might be contributed in different ways. One example might be a VoiD description for a dataset. A dataset description may consist of a mixture of hand-authored information -- e.g. a title, description, and example resources -- plus some statistics derived from the dataset itself, e.g. size and class partitions. An administrator might update the descriptive aspects while the rest is updated asynchronously by a background application that analyses the dataset.

Multiple applications writing to the same graph could lead to contention for system resources or the need to implement complex locking behaviour.

Solution

Apply a combination of the Graph Per Resource and Graph Per Source patterns and factor out the different aspects of a resources description into separate graphs. Use a Union Graph to collate the different aspects of the description of a resource into a single view.

Example(s)

A content management application stores information about articles. This includes descriptive metadata about the articles as well as pointers to the content. Content metadata will be manually managed by users. In the background two additional processes will be carrying out additional tasks. One will be retrieving the content of the article to perform text mining, resulting in machine-tagging of subjects in the article. The second will be harvesting related links from the rest of the system and the web. The "aspect graphs" are created: one for the core metadata, one for the tags and one for the links:

```
#core description of a resource; provided by user
<http://data.example.org/graphs/core/document/1> {
 <http://example.org/document/1> dct:title "Bath in the Summertime".
}
#tags; maintained by process 1.
<http://data.example.org/graphs/tags/document/1> {
 <http://example.org/document/1> dc:subject "Bath".
 <http://example.org/document/1> dc:subject "Travel".
}
#related links; maintained by process 2.
<http://data.example.org/graphs/links/document/1> {
 <http://example.org/document/1> dct:related <http://travel.example.org/doc/bath>.
}
#System metadata graph, listing topic of each graph
<http://data.example.org/graphs> {
 <http://data.example.org/graphs/core/document/1> foaf:primaryTopic <http://exampl
 <http://data.example.org/graphs/tags/document/1> foaf:primaryTopic <http://exampl
 <http://data.example.org/graphs/links/document/1> foaf:primaryTopic <http://examp
}
```

As the above example illustrates, graph URIs for the different aspects of a resources description can by generated by using Patterned URIs. A fourth graph, covering system-wide metadata is also maintained. This graph lists the `foaf:primaryTopic` of each graph, allowing applications to discover which graphs relate to a specific resource.

An application consuming this data could rely on either a system default Union Graph to provide a complete view of a resource. Partial views might address individual named graphs. Using a CONSTRUCT query it is also possible to construct a view of a resource using just those graphs referenced in the system metadata graph:

```
CONSTRUCT { ?s ?p ?o. }
WHERE {
  <http://data.example.org/graphs> {
      ?graph foaf:primaryTopic <http://example.org/document/1>.
```

```
  }
  ?graph { ?s ?p ?o. }
}
```

Discussion

Named graphs provide flexibility in how to organise an RDF data store. In some cases storage is oriented towards the sources of data, in others around individual resources. The Graph Per Aspect pattern provides a combination of those features that allows for very fine-grained graph management. The description of each resource is divided over a number of graphs, each of which is contributed to the system by a different source or application component.

As with the other named graph patterns reliance is made on the Union Graph pattern to bring together the description of a resource into a single consistent view.

Separating out aspects of resource description into different graphs also provides a way to shard a dataset. Different aspects might be stored in different triple stores across a network. These are then brought together in the application for building a user interface. With knowledge of how graphs are partitioned across the network, as well as which graphs contain which statements, an application can use Parallel Retrieval to synthesis a local working copy of a resource's description. This aggregation could happen within a server component or on the client-side

The small, focused graphs created by use of this pattern and, more generally, by the Graph Per Resource pattern are very amenable for delivery to mobile & web clients for local processing. By separating out the different aspects of a resource into graphs that are likely to change with differing frequencies, caching can be made much more efficient.

Related

- Named Graph
- Graph Annotation
- Graph Per Source
- Graph Per Aspect

Further Reading

- Keep Your Triples Together: Modeling a RESTful, Layered Linked Data Store

Graph Per Resource

How can we organise a triple store in order to make it easy to manage the statements about an individual resource?

