SPARQL Diamonds

an architectural pattern for rapid development of data-driven applications

Danny Ayers, 2021
hyperdata.it
danny.ayers@gmail.com

Abstract

This document proposes a software-architectural pattern as a means of providing solutions to recurring application development demands in the context of the Web of Data. As such a pattern it could be described in the abstract, but it is felt that it has most utility in concrete scenarios involving the Web browser in conjunction with SPARQL servers and the Web at large. With this in mind some simple applications for this environment will be described.

The infrastructure components of a 'SPARQL Diamond' are a SPARQL server and a Web browser. A text templating engine intermediates SPARQL queries from the user interface and the server endpoint (request). Templating is again applied between the server (response) to deliver desired HTML rendering of the results.

The primary intent here is to demonstrate that data-driven applications can be created with minimal effort, given a starting point for development. Reference will be made to the current paucity of adoption of Semantic Web technologies, with the proposition that the description of simple patterns may reduce the demand on developers.

Keywords: SPARQL, RDF, Web, software pattern, Wikidata

Introduction

The phrase 'The Semantic Web' has been fairly widely known in the Web development community since at least the 2001 publication of the Scientific American article of that name [1]. The core idea is to evolve the World Wide Web from a primarily document-oriented system to one which could manage arbitrary data using the same fundamental concepts (notably resources with their identifiers and representations as described in the HTTP protocol [2]). A more intuitive label for this evolution was later coined : 'Linked Data' [3].

In 2001 the necessary technologies were still in their infancy so it is no surprise that there was little adoption at the time. But with the SPARQL 1.1 specifications [4] achieving W3C Recommendation status in 2013, arguably all the pieces were in place for general adoption. However, relative to other contemporary developments in Web technology, uptake has been minimal.

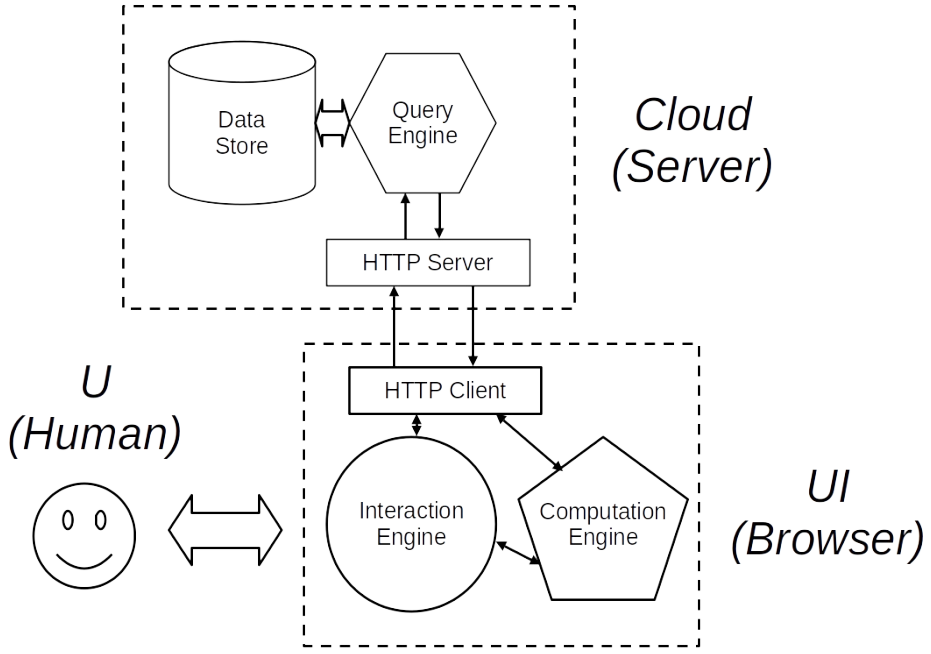
There are no doubt many factors that have contributed to this state of affairs, but one is fairly certain : the perceived complexity of developing a Linked Data-oriented application. In the early days at least, this was a fair perception : the benefits could be obscured by having to deal with a confusing format (RDF/XML) and the mere existence of more arcane technologies such as OWL and its associated reasoners may have overwhelmed the developer. Where there has been marked success it has often been somewhat through the back door, notably with site developers exploiting page metadata to provide *Search Engine Optimization*. The inclusion of, for example, schema.org terms in a HTML page can improve that page's visibility. Meanwhile the schema.org vocabulary is based on the RDF data model, developed using RDF Schema.

