Graphity – A Generic Linked Data Platform

Position paper for Linked Enterprise Data Patterns W3C workshop December 2011

Martynas Jusevičius martynas@graphity.org
Aleksandras Smirnovas aleksandras@graphity.org
Julius Šėporaitis julius@graphity.org

Before building the danish entertainment site HeltNormalt we were facing architectural and design decisions. As a successor of the daily Wulffmorgenthaler comic strip, it had to include a dozen of content types, containing (one or more) images, plain text, XHTML, or a combination of these.

We chose Linked Data as the primary design principle, and this choice brought us a number of great advantages. We would like to share the knowledge we gathered during our development process and show that Linked Data can both be used to build mainstream websites, as well as serve as a platform for the next generation of innovative data-driven Web applications.

Background

Having experience with both PHP and Java Web applications, Web frameworks like Symfony[1] and Zend[2], we didn't feel like they were adequate or flexible enough for a modern Web application. Based on those experiences, our main concerns were following:

Code is not portable enough, developer lock-in is too deep
Frameworks are platform-specific (i.e, run either on PHP or Java but not both). Moreover, they introduce framework-specific conventions (e.g., template languages like Smarty[3] or YAML[4] markup).

Frameworks hide HTTP and build on too many abstractions
Java EE, .NET and similar enterprise frameworks wrap the inner workings of HTTP under thick layers of abstractions so that the end developers might not even know they are working on a Web application. However, we don't think it is the right approach – abstractions are leaky, and it looks like we are trying to build high-level languages for the Web without ever having found out what its assembler is.

From experiences with ORMs like Propel[5] and databases like MySQL, our concerns were following:

Unnecessary code and bugs are introduced by data model conversions
Common example is transforming relational data to XML or to HTML (which are based on a tree model)

Schema is hard to manage, reuse, and a limitation for semi-structured data
Mandatory relational schema would not allow for ad-hoc modeling of the semi-structured content we had at hand. Metadata properties had to be common for all data while content properties were content type specific.

Moreover, reuse and integration of relational schemas is very poor, and updating schemas online without disrupting a working website is quite a challenge.

Besides that, our team also had experience in RESTful design (having built applications on an in-

1 http://www.symfony-project.org
2 http://framework.zend.com
3 http://www.smarty.net
4 http://www.yaml.org
5 http://www.propelorm.org

house PHP framework as well as JAX-RS[6]), the Linked Data/Semantic Web stack (RDF, SPARQL, OWL), and XML/XSLT, which we wanted to incorporate as well.

Requirements

The decision was made to design the site from scratch, and build a platform that could be reused with any data while running HeltNormalt as the first instance. We set the following requirements:

- Embrace HTTP and REST instead of hiding them
- Embrace Model-View-Controller (MVC) and object-oriented design
- Use a single unified data model throughout the whole design, if possible
  - it should allow flexible semistructured data
- The platform should be generic
  - it should work on any data source no matter its domain, as long it is accessible via the unified data model
  - default behavior should be provided
  - it should be possible to override the default behavior
- Do not reinvent the bike
  - reuse and combine established standards
  - reuse and combine popular ontologies
  - reuse external data
  - do not create new wrappers, formats, tools, or conventions unless absolutely necessary
- Make the design as flexible and portable as possible
  - loose coupling
  - Don't repeat yourself (DRY)

Design

In the light of the experiences and requirements above, the design outcome was a generic Linked Data framework with the following components:

Model

Since we did not use an ORM or stored procedures etc., our model is simply the datastore, consisting of RDF database (TDB[7]) and SPARQL Update endpoint (Fuseki[8]). The data model *is* the Model.

Data provenance was solved the following way: every piece of RDF submitted to the system (including ontology etc.) gets its own named graph. Transaction metadata is described using the Provenance Vocabulary[9] and the Open Provenance Model[10].