Context

Web applications typically offer forms for editing the description of an individual resource, e.g. the title, description and tags that apply to a specific photo. RESTful APIs typically support replacing the description of a resource using a PUT operation. In both cases it would be useful to be able to serialize all

of the statements relating to a given resource, including any edits, and directly replace the relevant triples in a triple store backing the application or API.

Solution

Store the description of each resource in a separate Named Graph using a graph URI derived from the resource URI as the graph URI. When the resource is updated, simply replace the contents of that graph with the latest state of the resource.

Example(s)

A user is editing the description of `http://example.org/picture/1`. The application delivers a web form to the authorised user which presents the current description of the resource in an interactive editing form. When the client submits the changes back to the server, it does so by serializing the state of the form as RDF which is then processed by the server.

On receiving the update from the client, the server-side code computes the graph URI for the resource being edited. For this application graph URIs are derived from resource URIs by a simple rewrite. For example the data for `http://example.org/picture/1` is stored in `http://data.example.org/graphs/picture/1`.

To apply the update the server-side code simply determines the appropriate graph URI and then stores the data in a SPARQL 1.1. Graph Store Protocol enabled store using a simple PUT operation.

Discussion

Partioning the triple store by resource rather than by source provides an easy way to quickly access the description of an individual resource without having to use a SPARQL query. This is particularly true if the graph URI for a resource can be derived using a simple algorithm, e.g. rewriting to a different base URI, pre-pending a known prefix, etc.

The SPARQL 1.1. Graph Store Protocol provides a way for code to find and update the description of a resource without needing further knowledge of the data stored about any specific resource. This allows server-side code to remain relatively generic: it doesn't need to know details about what kinds of data is captured about individual resources, it just needs to know where to persist the data its given.

When applying this pattern it is common to store a bounded description of the resource in each graph ensuring that each graph has a clear scope.

While this pattern makes it easy to manage a bounded description for a resource, including its relationships to other resource, it doesn't help with managing references to the resource elsewhere in the store. E.g. if a resource is deleted (by removing its graph), there may still be statements relating to that resource elsewhere in the graph. Additional SPARQL Update operations, for example, would be needed to remove these statements. Depending on the complexity of the graph structure and the needs of the application these overheads might make this pattern unsuitable.

In circumstances where additional context is required, e.g. in-bound relations, for an application view, a fall-back to SPARQL queries over a Union Graph of the store can provide additional flexibility. However for the simplest cases of serving Linked Data pages, this pattern makes for very simple and light-weight application code.

For very simple use cases the graph URI of the graph holding the description of the resource could in fact be the resource URI. However in practice this can limit system evolution. As explained in the Graph Annotation pattern, the ability to store additional graph metadata can be useful. And in that case a separate graph URI is essential to avoid potential confusion around the scope of individual statements.

In some applications several different processes might contribute to the creation of a description of a resource. Using a single graph per resource might therefore involve contention around several processes. The Graph Per Aspect pattern allows this contention to be removed by using multiple graphs per resource.

Related

- Named Graph
- Graph Annotation
- Graph Per Source
- Graph Per Aspect

Further Reading

- Named Graph (Wikipedia)
- Managing RDF using Named Graphs

Graph Per Source

How can we track the source of some triples in an RDF dataset?

Context

A common application requirement is to create a local cache of RDF data which originates from other sources. For example an application might crawl or harvest data from the Linked Data cloud and store it local triple store. Data from relational databases or other sources might also be regularly converted into RDF and added to a triple store.

While the applications directly consuming this data may not need any knowledge of their origin when querying the dataset, the applications doing the data management activities (e.g. crawling or data conversions) will need to be able to clearly identify which triples came from which source dataset or URL.

Solution

Use a separate named graph for each data source. For the named graph URI either use a well-known URI for each data source, e.g. the URI of the dataset, or simply the URL from which the data was retrieved.