The technique to be described has no doubt been used elsewhere before, the novelty offered here is to examine it as a potential pattern, a starting point, a means of reducing the cognitive load on the developer.

Source code for the demonstrations described here and other related resources are available at [5].

Operating Environment

The environment of SPARQL Diamonds is shown in Fig. 1.



This is very similar to a first-generation database-backed Web site, viewed in a browser. The key difference is that instead of the query and presentation construction being carried out almost entirely server-side, here when working against a SPARQL Server, more of the work is carried out at the client. This general architecture is not unknown nowadays, some sites use a 'rich client' dynamically processing interactions through an API to the server.

Although the block labeled *Cloud (Server)* shows typical SPARQL Server internal subcomponents, the details of implementation are irrelevant. All that matters is some degree of support for the SPARQL Protocol [6]. In broad terms the *Interactions Engine* is the browser HTML DOM (and it's representation to the end user), the *Computation Engine* being the Javascript interpreter found in modern browsers.

As a Web-based system, SPARQL Diamonds are clearly dependent on the HTTP protocol. More specifically, they are inclined towards the REST [] architectural style. [1]

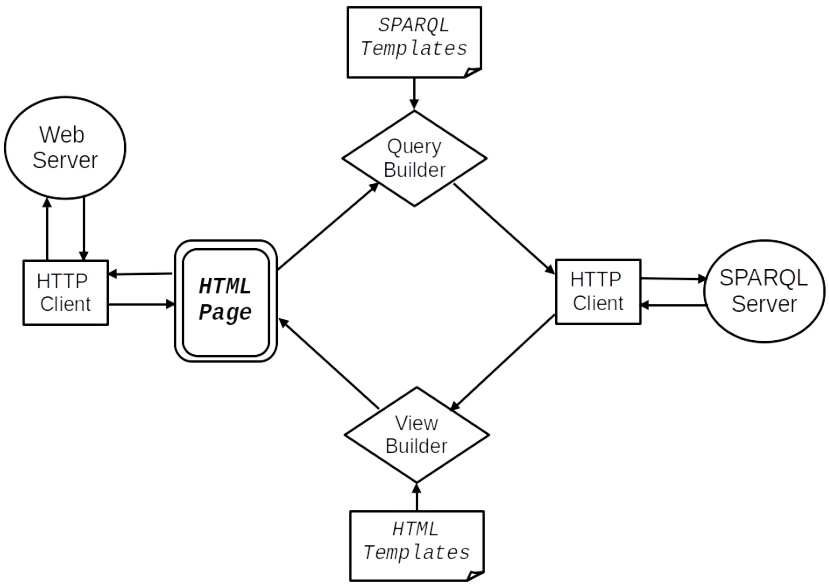
1 *Strictly speaking, SPARQL servers fall outside of the REST style, due to the use of the SPARQL Query Language inside the query part of URLs rather than REST's notion of potentially opaque identifiers that act purely as names for given resources. But in practice this can be viewed as an implementation detail - the combination of endpoint URL with query could be interpreted as merely an arbitrary string identifying a resource, or even aliased from a more human-readable URL.*

Client-side, the configuration described here will be that of a typical Web browser. But this doesn't preclude the use of the Diamonds pattern in a different environment such as a dedicated App for a mobile device. The only requirements are the SPARQL server, a HTTP client, a processing component to manipulate data and control its flow and some means of interacting with the end user.

In a practical, live application, access control must be considered. This is orthogonal to the topic here. The vulnerable surface of SPARQL Diamonds is virtually identical to that of most Web service systems, standard approaches to security may be applied.

The Diamond

Figure 2 shows the SPARQL Diamond pattern in graphic form. On the left, *Web Server*, *HTTP Client* and *HTML Page* are as found in a typical Web document setup. A browser is given the document URL, the HTML representation of that resource is retrieved from the server by the HTTP client and displayed. The blocks to the right form the SPARQL Diamond – the symmetries suggested the name.



