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 can 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 two 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 approach could potentially be used to create content-delivery sites with the same general functionality as typical database-backed systems. However it is felt that it particularly lends itself to rapid prototyping and the enhancement of existing pages with 3rd party information.

The primary intent here is to demonstrate that by exploiting widely-deployed systems together with a small number of utility libraries to implement the SPARQL Diamonds pattern, data-driven *applications can be created with minimal effort. Reference will be made to the current paucity of adoption of Semantic Web technologies, with the proposition that the description of simple patterns can 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. 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 [https://httpwg.org/specs/ HTTP Documentation : IETF RFCs and IANA registries]). A more intuitive label for this evolution was later coned : ‘Linked Data’.

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 [https://www.w3.org/TR/sparql11-query/] 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. 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.

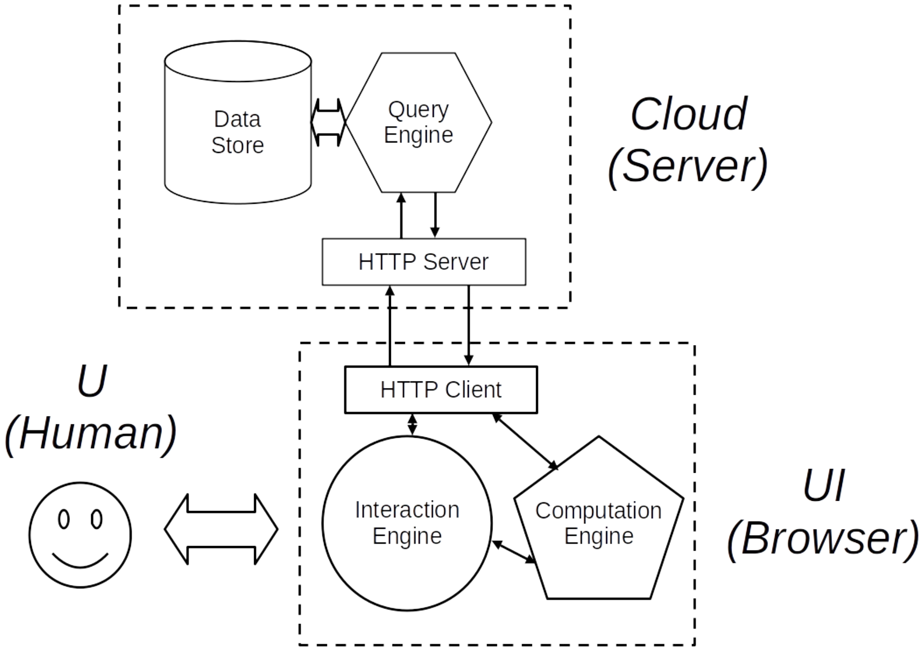
> *A secondary goal is to make life easier for developers unfamiliar with Semantic Web technologies by minimizing incongruencies*

The technique has no doubt been used elsewhere before,

https://github.com/danja/sparql-diamonds

Operating Environment

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



This is very similar to a traditional database-backed Web site.

The key difference is that instead of the query and presentation construction being carried out server-side, here it is carried out against a SPARQL Server, that work is carried out at the client. Although the block labelled *Cloud (Server)* shows typical internal subcomponents, the details of implementation are irrelevant. All that matters is some degree of support for the SPARQL Protocol [https://www.w3.org/TR/sparql11-protocol/]. In broad terms the *Interactions Engine* is the HTML DOM (and it's representation to the end user), the *Computation Engine* being the Javascript interpreter typical of current browsers.

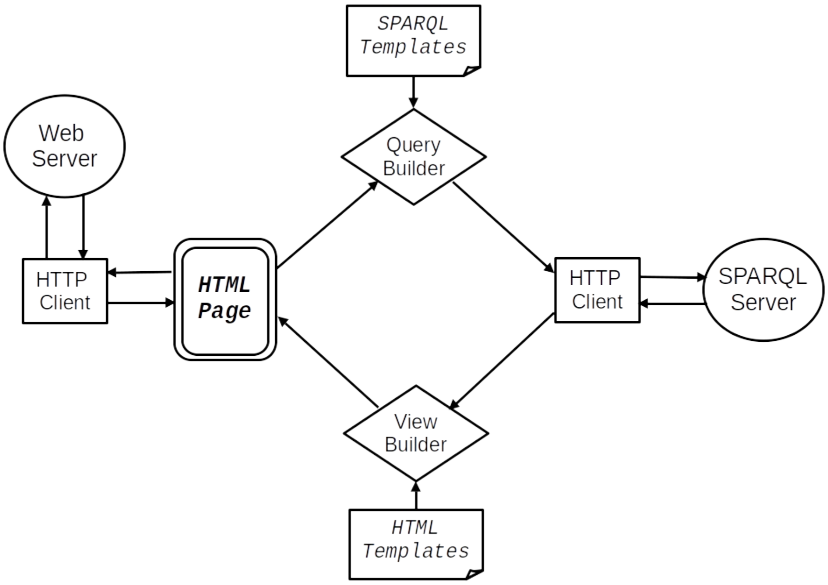
As a Web-based system, SPARQL Diamonds are clearly dependent on the HTTP protocol. More specifically, they are inclined towards the REST [https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm] architectural style. [1]

[1] *Strictly speaking, SPARQL servers fall outside of the REST style, notably 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*

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 a HTTP client, a processing component to manipulate data and control its flow and some means of interacting with the end user.