6 http://jcp.org/en/jsr/detail?id=311
7 http://openjena.org/TDB/
8 http://openjena.org/wiki/Fuseki
9 http://trdf.sourceforge.net/provenance/ns.html
10 http://open-biomed.sourceforge.net/opmv/ns.html

View

It works by sending SPARQL `CONSTRUCT` or `DESCRIBE` queries to ask for model state, and transforms the resulting RDF/XML into (X)HTML using XSLT.

It is well-known that RDF/XML is difficult to transform, however we established XSLT techniques that make this task easier. There are approaches to use a simplified subset RDF/XML or transform it into a more accessible XML dialect, however we think only raw RDF/XML gives full flexibility, combining both RDF and XML worlds.

Whole (X)HTML pages are being built from a single RDF/XML result. A single main XSLT stylesheet performs the transformation, it imports necessary content type-specific stylesheets.

View is the MVC equivalent of resource representation found in REST. It is merely an extension of HTTP response message, adding the content body to it.

Controller

Matching request URI against known path patterns and also looking up if it exists in the Model, Controller delegates processing to specific Resource classes. This method is actually quite similar to XSLT template matching.
Resources can in turn change Model state by sending SPARQL Update queries, and return appropriate Views.

Written in PHP, Controller is the only platform-dependent component. It borrows JAX-RS annotations such as `@Path` and `@Consumes` and in that way is probably the first JAX-RS implementation in PHP.

Access control is built-in since Resources include metadata based on the Basic Access Control ontology[11]. Cache control is automated using SIOC[12] post/container metadata.

The framework is merely a thin wrapper combining existing tools with minimal stack trace, managing the flow of incoming RDF and translating responses into (X)HTML. The goal is to keep it minimal, stable, and bug-free.

Second part of the platform are the plug-in packages (bundles of Resource classes, libraries, SPARQL queries, and XSLT stylesheets). HeltNormalt is the first of them, and although its code will not be public, the mechanism allows for any arbitrary packages to be run on the framework.

Implementation

Our design choices were the prerequisites for a generic platform, but the actual implementation and continuous refactoring followed some distinct patterns, which closely resemble the principles of Inversion of Control[13] and Dependency Injection:

- Look for patterns in class and method signatures and templates
- Abstract these classes/methods into a single pattern
    - externalize type information, if necessary (e.g. as a parameter)
    - move the code to the superclass/master file
    - move descriptions and relationships to RDF

11 http://www.w3.org/ns/auth/acl
12 http://sioc-project.org/
13 http://martinfowler.com/bliki/InversionOfControl.html

- Only allow single point of entry
  - provide plugin mechanisms to override default behavior
  - Hollywood Principle: "Don't call us, we'll call you"

Results

Results confirmed and exceeded our expectations. There still is room for further generic refactoring, but we met our requirements:

- Model is generic in the form of RDF; object-oriented interfaces compatible with Jena. Dublin Core, FOAF, SIOC, AtomOwl[14], DBpedia, Zodiac[15], Linked Data API[16], Basic Access Control, OPVM ontologies reused.
- View is generic in the form of XSLT stylesheets that can be overridden
- Controller is compatible with JAX-RS and separate from 3rd party code

In general, moving application logic from source code to declarative data has following advantages:

- less source code means less bugs, less maintenance and lower learning curve
- data in the system can be incorrect but it cannot contain bugs or be invalid in respect to it's data model (given the datastore and/or validation works correctly)
- application logic can be managed using same tools (or the application itself) that are used to manage domain model data
- data is platform-agnostic

The codebase shrank by an order of magnitude, while maintaining similar lever of features and being more flexible.