As with most modern Web pages, Javascript in the HTML is called to provide dynamic features (whether anything occurs on page load will be application-specific). Typically the user will provide input data via form elements.

This will be processed by the *Query Builder*. This is a text templating engine supplied with a preset template. Named fields within the template will be given values dependent on the user input yielding a valid SPARQL query. In many cases, named fields and their values may be taken directly from the user input form.

This query will be formatted as a HTTP Request and passed through a HTTP client to the *SPARQL Server*.

On receiving a response from the server, its contents will be passed to a *View Builder*. Due to the structure of SPARQL results some transformation of this data is likely to be necessary before passing it to the templater (in practice will be the same templater as above, in a different role). Where the *Query Builder* takes user input and combines it with a template to create a SPARQL query, the *View Builder* takes the query results and combines these with a template to provide a HTML representation of those results.

According to the application requirements, this will then be presented to the user, typically by insertion in the HTML document object model (`DOM`).

Requests and Responses

The SPARQL specifications support a wide range of facilities that comfortably cover requirements for data store manipulation. Only a tiny fraction of the potential techniques are needed to describe SPARQL Diamonds (and likely the majority of use cases). Those few techniques, in broad brush-strokes:

SPARQL queries are delivered and results returned using HTTP as a transport (the alignment with HTTP and the REST architectural style does go further, there should be few surprises). The text nature of HTTP and the developer-oriented tools within browsers makes monitoring and debugging relatively straightforward.

From a client's request over HTTP, a SPARQL server will take the query, apply it to a datastore and return the corresponding results over HTTP. In a typical setup, leveraging the methods of HTTP, an idempotent SPARQL `SELECT` may be applied using a HTTP `GET`. A SPARQL `INSERT` or `DELETE` may be passed via a HTTP `POST` to modify the contents of a store.

Following the HTTP protocol, on addressing the target URL, the method and other headers have great significance. Proper discussion of these is well out of scope here, the relevant specifications should be consulted for the details. But two particular phenomena may confuse the developer:

Firstly, while the HTTP protocol is clear when it comes to the media types requested and those of the content actually delivered (generally through `Accept:` and `Content-Type:` headers), certain systems such as proxies may not respect these fully. As a workaround it is common to see a query URL having an extra parameter, for example:

```
http://example.org/sparql?output=json&query='…'
```

While inelegant and redundant, the `output=json` parameter is an extra hint to the server of what format results are required.

The second point of note is an issue that can crop up with any Web services , CORS (Cross-Origin Resource Sharing) [8], a security mechanism through which a browser and server negotiate the allowed origins of resources to load. Even if the Web server and SPARQL server engaged in SPARQL Diamond-style operations are on the same host, the difference in port for each may cause immediate failure at the browser. The solution is a straightforward addition of headers (this is covered by default in the Fuseki SPARQL server used for the bookmarks demo here, but foreknowledge can save a lot of developer time).

The implementation to be described here has the Diamond built in conventional Javascript using the browser's primary HTTP client to retrieve the initial view as per usual. The SPARQL Server interactions are achieved programmatically through Ajax (originally from 'Asynchronous JavaScript and XML') and view modifications through programmatic manipulation of the HTML DOM. As with a large proportion of current Web sites, rather than addressing the DOM and using `XMLHttpRequest` objects directly, the jQuery Javascript library [9] is used to provide a more developer-friendly programming interface. Many sophisticated Javascript frameworks exist to aid Web development, often offering template rendering. Another very common feature being the automatic binding of user interface elements to Javascript models. The Diamonds pattern is framework-agnostic, but to simplify bindings without obscuring the functionality too much, jQuery seemed a reasonable compromise for the the demonstration code.

