HTTP Database Connector (HDBC): RESTful Access to Relational Databases

Alexandros Marinos
Department of Computing
University of Surrey
a.marinos@surrey.ac.uk

Erik Wilde
School of Information
UC Berkeley
dret@berkeley.edu

Jiannan Lu
Department of Computing
University of Surrey
jiann.lu@yahoo.com

ABSTRACT

Relational databases hold a vast quantity of information and making them accessible to the web is an big challenge. There is a need to make these databases accessible with as little difficulty as possible, opening them up to the power and serendipity of the Web. Our work presents a series of patterns that bridge the relational database model with the architecture of the Web along with an implementation of some of them. The aim is for relational databases to be made accessible with no intermediate steps and no extra metadata required. This approach can vastly increase the data available on the web, therefore making the Web itself all the more powerful, while enabling its users to seamlessly perform tasks that previously required bridging multiple domains and paradigms or were not possible.

Categories and Subject Descriptors

H.3.5 [**Information Storage and Retrieval**]: Online Information Services—*Web-based services, Data sharing*

General Terms

Design, Documentation, Languages

Keywords

REST, Web Architecture, Relational Databases

1. INTRODUCTION

Relational databases hold a vast quantity of information and making them accessible to the web is an open issue. The need to make it accessible with as little difficulty as possible has led us to examine the potential of making a generic bridge between the architecture of the Web as expressed by *Representational State Transfer (REST)* [1], and the metamodel of relational databases. REST is a set of constraints for architecting distributed systems. These constraints include manipulation of resources through representations, a uniform interface, statelessness, and hypermedia as the engine of application state. In the implementation of REST that is the *World Wide Web*, these constraints are applied through its fundamental specifications, HTTP, URI and representation formats such as HTML, XML, and Atom among others. The *HTTP Database Connector (HDBC)* aims to make databases available through the Web, along with a large subset of the querying capabilities of SQL, aimed for both machine and human users, in a manner that is consistent with REST. It should achieve this without putting the data owners through unnecessary steps.

WWW 2010, April 26–30, 2010, Raleigh, North Carolina, USA.
ACM 978-1-60558-799-8/10/04.

2. RELATED WORK

Relational databases have standard connectors for most programming languages, such as Java's *JDBC*, which make them accessible to programmers coding in that language. However, no such "connector" exists for making all the power of relational databases available on the Web. Of course, software packages exist to query and manage relational databases on the Web (e.g., *phpMyAdmin*), but they are geared towards human users and even there, their architecture does not observe the constraints of REST well. Methods to make models in some Web frameworks available as RESTful Web services (e.g., *Djata*) require an additional level of modeling beyond the relational database, a potential barrier to entry for the use cases considered. Some projects exist to make databases directly available on the Web (e.g., *SQLREST* and *REST-SQL*), but the querying capabilities that they provide are limited to listing the contents of a given table. Finally, Google's *GData* and Microsoft's *WCF Data Services* offer querying models for data on the Web, but neither of them are tailored to the relational model and therefore lack essential SQL features such as joins.

3. CONNECTOR DESIGN

Making Resources Available. The first step for any RESTful system is to decide what resources will be made available. This enables usages such as users exchanging query URIs, a common use case on the Internet, which however is rather unusual when it comes to database queries. In the case of the relational model, the obvious candidates are tables (relations) and the records (tuples) which make them up. These can be cast as resources, with the table resources providing links to the record resources. As with all resources, these should be granted their own URIs. We can also model query results as resources, with the respective URI encoding the query. Such queries can contain filters, ordering, paging, and joins. Finally, the database schema itself can be modeled as a set of resources, therefore enabling inspection and modification of the schema.

Uniform Interface. The uniform interface constraint requires that a resource is accessible by a limited set of operations with well-defined semantics. HTTP offers the operations `GET`, `PUT`, `POST`, and `DELETE`, among others. With

operations beyond `GET`, we can achieve write access to the database. For instance, by using `PUT` on a table URI with a new record as a request body, we are requesting that the new record is added to the table.

Representations. In REST, resources can have multiple representations. In HTTP, the appropriate representation is theoretically determined through content negotiation. Alternatively, some services rely on URI suffixes instead for reasons of practicality. Multiple representations for a resource allow human-to-machine and machine-to-machine use cases to co-exist with the same URI structure, with only the returned representation differing. Humans can consume HTML, while machines may prefer XML, JSON, or Atom. The authors have previously discussed the benefits and implications of serializing query results as feeds [2]. In the case of serializing query results as Atom, users can subscribe to queries through a feed reader and therefore be notified of updates to their queries, a capability that can be backed by *streaming databases*. In addition, the publishing capabilities of AtomPub can be used as a way to relay updates back to the database.