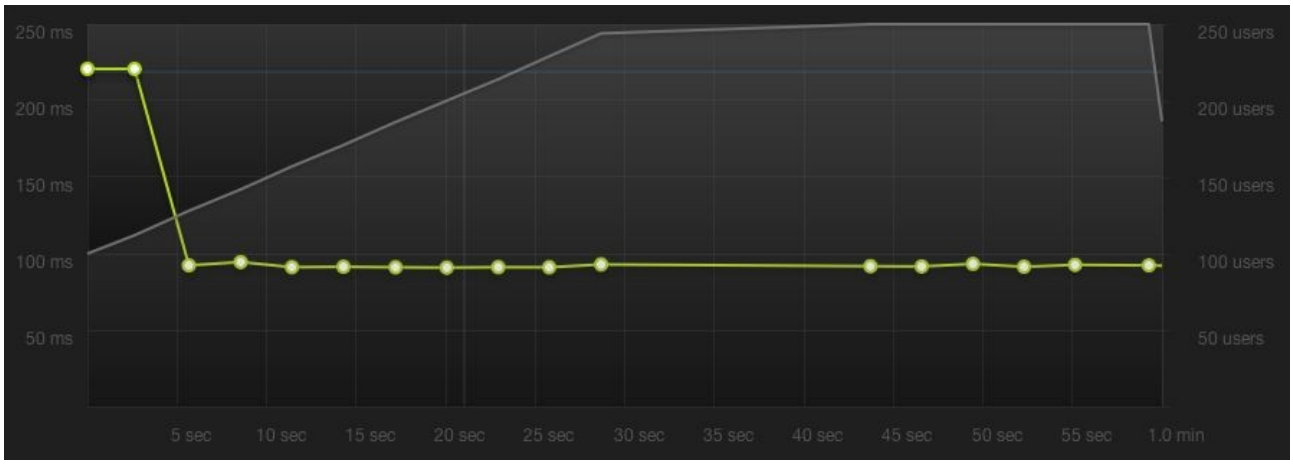
Component	Language	# of files	# of lines
HeltNormalt			
Model	SPARQL	9	614
View	XSLT	7	1898
Controller	PHP	8	652
Wulffmorgenthaler			
Model	PHP	93	19125
View	PHP	67	2528
Controller	PHP	51	7625

Comparison with the previous codebase
(HeltNormalt metrics reflect only one corresponding content type)

The platform can also achieve exceptional performance (with Varnish[17] proxy cache in front,

14 http://bblfish.net/work/atom-owl/2006-06-06/AtomOwl.html
15 http://data.totl.net/zodiac/
16 http://code.google.com/p/linked-data-api/wiki/API_Vocabulary
17 https://www.varnish-cache.org

Memcache[18] as RDF/XML cache, and XSL Cache[19]). Naturally, it depends on the complexity of SPARQL queries and XSLT stylesheets.



Green line - response times in milliseconds; gray line - number of user requests per second. Measured using blitz.io

Conclusions & Future Work

We would go as far as to say that RESTful controller combined with RDF as data model, SPARQL Update as the query language, and XSLT as the template language are the ultimate components for the next generation (semantic) Web applications. We think the World Wide Web Consortium could define a set of best practices in this area, and engage in a dialogue with prominent communities to promote established standards such as RDF and XSLT.

While Graphity might not be the ultimate implementation, it is a step in the right direction. The core framework is already opensourced on graphity.org under Apache license, where we also hope to build a developer community around it. We were inspired by projects such as Linked Data API[20], Callimachus[21], and Information Workbench[22].

In the near future, we would like to port the Controller (while reusing the rest of the components) to different platforms. Java (JAX-RS) port is easiest and therefore almost ready, .NET and possibly Python come next.

Also of high priority is the development of a Web application container similar to Tomcat, which would be a novelty in the PHP world. Move to cloud services would open multiple possibilities – first for data storage on services like Dydra[23], and possibly the whole Graphity could run on Amazon Web Services or a similar service.

The possibilities for pluggable packages are virtually endless. We have plans to build a Linked Data browser incorporating RDFizers[24] for well-known data formats and Web APIs together with generic presentation and editing interfaces. We also have plans for packages in areas such as blogs, social networks, analytics, maps, media etc. We expect contributions from 3rd parties that would like to integrate and promote their own ontologies or datasources.

18 http://php.net/manual/en/book.memcached.php
19 http://code.nytimes.com/projects/xslcache
20 http://code.google.com/p/linked-data-api/
21 http://callimachusproject.org
22 http://www.fluidops.com/information-workbench/
23 http://dydra.com
24 http://simile.mit.edu/wiki/RDFizers