Templating Engine

Templates are incredibly useful in every industry and within software; text search-and-replace is an invaluable tool. Given that the protocols and formats around the Web can generally be expressed as text, templating should be useful. Take for example the following :

```
<a href='~{link}~'>~{link}~</a>
```

The character string here is HTML, but on rendering directly would make little sense. But with replacement using predetermined regular expressions, where *link* is a named string variable and `~{ }~` are chosen delimiter sequences, it can become :

```
<a href='http://example.org'>http://example.org</a>
```

In the same fashion, this :

```
SELECT ?x WHERE {
   ?x rdfs:label "~{label}~"
 }
```

Can become :

```
SELECT ?x WHERE {
   ?x rdfs:label "Cat"
}
```

In contemporary Web browsers, such replacement is trivial – Javascript supports string replacement natively, but it is extremely easy to make mistakes in such situations, handling text naively (SQL Injection attacks being a case in point). Fortunately libraries have been developed to overcome most mishaps.

A community specification, 'Mustache' [10] was used here, so-named for the style of delimiters : `{{ ... }}`. Alas, these delimiters can occur as parts of SPARQL queries. Fortunately, Mustache, as most libraries, allows for arbitrary delimiters. (The Hogan [11] library from Twitter, implementing "logic-less templates" was used in the demos). As seen above, the unlikely character sequence of `~{ … }~` was chosen to avoid conflicts with SPARQL syntax and reduce confusion in the eyes of the developer.

The data to be inserted into templates is provided as simple Javascript objects. In both examples above, a dictionary containing a single name-value pair has been supplied. For the HTML templating :

```
{ link: "http://example.org" }
```

and for the SPARQL templating :

```
{ label: "Cat" }
```

A dictionary of an arbitrary number of name-value pairs may be supplied, making it convenient to fill in many fields at once. Lists may also be used with the Mustache specification, as described below.

Data Transformation

Where there is a direct correspondence between fields in a user input form and the variables of interest in a SPARQL query, data preparation can be very

straightforward : step through the names/ids of the form fields and their correspond values (as entered by the user) and add these as pairs to a dictionary.

The Mustache specification provides useful handling of list structures and can even apply template-defined functions (lambdas) to the data. The extent to which these facilities, or comparable facilities of other templaters are used is down to the developer's preference. (For the demonstrations only simple search/replace and list were used).

Data provided by a HTML form will typically require little or no preprocessing for query inclusion, as it occurs as a set of name-value pairs already, essentially the shape the templater requires. However SPARQL results data is rather more complex.

By way of example, if the Wikidata Query Service [12] is given the following SPARQL :

```
PREFIX schema: <http://schema.org/>

SELECT DISTINCT ?link WHERE {

    ?link schema:name "Knot"@en .
}
```

The results (via *Downloads, JSON File (verbose)*) look like this:

```
{
  "head" : {
    "vars" : [ "link" ]
  },
  "results" : {
    "bindings" : [ {
      "link" : {
        "type" : "uri",
        "value" : "https://commons.wikimedia.org/wiki/
Knot"
      }
    }, {
      "link" : {
        "type" : "uri",
        "value" : "https://en.wikipedia.org/wiki/Knot"
      }
    } ]
  }
}
```

For a simple case such as this, a procedural approach seems perfectly valid. Given the results above in a JSON object **`json`**, a list of bindings may be extracted with the Javascript :

```
bindings = json.results.bindings
```

That list may be iterated through to extract the individual values:

```
value = bindings[i].link.value
```

Which may each then be formulated into a dictionary:

```
pair = {link: value};
```

- and passed in turn with the link template as above to produce a series of HTML-formatted links.

Depending on the functionality of the templating engine, some of the formatting may be deferred to the template. To render lists, a Mustache template will be of the form :

```
~{#links}~
  <a href='~{link}~'>~{link}~</a>
~{/links}~
```

To fill the fields here the data will be of the form :

```
{
    links: [
      {link: 'https://commons.wikimedia.org/wiki/Knot'},
      {link: 'https://en.wikipedia.org/wiki/Knot'}
    ]
}
```

Yielding (as in the more procedural case) :

```
<a href='https://commons.wikimedia.org/wiki/Knot'>https://
commons.wikimedia.org/wiki/Knot</a>
<a href='https://en.wikipedia.org/wiki/Knot'>https://
en.wikipedia.org/wiki/Knot</a>
```

With minimal additions, this can appear in a browser as Fig. 1.