Example(s)

An application is harvesting RDF documents from the web. Upon discovering the URL `http://www.example.org/person.rdf` the application does a GET request to retrieve the document. The RDF statements found in that document are then stored in the applications triple store in a new named graph with the source URI as the graph identifier:

```
#Named graph URI is source document
<http://www.example.org/person.rdf> {
  #Triples from source document
  <http://www.example.org/person/joe> foaf:name "Joe Bloggs".
}
```

The harvesting application can easily determine whether it has already harvested a URL by checking to see whether a named graph with that URI exists in the store. The next time that the application retrieves the content of the URL, e.g. as part of a cache refresh, the contents of just that graph can be replaced.

Discussion

The URI that labels a named graph can be used in several different ways. One very common approach is to use the label to hold the URL from which a set of triples was retrieved. By using the graph URI to identify the source of the data it becomes possible to track the provenance of the data within the store. This can simplify common dataset construction scenarios, e.g. mirroring of data harvested from the web.

Checking for the presence of a graph within a store can be achieved using a simple assertion query:

```
ASK WHERE {
 <http://www.example.org/person.rdf> { ?s ?p ?o. }
}
```

The query will return true if there are any statements associated with a graph with the specified identifier

An application may need to be present to an end-user the list of locations from which some RDF statements have been found. This too can be achieved using a simple SPARQL query:

```
SELECT ?graph WHERE {
 ?graph {
    <http://www.example.org/person/joe> foaf:name "Joe Bloggs".
 }
}
```

In the above example the query will return the URIs of all graphs that contain the specified RDF triple; "which sources say that `http://www.example.org/person/joe` has a `foaf:name` of "Joe Bloggs".

It is often important to capture some additional information about the source of the URL. For example a web crawler might want to record the retrieval data and HTTP headers associated with the original web request. This would allow for more efficient cache maintenance. The Graph Annotation pattern describes this in more detail.

It is worth noting that in some cases a direct application of this pattern can make graph annotation more awkward: by using the source URL as the graph URI we can no longer distinguish between statements made about the *graph* and statements made about the *source document*. E.g. the date of creation of the graph and the date of creation of the document. This may be an issue for some application scenarios, although for others (e.g. simple aggregation) this may not be an problem.

As described in the Graph Annotation pattern the solution to this issue is to use a different identifier for the graph, e.g. a Patterned URI derived from the source URL. The source URL of the graph can then be captured as an annotation.

Related

- Named Graph
- Graph Annotation
- Graph Per Resource

Further Reading

- Named Graph (Wikipedia)

Named Graph

How can we identify a useful sub-set of RDF triples within a triple store?

Context

A triplestore is a set of RDF triples. Upon loading some RDF data into such a collection we lose the ability to be able to identify a sub-set of RDF triples, e.g. to identify their source of origin. We also lose knowledge of whether a single triple was asserted once or multiple times, e.g. by separate distinct sources. While it is possible to extract triples based on particular graph patterns, e.g. all triples with the same subject, predicate, or object there is no way to recover the original contect.

Solution

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

By assigning a URI to a set of triples and my retaining that additional identifier in the store (a *quad store*), we can either treat the whole collection as a single set of triples, or work with an subset based on its graph identifier.

Example(s)

The following example shows an RDF graph using the TRiG notation (Turtle plus graphs)

```
@prefix ex: <http://www.example.org/> .

#Named graph in TRiG
<http://www.example.org/graph/1> {
  <http://www.example.org/document/7> rdfs:label "Example".
}
```

The following example shows an RDF graph using the NQuads notation (N-Triples plus graphs)

```
<http://www.example.org/document/7> rdfs:label "Example" <http://www.example.org/g
```

In both cases the graphs contain a single RDF triple

Discussion

The idea of extending the triple into a quad, through the addition of an additional URI has been around for many years. The Many early triple stores supported the notion of a quad, with the majority now supporting named graphs as a key feature. SPARQL 1.0 was the first semantic web specification to directly reference

the concept, allowing queries to be addressed to an individual named graph or a collection of graphs. Syntaxes for serialising named graphs also exist, allowing quads to be exchanged between systems. The most commmonly used quad syntaxes are TRiG (a derivative of Turtle) and NQuads (a derivative of NTriples.