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 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

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.

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 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`).

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 [ref] is used to provide a more developer-friendly programming interface.

Linked Data

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 – contemporary 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, 'Moustache' [ref] was used here, so-named for the style of delimiters : {{ … }}. Alas, these delimiters can occur as parts of SPARQL queries. Fortunately the Hogan [ref] library from Twitter, implementing "logic-less templates" (essentially text search & replace) allows for arbitrary delimiters. So as seen above, the unlikely character sequence of ~{ … }~ was chosen to avoid conflicts with SPARQL syntax, or 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.

Additionally, 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 described here only very simple search/replace was 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 [ref] 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.

While the syntax involved in addressing parts of JSON objects can get confusing, given that JSON SPARQL Results all have a common structure, the coding becomes trivial with the aid of copy & paste.

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}~
```

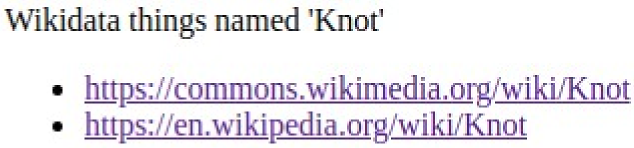
To fulfil the fields here the data will be of the form :

```
{
    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.



Wikidata things named 'Knot'

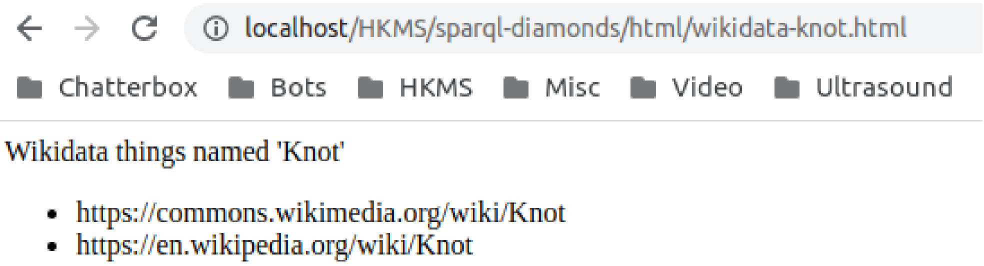
- https://commons.wikimedia.org/wiki/Knot
- https://en.wikipedia.org/wiki/Knot

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. [ref] and specifically as applied to SPARQL results at [ref]. 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.

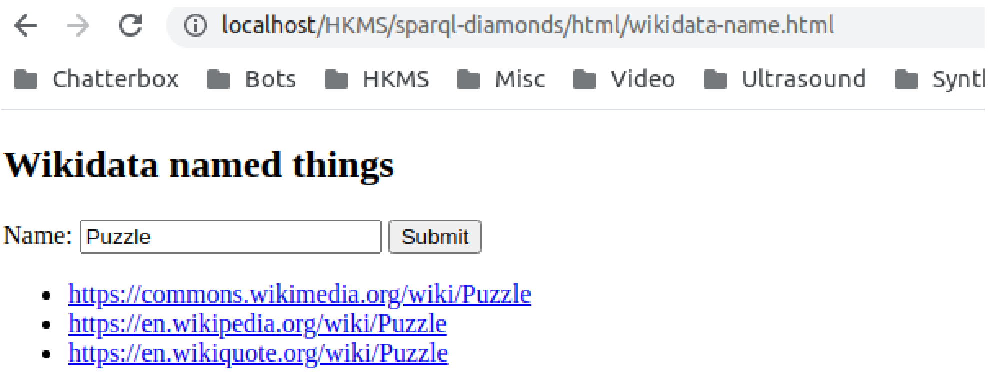
Implementation

// replace



Wikidata things named 'Knot'

- https://commons.wikimedia.org/wiki/Knot
- https://en.wikipedia.org/wiki/Knot



Wikidata named things

Name: Puzzle Submit

- https://commons.wikimedia.org/wiki/Puzzle
- https://en.wikipedia.org/wiki/Puzzle
- https://en.wikiquote.org/wiki/Puzzle

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.

// footnote In fact 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.

how do other people do it?

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.

- Simplicity : the developer can focus on formulating queries and result presentation
- 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 (eg. jQuery)

RDF-unaware : the data structure operated on by the client is that of JSON rather than RDF meaning that graph manipulation techniques aren't available

Most of the time is spent navigating the data

error handling

security

Future Work

Three related avenues for future work are suggested. The first, most apparent and practical, is to hone the implementation as described here. Secondly, it may be useful to explore other potential deployment scenarios in which the SPARQL Diamonds pattern may be useful, such as in server-side 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/

[] 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/