Figure 1: From wikidata-knot.html at https://github.com/danja/sparql-diamonds

Other work has been done on transforming one JSON data structure into another, eg. [13] and specifically as applied to SPARQL results at [14]. Work is ongoing in the context of SPARQL Diamonds to develop support code to simplify transformations. But as suggested above, in practice the ad hoc creation of per-case transformations is straightforward and even when handled as inline procedural code such manipulation is relatively easy.

Completing the Diamond

In the example above the whole query is predetermined. But by extending the same templating approach it is straightforward to incorporate user input.

If a template is given as:

```
PREFIX schema: <http://schema.org/>

SELECT DISTINCT ?target WHERE {
    ?target schema:name "~{name}~"@en .
    ?target schema:about ?concept .
}
```

- the placeholder in the template may be replaced.

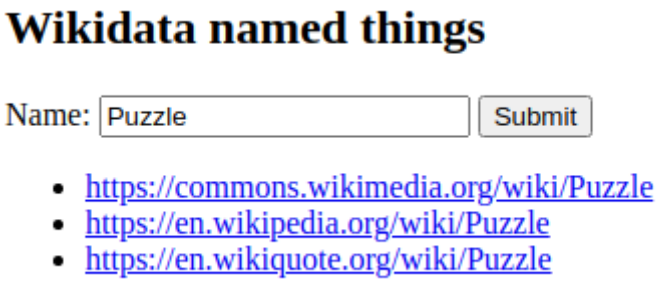
Dynamically pulling out user input from a form field is a common technique, straightforward with the aid of jQuery. Once the value has been obtained it needs to be associated with the field it has to replace, and the template applied. In the Javascript, this can be achieved with:

```
var dict = { name: name }

var wikidataNameSparql = templater(wikidataNameTemplate,
dict)
```

Note that the left-hand 'name' will be associated with the variable identifier, the right-hand 'name' its value.

With the SPARQL query completed, it may be sent to the server in exactly the same way as in the case of the preset query above.



Modifying Data

The examples described above project pre-existing data found in a remote store to the browser. While this may be useful, it is clearly limited. The capabilities offered by SPARQL mean that modification of store contents is possible. With the SPARQL Diamond pattern, exactly the same technique as described above can be used for this purpose. By way of example, a simple interactive application demonstration has been built.

It's common to bookmark Web pages of interest in a browser, but the bookmark manager in browsers is typically very primitive. Typically the link and its title is stored locally in a simple hierarchical structure, without any facility to augment it. The RDF model provides an excellent means by which information about a resource may be augmented with related material.

Here we present a simple bookmarker.

To provide context for what follows, here is a screenshot of the bookmarker demo:

On the left is a list of links bookmarked by the user. On the right (on the same page in this setup) is a form through which the user may add a bookmark.

The RDF model underlying this may be seen by example (in the Turtle [] syntax) :

```
@prefix dc:   <http://purl.org/dc/terms/> .
@prefix foaf:   <http://xmlns.com/foaf/0.1/> .
@prefix x:     <http://purl.org/stuff/> .

<http://hyperdata.it/bookmarks/1394a43c>
      a x:Bookmark ;
      x:url <https://en.wikipedia.org/wiki/
Exquisite_corpse> ;
      dc:created "2012-04-17T16:06:30Z" ;
      dc:title "Exquisite Corpse" ;
      dc:description "Wikipedia page on a surrealist
game" ;
      foaf:maker
            [ foaf:nick "danja"] ;
      x:tag "surrealism" ;
      x:tag "art" ;
      x:tag "game" .
```

Three vocabularies are used : that of the well-known Dublin Core Metadata Initiative [ref], FOAF [ref] and a sandbox vocabulary space maintained by the author. The bookmark resource instance, the object of interest, is identified with a URI which is potentially dereferenceable (URL). It is typed as a bookmark object (in Turtle the `a` token is interpreted as `rdfs:type`).

The bookmarked link is given as a property attached to the bookmark instance, as are the literal annotations of `created`, `title` and `description`. Rather than provide an explicit identifier for the creator of the bookmark, this is made indirect through a bnode in the `maker` → `nick` (= nickname) relationship.

Although RDF(S) has constructs for creating sets and lists (Containers and Collections), the set of *tags* (associated keywords) is given by implication through their shared property, `tag`. This is logically valid, as well as being pragmatically convenient, Containers & Collections tend to complicate matters.