By extending the core RDF model from a triple to a quad, Named graphs provide a useful extra degree of freedom when managing an RDF dataset. A useful analogy when thinking about a quad store is that of a collection of documents: each named graph is a separate "document" in the database that can be manipulated independently from any others. However the document identifier (the graph URI) can be ignored when working with data, relying on RDF graph merging rules to provide a view across all documents. A graph store therefore offers a useful way to manage sets of statements without losing the ability to easily merge data from across sources.

Graphs have been usefully applied to solving a number of different data management problems in RDF applications. 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

The lack of standardisation about what the graph URI associated with a named graph identifies means that different applications have used it for different purposes. For example one application might simply use the URI as a local identifier for a set of triples, whereas another might associate extra semantics with the URI, e.g. using it to hold the URL from which the data was originally retrieved.

The simplest way to thing about the graph URI is as a simple identifier or label that can be associated with some triples. The fact that the identifier is a URI has some added benefits. For example we could then capture RDF statements to describe the graph. This has been applied as an alternative to reification for handling versioning and provenance in datasets, but graph annotation is useful in a number of scenarios.

Related

- Reified Statement
- Graph Annotation

Union Graph

How can we generate a simple view over the statements in a collection of named graphs?

Context

The Named Graph pattern describes an approach for managing RDF statements as a collection of graphs. The Graph Annotation pattern illustrates how to capture information about these graphs. However for many use cases, a consuming application may need to ignore the graph structure of a dataset and instead

work on a single consistent view of the data. In other circumstances we may only want to query a certain set of graphs (e.g. just the data, or just the annotations). So how do we gain the benefits of named graphs without having to teach all applications about the internal structure of our triple store?

Solution

Write queries against a "union graph" that provides a simple view over a collection of named graphs by relying on RDFs data merging rules.

Example(s)

See below

Discussion

Named graphs are a useful way to structure a store to simplify the compilation and maintenance of a dataset. But we might consider that the graph structure of a dataset -- i.e. how it is composed of a number of named graphs -- to be an implementation detail that needn't concern applications built on that data.

The RDF data model provides rules for merging of RDF statements into a set. Those rules can be applied to create a simple synthetic view over a collection of named graphs that ignores the graph structure completely. A union graph is the RDF merge of one or more named graphs.

Most RDF triple stores offer the ability to create or use union graphs in some form. SPARQL also provides some options for creating additional "union graphs" (otherwise known as "synthetic" or "view" graphs).

We can think of a quad store as a set of named graphs, each of which has an identifier. All triples are therefore associated with a specific named graph. However when applying some pattern matching, e.g. in a query, or inferencing rules, we can safely ignore the graph URI component of the quad, this result in a simple triple based view of the data. This is a simple union over all the graphs. Some stores allow this union to be referenced in a query or application code, in order to allow graph-agnostic processing.

In SPARQL a dataset consists of a default graph, which doesn't have an identifier, plus zero or more named graphs which each have a URI. Triple stores that are geared towards SPARQL may similarly offer a default graph in which triples can be stored and the ability to store data in additional named graphs. TDB (part of Apache Jena) provides such a facility. Triples may be added to a dataset and these are then stored in the default, unnamed graph. In a SPARQL query, triples patterns are matched only against this graph. However TDB also offers a way to address the union graph of the store: i.e a view over all of the named graphs in the store (plus the default). TDB may also be configured to not allow updates to the default graph. In this case the default graph is automatically constructed from the union graph.

Other triple stores do not support an default unnamed graph, requiring all triples to be associated with some graph. In this case when executing a SPARQL query the default graph will be selected based on some system wide default (e.g. the union of all graphs) or by the query itself using a FROM clause.

The SPARQL FROM clause provide another way to define custom union graphs. The FROM clause is used to identify the default graph for a query. The most typical use is to identify a single RDF graph. However if multiple FROM clauses are specified in a query then the contents of those graphs are merged (typically in-memory) to provide a union graph that will form the default graph for the query. This feature of SPARQL can therefore provide another way to assemble a useful graph-agnostic view of a dataset.

