Fusepool P3 Architectural Principles

Basic architectural design choices for Fusepool P3

Reto Gmür

0.7 2.2.2014

Bern University of Applied Sciences

TI

Contents

1 Introduction

This documents aims to describe the basic architectural design choices and their motivation for the Fusepool P3 software. According to the DoW (Description of Work) Fusepool P3 develops a set of integrated software components. The DoW mentions different technologies (RDF, Sparql, LDP) and tools that will be used (Marmotta, Clerezza, Stanbol, OpenRefine) that will be used. For the interaction of the components the DoW specifies that the server side Java components will use OSGi and that all components must be able to consume and produce RDF through a REST API [DoW 2.1]. At the kickoff meeting if has been argued that if the component interacting via REST API are small enough componentization via OSGi may not be needed.

This would be an advantage in so far as the interactions standards between components could be standardized across all components independently. This document aims to show how such an interaction can be defined for adoption within the Fusepool P3 project.

While the concrete interaction between components should be specified within the respective work packages this document should shall specify the overall patterns within which such interaction is defined.

The goal of this document is to provide architectural principles so that:

- Reusability of Fusepool component is maximized
- Distributed development is facilitated
- It is easy for developer to understand and extend the software
- Ensure longevity of the software
- The software is simple to deploy and maintain in organizational IT environments [DoW 1.2.2]

2 Basic component interaction patterns

The interaction between components shall be defined using one of the following methods:

- Semantic REST API
- Linked Data Platform
- SPARQL 1.1 Protocol

This three basic architectures require different levels of specification to define the concrete interaction between two components.

2.1 Semantic REST API

The term REST was introduced by Roy Fielding in his dissertation in 2000. According to this dissertation REST is defined by four interface constraints: identification of resources; manipulation of resources through representations; self-descriptive messages; and, hypermedia as the engine of application state[1].

[1] http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#sec_5_1_5

To consume a REST service one should need nothing more than an entry-URI and a common media-type. Roy Fielding writes[2]:

> *A REST API should spend almost all of its descriptive effort in defining the media type(s) used for representing resources and driving application state, or in defining extended relation names and/or hypertext-enabled mark-up for existing standard media types.*

In our case we will not need to define new media types as the resources shall be represented with RDF Graphs for which serialization formats and respective media types are defined. Consistently with the LDP specification our server side services shall support at least text/turtle. Some client side components might require RDFa. The "extended relation names" Fielding mentions are in our case the ontology (properties and classes) used in the RDF Graph. The client does a GET on the Rest Service URI and its gets a graph describing the service. The client should not need to know about various HTTP URI and how the API should be invoked the client need only to understand Turtle or another supported RDF formats and the ontology. So client and server need only to agree on the ontologies used and the Entry-URI.

2.1.1 Example REST API definition

[TODO]

2.2 Linked Data Platform

The Linked Data Patform (LDP) is a specification currently being developed by a W3C working group. The specification shall describe RESTFull application integration patterns using read/write Linked Data. The API described in the current draft pretty much conforms to the Semantic REST API principles described above. The main requirement could be conveyed in the ontology of the used terms.

The specification doesn't fully describe the behavior of a server but rather constrains the behavior thus facilitating interaction. For our purpose it is a set of design principles to follow were they can be used to implement our usecases. Where the LDP specification doesn't provide specification we should complement the API following Semantic REST API principles.

2.3 SPARQL 1.1

SPARQL is a language to query and manipulate RDF Databases. Whenever components directly interact with an RDF Database, i.e. there is no other component digesting and preparing the data for the specific client the SPARQL query language and the HTTP based SPARQL protocol shall be used.

In the past many vendor or programming language specific methods have been used to access RDF databases. In fact till SPARQL 1.1 which was released in 2013 using such APIs were the only means to manipulate the data in RDF databases. The usage of such an API (the Sesame API) has been suggested at the kickoff meeting as a project partner has already developed a considerable amount of code using this technology. Because of this the postulate to use only SPARQL to access the database is probably the most controversial architectural principle in this document. It is thus worth to explain the reasons for this choice.

[2] http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

2.3.1 Modularity of Software

For programs running exclusively on the Java Platform modularization can be achieved using technologies such as OSGi or the Java Module System (JSR 277). If an application should span different platforms modularization can be achieved by developing components communicating with each other over network connections. This is typically implemented using Web Services and following a Service-oriented Architecture design pattern.

For the Fusepool P3 the consensus at the kickoff workshop seemed to be that we do not adopt a modularization approach within the Java runtime environment (such as OSGi) but instead have different small components which may or may not be implemented in Java which communicate and interact using semantic RESTfull APIs or other standardized HTTP based protocols.

Communication via a Java API using classes and methods represents a much tighter coupling than communication via a REST like or SPARQL interface with fixed verbs. In fact the technologies Bill de hÓra recommends to use such API only for computation or where significant state management is required[3] this is not the case for the RDF data storage of the envisaged web applications and services.

Using SPARQL means having a clear separation between the storage backend and its client. The two communicate via a standardized protocol with a narrow set of verbs, this satisfied the postulate of loose coupling between components.