Referring back to the screenshot of a browser, beginning at the right, on clicking 'Submit' the form data will be obtained as variables in browser Javascript. Most of the fields are name-value pairs. With a view to using this with a templater, it makes sense to maintain this structure. If we let the attributes in the fields in the form echo the subsequent intention:

```
...
<input id="title" type="text" />
<textarea class="description" id="description"></
textarea>
…
```

- after the user has submitted their input, it may be picked up and structured appropriately with very little code:

```
$('#fields :input').each(function (index, field) {
   bookmarkData[$(field).attr('id')] = (field).val()
 })
```

This is arcane, idiomatic jQuery/Javascript (marginally less ugly than addressing the DOM directly). But its operation is simple – pull the names and (user input) values from the form, create a mapping like:

```
{
   "title":"<user input>",
   "description":"<user input>"
}
```

As described earlier, exactly the object structure that can be applied to a template. But things aren't quite that simple. As evident in the Turtle, not all the required data is immediately available as flat name-value pairs.

Critically, the whole bookmark object is given an identifier. There is little to be gained by giving this much significance, but rather than assigning a random string, something based on the bookmark data may be useful. In the demonstration this is achieved by appending a hexadecimal hash of the link URL to a base URL (in a controlled domain).

The date/time of the creation of the bookmark is useful information. This is generated by using Javascript functions to create a standards-friendly string (ISO/W3CDTF) of the time of posting.

Another special case is the set of tags, which the user will submit as a comma-separated list. Getting the individual string values is trivial in Javascript (`split` on the comma yields an array, `trim` on those elements removes whitespace – further validation is optional). Placing that into a shape a name-value template might require might demand creativity, but the Mustache template specification simplifies this. Given data of the form:

```
            {
            "bookmark":"http://hyperdata.it/bookmarks/
1394a43c",
          "tags":[{"tag":"tag1"},{"tag":"tag2"}]
          ...}
```

the template:

```
~{#tags}~
          <~{bookmark}~> x:tag '~{tag}~' .
~{/tags}~
```

- will render suitable triples of the form :

```
<http://hyperdata.it/bookmarks/1394a43c> x:tag 'tag1' .
<http://hyperdata.it/bookmarks/1394a43c> x:tag 'tag2' .
```

At this point it would be fair to say that the apparent elegance of the SPARQL Diamonds pattern breaks down somewhat. But any other approach would demand the same attention to special cases of mismatch between the user interface and the data model. Having the majority of data mappings handled in a generic manner highlights where the exceptions are. In practice, there are few general kinds of exceptions, and those exceptions can be dealt with with relative ease once isolated.

After the SPARQL `INSERT` query has added data to the store, the display of contained data of relevance is managed in exactly the same way as the Wikidata examples. A little transformation followed by templating. But here again there is a special case : the `created` field is stored as an ISO string, eg. `2021-04-17T16:06:30Z`. This is not human-friendly. But open source libraries are available to reformat this with a single function call. (The moment [] library is used in the demos).

As a small addition to the bookmarks demonstration, a 'bookmarklet' [] was created. These are browser-bookmarkable links which contain Javascript. While minimal, hopefully it shows a genuinely useful bookmarking tool is possible. Here, on clicking the bookmark button in a browser, the script will take the URL of the current page, redirect to the bookmarking page and insert the page's address into the link field of the form. Given that fairly arbitrary Javascript may be run on loading the bookmarking page, a HTTP `GET` could be applied, the HTML `<title>` and other metadata information extracted and use to pre-augment the bookmark data.

Alternate Approaches

The SPARQL Diamond is obviously only one of a multitude of possibilities for interfacing with an online data store. One configuration deserves special mention, that of maintaining an RDF model at client. Several RDF API libraries are available [https://www.w3.org/community/rdfjs/wiki/Comparison_of_RDFJS_libraries]. This approach offers considerable advantages in terms of flexibility, for example with a SPARQL CONSTRUCT request, the resultant RDF can be merged directly into the local model. However this flexibility is typically at the cost of increased programming complexity.