The options here are on the surface confusing. But they offer some useful further options for managing and querying datasets. The important part is to understand that some options are provided by the underlying storage, whereas others are a function of the SPARQL query language:

- a triple store may manage data either as a collection of named graphs, or directly as a SPARQL dataset, i.e. a default graph plus zero or more named graphs. In the latter case triples can be added/removed from the default graph
- a triple store may provide a view over its entire contents, regardless of the partitioning into graphs. This is the most common form of union graph. It is also likely to be efficient for pattern matching, etc. This union graph may be in addition to whatever basic storage options the store provides
- a triple store may offer options for how a sparql dataset is created for a given query, or it may enforce a specific structure. E.g. a store may enforce that the default graph is a specific stored graph, or that it is a union graph providing a view of all of its contents
- a triple store may offer some additional flexibility for a sparql query to define its dataset including specifying a single graph as the default, the store-wide union graph, or a query specific union graph which is populated by merging together a number of named graphs identified in the FROM clause of the query

Not all triple stores offer this kind of flexibility but many offer at least some options for querying the whole store as a union graph.

There is scope here for further innovation on the part of store implementors, e.g. to offer additional ways to either statically or dynamically create union graphs over a store. For example an application may want to identifier the most recently updated graphs; graphs with a particular kind of content; public and "private" graphs, etc.

Related

- Named Graph
- Graph Annotation

Further Reading

- Managing RDF using Named Graphs
- TDB Datasets
- TDB Dynamic Datasets
- SPARQL 1.1 Specifying RDF Datasets

Chapter 6. Application Patterns

Abstract

Application architecture has always been a rich source of design patterns. Much of the design pattern literature covers useful architectural and code design patterns that can lead to the creation of more maintainable and evolvable software.

We are still at the early stages of exploring how best to create Linked Data applications. Indeed there is still much debate about what constitutes a Linked Data application at all. Is it any application that uses RDF, perhaps entirely based on a local triple store? Or must it be an application that is capable of continually discovering new information from across the web of data?

We still have much to learn about how to create applications that are truly flexible enough to process and display a wide variety of different data types. This covers everything from software architecture, design and user experience. Over time we might expect to see more research and development of Linked Data browsers. Or, perhaps web browsers will simply become more data aware and develop ways to help users make more of the data that is increasingly embedded in or linked from web pages.

Regardless of the type of applications we are constructing, there are a number of ways that specific features of RDF, SPARQL, or HTTP accessible Linked Data can be exploited to create flexible software architecture or simply useful behaviour. This chapter captures a variety of design patterns that relate to a number of different aspects of application development.

The existing patterns literature is at our disposal for helping to create Linked Data applications, but which features of semantic web technology can help us to better solve problems or meet requirements?

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

Named Query

How can details of the SPARQL protocol be hidden from clients?

Context

SPARQL protocol URLs quickly become complex when dealing with any non-trivial query. Very large queries can be so long that some clients or browsers may have issues with length of the URLs. The only solution in this case is to switch from a GET request to a POST request. But as a query is an idempotent operation it is better to use GET, with appropriate caching headers, rather than a POST.

In other circumstances a service might want to restrict the set of queries that can be invoked against a SPARQL endpoint. Or the use of SPARQL might be entirely hidden from calling clients. In both of those cases removal of direct access to the SPARQL endpoint may be desirable.

Solution

Assign a short URL to the SPARQL protocol request. The URL maps directly to a SPARQL query that is executed on request. Clients can use the short URL instead of the full SPARQL protocol request to extend the query.

Example(s)

An application exposes a SPARQL endpoint at `http://api.example.org/sparql` and a collection of named queries from `http://app.example.org/queries`.