Statelessness. The aforementioned use case of sending query URIs through an email or IM message hinges on another property of RESTful systems, namely statelessness. It requires that all that is needed to interpret a request is contained within that request, and not in intangible server-side state. Thus, a URI that is bookmarked can be used at a different time or even from a different machine to request the same resource. This is relevant to the issue of authentication, where HTTP authentication and cookies are two popular approaches. Regardless of the specifics of the authentication mechanism, by linking the internal user access mechanism of a database to a HTTP-compatible authentication mechanism, there can be granular control of access rights per user without additional information stored outside the database.

4. HYPERMEDIA CONNECTIONS

Perhaps the least understood constraint is that of *Hypermedia as the Engine of Application State*. The intent is that RESTful APIs should be discoverable and not dependent on external information for construction of the URIs. This decouples the clients from the URI-space of a server. Clients follow links, which can also be discovered through HTML forms, and discover resources one at a time. The next potential states at each step are determined by the server. Thus, if the server decides to change their URI-space, a client can simply rediscover the resources. Thus, while we have earlier mentioned encoding the queries as URIs, the precise manner in which this happens should not concern the end-users as they should be discovering their resources through link traversal and not URI construction. This implies that in a complete system, queries can also be constructed by following links, progressively narrowing a coarse initial query with additional elements, through propagating links.

5. IMPLEMENTATION

The current version of the implementation[1] of this software is focused on machine-to-machine interactions, in particular based on the atom publishing protocol. A user can therefore browse an Atom feed serialization of a table or query result and therefore subscribe to the feed through their feed reader of choice and be informed of any updates to the resource. An example query resource with a simple filter URI would be formatted in URI template syntax as follows: `http://example.org/{database}/{table_name}({id})`. A more complex query involving a join such as the template `...{database}/{table1_name}/{table2_name}` would yield an inner join of the two tables, assuming that they have a foreign key relation declared in the database schema. The query engine infers from this how the tables are to be joined. In case a more complex join is required, more specific information can be embedded in the URI. Of course, features such as filtering, joining, sorting, paging, and simple SQL functions such as `count()` can be combined to yield complex queries. Beyond `GET`, HDBC also offers the ability to relay updates to the database by using `PUT`, `POST`, and `DELETE`.

The implementation has been adapted to both Apache Derby (also known as JavaDB) and MySQL which has brought to our attention an interesting inconsistency between the two different databases: While MySQL is case insensitive in the strings it receives in its queries, JavaDB is not. There is no simple way to remedy this incompatibility at the URI level, so the behavior at the URI level corresponds to the database engine that is being used in the back end.

[1]The source code can be found at `http://code.google.com/p/rest-hdbc/`

6. CONCLUSIONS

In this paper we present first steps towards a RESTful way of interacting with a query-oriented back-end. While the approach we present is specifically designed to work with the relational structures of the SQL model and language, the same patterns could be applied to create RESTful access methods for other query-oriented back-ends as well. SPARQL would be a promising candidate, the current version of SPARQL shares SQL's approach of a "service endpoint" through which all calls and responses are channeled.

Starting from the design patterns presented here, it is our intention to extend these to cover more advanced scenarios such as RESTful transactions and the composition of various of these services (possibly across different back-end data models, thus merging the worlds of SQL and RDF data on the REST level). In essence, such an approach would then allow the publication of "linked data" regardless of the back-end technology, and instead of requiring to cast all data into a single metamodel (which is what the narrow definition of "linked data" as RDF-only requires), this view of linked data would focus on REST as the main architectural principle of the Web.

7. REFERENCES

[1] Roy Thomas Fielding and Richard N. Taylor. Principled Design of the Modern Web Architecture. *ACM Transactions on Internet Technology*, 2(2):115–150, May 2002.

[2] Erik Wilde and Alexandros Marinos. Feed Querying as a Proxy for Querying the Web. In *Eighth International Conference on Flexible Query Answering Systems*, volume 5822 of *Lecture Notes in Artificial Intelligence*, pages 663–674, Roskilde, Denmark, October 2009. Springer-Verlag.

This work was partly supported by the (INFSO-IST) OPAALS Project, funded by the European Commission (Project number:034824)