2.3.2 Premature optimization

A main motivation for tight coupling is an alleged efficiency benefit considered to outweigh the advantages of loose coupling such as adaptability and interoperability. However Turing Award Winner Donald Knuth wrote that premature optimization is the root of all evil. This is because performance consideration often cause the design and thus the manageability of an application to be compromised in places of the application and at a development stage where performance is not an issue. On a first stage of development should focus on a clear design and a manageable code. At this phase constant refactoring may change the design of the application till the final product is reached. For this iterative development a hard to read code written to performance optimized is a major blocker. At a late stage of the development performance optimizations can be specifically implemented where a bottle neck could be detected empirically (i.e. by profiling the application).

It is always much easier to transform code designed for manageability and complexity reduction into code optimized for performance than the other way round. This is especially true for SPARQL client vs. native triple store API: because the native triple store API offers features which are not defined by the RDF specification (such as persistently referable BNodes) in the general case a refactoring of the data access functions is only possible from SPARQL to native API but not in the other direction. In practice if the application should be refactored later to use SPARQL this would likely to imply quite a pervasive refactoring, while a change in the other direction could be achieved very locally at identified performance critical bits of code.

2.3.3 Compatibility across backends

The Sesame API allows connecting to many third party triple stores including Openlink Virtuoso (using a JDBC connection), Ontotext Owlim. Via adapters[4] it should also be possible to access Jena datastores

[3] http://www.dehora.net/journal/2003/06/foundations_for_component_and_service_models.html

[4] As the adapters do not seem to be actively maintained it is questionable how well this work with current versions: https://github.com/afs/JenaSesame (last update 2 years ago), http://sjadapter.sourceforge.net/ (last update 10 years ago), http://sourceforge.net/projects/jenasesamemodel/ (last update 7 years ago)

such as TDB and SDB using the Sesame API. However even though many triple stores can be accessed via the sesame API this is not true for all triple stores. Some triple stores like Jena TDB and 4Store can only be connected via third party modules of questionable liveliness. Other solutions have no Sesame binding to date this ranges from supercomputing devices such as YarcData Urika from Cray[5] to javascript stores running on node.js like rdfstore-js[6].

2.3.4 Adherence to standards

The main question is not about counting today's solution where it might indeed be true that a high percentage offer some kind of Sesame API binding but about choosing a sustainable solution. To dare a comparison: it might have well been that in 1996 many more computers were able to show Word Document than HTML documents, yet it was probably advisable for most start-ups to focus on providing data in HTML rather than Word. Sparql is one of the core standards of the semantic / linked data web we build our applications for. Our product depend on the long term success of these standards and it makes thus sense to stick to them rather than choosing a solution provided by a vendor and tight to a particular programming language.

2.3.5 Debuggability

By using SPARQL it is well defined what goes over the network and the messages sent are identical for any storage. This allows to monitor the network traffic and understand what is going and what is going wrong. Plain text based protocols (especially HTTP based) have gained a lot of traction because they allow human inspection which is important for debugging and during development. When using Sesame API it is not defined how the vendor specific Java library implementation the API communicates with the backend, for Virtuoso this uses a JDBC driver and a property protocol, other may use the readable but not standardized Sesame Protocol, yet others may use proprietary binary protocols. As with HTTP and HTML with SPARQL it is very easy for Network administrators as well as for developers to see what is happening, this easies debugging as well as network optimizations.

2.3.6 Correct Blank Node handling

A more technical aspects is that the Sesame API allows to address Blank Nodes (BNodes) persistently using an identifier. This means the application might reconnect to the RDF store and for example add additional properties to a previously added BNode. While this might seem like a useful feature it effectively blurs the distinction between named nodes and blank nodes, the only remaining difference is the merely syntactic aspect that the former are identified by IRIs and the latter by blank node IDs. This doesn't forces developer to use Named Node where local referenceability is needed which leads to bad design.

On the other hand the backend has to keep redundant information as it cannot dispose of redundant bnodes.

Take for example the graphs:

G

```
ex:a  ex:b  _:b1.
ex:p  ex:b  _:b1.
ex:a  ex:b  _:b2
```

[5] http://www.cray.com/Products/BigData/uRiKA.aspx

[6] https://github.com/antoniogarrote/rdfstore-js

and G'
ex:a ex:b _:b3.
ex:p ex:b _:b3.

G and G' are equivalent, however if the backend is accessed via a Sesame API it is not free to remove any triple. A store accessed with SPARQL by contrast might for example ignore INSERT operations adding redundant operations or use other means to keep the graph leaner.

2.3.7 Specification in DOW

While the DOW mentions SPARQL in several places so it is definitively part of the set of standards used by the platform, it doesn't specially mandate SPARQL for the communication with the backend. However it envisages a "federated ecosystem based on a set of common standards that guarantee interoperability but within which variation can take place", as opposed to Sesame SPARQL is not tied to the Java platform so it allows more variation while guaranteeing the interoperability with a smaller set of standards.

The Fusepool P3 projects wants to build "Data publishing tools that are […] very simple to deploy and maintain in organizational IT environments". Deployment in existing IT infrastructure is clearly easier if the software interacts with other software (such as a triple store) using well defined standards. Even if it might be possible that the enterprise database the organization uses (e.g. Oracle Database) provides Sesame binding deployment to a new DBMS would still require at least exchanging drivers within the software. By using a standardized SPARQL communication over the network no driver exchange is necessary. An exchange of driver is particularly tedious if the software doesn't use a module system such as OSGi.