One example query that clients might potentially execute is:

```
SELECT ?uri ?homepage WHERE {
 ?uri foaf:homepage ?homepage.
}
```

Rather than requiring clients to compose the full SPARQL protocol request for that URL it could instead be defined as a named query for the service. The query could be associated with the following URL: `http://api.example.org/sparql/list-homepages`. A GET request to that URL would be equivalent to the SPARQL protocol request to the endpoint, i.e. would execute the configured SPARQL query and return a response in one of the standard SPARQL protocol formats.

Discussion

Named queries is useful in a number of circumstances. Assigning short URLs to queries can remove issues with dealing with lengthy SPARQL queries that might get accidentally truncated in emails or be rejected by older HTTP clients or browsers. By providing tools for users of a service to create new named queries then a community can share and publish a useful set of queries.

Another benefit of binding queries to URLs, i.e. by creating new web resources, a service can implement additional optimisations that can improve response time of queries. E.g. query results might be generated asynchronously and the cached results supplied to clients rather than the query being executed on demand.

One way to protect a SPARQL endpoint is to reduce the legal set of queries to an approved list, e.g. that won't cause performance issues for the service. Named queries provide a way to provide a set of legal queries which are then bound to URLs. Direct access to the SPARQL endpoint can then be disabled, or limited to a white-listed set of client applications.

There are some additional nuances to consider when implementing this pattern. For example the SPARQL protocol could be extended to support Parameterised queries by injecting query string parameters into the query before it is executed. Additional parameters could be used to invoke additional kinds of pre- or post-processing behaviour including transformation of SPARQL protocol responses into alternate formats

Related

- Parameterised Query

Further Reading

- SPARQL Stored Procedure

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 critera 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

Parameterised Query

How to avoid continual regeneration and reparsing of SPARQL queries that differ only in a few bound variables?

Context

Many applications continually execute a small number of queries which differ only in terms of a few parameters. For example generating a Bounded Description of a resource might involve a standard query that varies only by the resource URI being referenced. Other queries might vary based on date ranges, page offsets, etc. Re-generating queries as text strings can be fiddly and makes for messy application code. This is particularly true when the query then needs to be serialised and submitted over the SPARQL protocol: URL encoding issues can easily cause problems. In addition a query engine incurs extra overhead when repeatedly parsing the same query.

Solution

Define each repeated query as as query template which can be parsed once by the query engine. The query should define variables for those aspects of the query that might vary. The variables can then be bound to the query before execution. Supplying the values for these parameters will typically involve an engine specific API call.

Example(s)

Apache Jena uses a `QueryExecutionFactory` to support creation of queries. Query objects can be pre-compiled. An initial set of bindings for a query can be provided in order to create a specific QueryExecution. These initial bindings are simply a map from variable name to value.

Discussion

This pattern is well understood in the SQL database world: Prepared Statements have been in use for many years. SPARQL processors are now starting to add similar features, which simplifies working with queries from application code.

At present the SPARQL Protocol does not support specification of initial bindings as additional query string parameters. However some Linked Data publishing platforms have added support for parameterised queries, as extensions to the core protocol, allowing additional query parameters to be automatically injected into the query prior to execution. This makes it simpler for users to share queries and adjust their parameters.

SPARQL 1.1 provides a `BINDINGS` keyword which can be used to declare that certain variables should be injected into a graph pattern. This was added primarily to support federated query use cases, but can also be used to support some parameterisation of queries.

A map of named-value pairs which describe some initial query bindings can also be interpolated directly into a query by looking for appropriately named query variables. This interpolation has been done in several ways:

- Using some additional custom syntax to mark up the variable, e.g. `%{var}`. This means that the query is no longer valid SPARQL into variables have been substituted
- By using the fact that SPARQL provides to naming syntaxes for variables, and defining `$var` to be those that are bound before execution and `?var` those that are bound during execution (or vice versa). This relies on local convention.
- By binding any query variable using any valid SPARQL syntax. The downside to this option is that additional external context is required to identify those variables that must be bound prior to execution. The other options allow these to be identified from just the query itself, although in practice it is often useful to be able to known the expect type, format or legal values for a parameter which will require additional configuration anyway.

Parameterised Queries are a core feature of the Named Query pattern.