Another set of solutions are possible by employing a server-side system between the SPARQL server (or other RDF store) and HTTP interface, with the browser being solely for user interface. [2]

Pros and Cons

The benefits of SPARQL Diamonds largely stem from the decoupling of the query and presentation phases of development from the operational wiring. A (very loose) analogy can be made with the *Model-View-Controller* of object-oriented programming. The model is implied by the query template (and instantiated by the results). The view is defined by the HTML template. The controller is determined by a handful of functions within the page Javascript (calling background libraries) which typically will require little or no change across application scenarios.

The good things here can be itemised :

- Simplicity : the developer can focus on formulating queries and result presentation, the important material
- Largely declarative : the data and its presentation are primarily determined by the templates
- Relatively fast and lightweight in operation
- Compatible with well-known techniques and standard libraries

There are many aspects of the SPARQL Diamonds pattern that may render it inappropriate in a given circumstance. Biggest of all, the browser is *RDF-unaware*. The data structure operated on by the client is that of JSON rather than RDF meaning that true Web of Data graph manipulation techniques aren't directly available. Code libraries do exist for retaining an RDF model in the browser (eg. []). In this context not only could a client exchange data with a SPARQL server using a shared RDF model (persisting in local storage as required), but non-RDF-aware services could have their data reformulated in a more Linked Data-consistent fashion.

2 The 'Diamonds' approach described here originated in an experimental server-side content management system [https://github.com/danja/seki]. After the 'lightbulb moment' of realising the processing could be shifted to the browser, a simple Wiki was built in this fashion [https://github.com/danja/foowiki], with a significant reduction in infrastructure requirements and considerably reduced development and deployment time.

The SPARQL Diamonds pattern is merely offered as one possible bridge between the predominant paradigms for Web services and (to selectively quote the W3C) perhaps helping it on it's way to its full potential.

Some of the disjunctions between popular Web development and the notions of Linked Data have been mentioned above. It would be irresponsible not to mention how working in practice with the latter differs from the former. Typically a developer will have a predetermined schema to work against. Not 'everything', which is the Web beyond their application.

The Wikidata knowledgebase addressed in the demonstrations above stands as an example of *Current Best Practice*. The developers have offered many routes into the data, not least through sample SPARQL queries in a user-friendly interface.

Yet a difficulty lies here, by the very nature of open-ended, Web-scale information. In the development of the demonstrations here, around the same amount of time was spent in haphazard queries to explore the data as was in coding a page that showed it. By the same token, discovering an arbitrary data model as offered by a typical API requires prior knowledge of its schema. Where the Linked Data approach leads here is by having a means by which to discover new information. The same 'follow your nose' protocol as the Web dictates.

Future Work

Three related avenues for future work are suggested. The first, most apparent and practical, is to hone the implementation as described here. The obvious way of doing this is to build applications using it, find what works and what doesn't. Integration with popular frameworks such as React [] could be productive. Secondly, it may be useful to explore potential deployment scenarios with different infrastructure in which the SPARQL Diamonds pattern may be useful, such as in server-based systems and mobile device Apps. But perhaps much more importantly is to discuss the situation as alluded to in the Introduction here, whereby 'regular' Web programmers encounter incongruities between the techniques they are familiar with and those of Semantic Web technologies. It is reasonable to believe that the identification and documentation of architectural patterns in object-oriented programming served to save many thousands of programmer-hours in that context. Perhaps a similar approach could be useful with Linked Data.

References

[] Berners-Lee, "Design Issues : Linked Data", 2006. https://www.w3.org/DesignIssues/LinkedData.html

[] jQuery Javascript library. https://jquery.com/

[] IETF RFC 8259, The JavaScript Object Notation (JSON) Data Interchange Format. https://datatracker.ietf.org/doc/html/rfc8259

[] Mustache template specification. https://mustache.github.io/

[] Lisena & Troncy, "Transforming the JSON Output of SPARQL Queries for Linked Data Clients", 2018. https://www.eurecom.fr/fr/publication/5520/download/data-publi-5520.pdf

[] json-to-json-transformer Javascript library. https://github.com/the-leapofcode/json-to-json-transformer

[] Wikidata Query Service. https://query.wikidata.org/