Related

- Named Query

Further Reading

- SPARQL 1.1. Bindings

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 infering 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 it's own rules about what consistutes "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.

URI Resolver

How can we customize the application behaviour associated with resolving (de-referencing) a URI into RDF statements?

Context

Linked Data applications typically acquire additional relevant data by adopting Follow Your Nose behaviour: any URI in a graph may be assumed to be de-referencable to obtain additional data.

However in practice simple de-referencing, i.e. performing a GET request on a URI, is not always desirable. For example:

- An mobile application may need to work in an off-line mode where remote data is not available
- Continuous testing scenarios may need to rely on predictable data for driving test assertions and, in addition, may need to be executable in a self-contained environment without use of network services
- Security constraints may require network requests to be routed via an intermediary service
- A local mirror may be available which can provide a better quality of service
- A remote service may be intermittently or completely unavailable, requiring a local mirror to be substituted on either a temporary or permanent basis

Solution

Application code should address de-referencing requests to URI resolver. Broadly, a URI resolver is a function that maps from a URI to a stream from which RDF triples can be consumed. A URI resolver might consist of an application component or could be deployed as a network addressable service (i.e. a proxy server).

Application code should defer to the URI resolver in order to source RDF statements and provide configuration options to specify which URI resolver (e.g. implementation or service location) should be used. Simple de-referencing behaviour may still be used as fallback behaviour if no URI resolver is available.

Example(s)

A Linked Data browser loads and displays resources as directed by user behaviour, e.g. clicking on links in the user interface. A user selects to view a resource. When a user requests that the browser displays a resource, `http://example.org/person/1`, instead of performing a GET request on the resource the browser invokes a pre-configured URI resolver to retrieve the description of the resource.

The URI resolver has been set up to direct requests matching a pattern of `http://example.org/*` to a local triple store that contains a mirror of the remote data. However when the user visits `http://other.example.org/document/123` the URI resolver does not have any prior knowledge of the resource and falls back to a simple GET request on the resource URI.

In neither case does the browser (or the user) need to know how the description was actually retrieved.

Discussion

Adding some extra indirection around the de-referencing of URIs into RDF statements provides some much needed flexibility when dealing with network issues such as intermittently available connections; unreliable remote services; and security constraints. Applications that support the configuration of URI resolvers provide options for customising and optimising application behaviour based on local requirements.

URI resolvers are not a new concept and have been used in many different systems. SGML and XML processing pipelines typically support URI resolver components to allow resources to be cached locally or even bundled with an application. More broadly, HTTP proxy servers fulfill the role for general web requests.

The indirection offered by URI resolvers make them an ideal location in which to provide additional behaviour. For example all of the following can be implemented using a URI resolver component:

- Caching of RDF descriptions as they are retrieved, e.g. in an in-memory, file system, or document store
- Substitution of a local mirror of the data in preference for the remote version
- Substitution of a local mirror of the data in preference for the remote version, but only where the remote service is unavailable
- Serving of a fixed response, regardless of URI (e.g. to support testing scenarios)
- Retrieval of both the remote description of a resource plus local annotations to mix public and private data
- parallel retrieval of the description of a resource that is spread across any combination of local or remote locations
- Provision of reasoning over retrieved data to augment data against a vocabulary
- Provision of support for resolution of non-HTTP URI schemes
- On-demand conversion of non-RDF data into RDF statements

With suitable configuration, URI resolvers can potentially be chained together to create a de-referencing pipeline that can deliver some complex application behaviours with a simple framework.

There are some Linked Data applications that provide URI resolver services, this includes generic Linked Data browsers. At their simplest the browsers simply provide additional HTML presentation of retrieved data. But in some cases the retrieved data is both directly accessible (i.e. the service acts as a proxy) and may be supplemented with local caches, annotation, or inferencing, as outlined above. To support de-referencing typically use a Rebased URI

Related

- Follow Your Nose

Further Reading

- Diverted URI pattern
- The Jena FileManager and LocationMapper
- Entity management in XML applications