A REST SERVICE FOR IRMIS3

G. Carcassi, BNL, Upton, NY, U.S.A.

Abstract

We will describe the architecture of IRMIS3, the role of the data service and how it allows easier upgrade paths by concentrating all the logic in one place. We will describe how the service is accessed, both for read and writes, and some details about its current implementation.

INTRODUCTION TO IRMIS

IRMIS [1] is a collaborative effort between several EPICS sites to build a common Relational DataBase schema and a set of tools to keep of the difference parts of an accelerator, including components and their relationships, the process variables of the control systems and the elements of a simulation lattice.

IRMIS3 [2] is an effort started at BNL to address some of the shortcomings of the previous version. In particular this paper will show how the choice of a Service Oriented Architecture [3] address concerns about RDB schema evolution, protecting the consistency of the data and interoperability with multiple languages and technologies.

IRMIS3 ARCHITECTURE

Prior to IRMIS3, all the tools would connect directly to the database. This poses a problem in allowing the schema to evolve: if a change is implemented, all the clients have to be modified and deployed at the same time that the new schema is deployed. Even if a common library is used for all clients, one has to know where all such clients have been installed, including scripts that have been written and that are not routinely used. Failure of a synchronized update can be minor (some clients do not work) to major (client saving with the old format, corrupting the database). This is even more problematic when it is the business rules that have changed: one would have to be sure that all the clients are behaving correctly.

The consequence is that the upgrade is a stop-the-world process, which is rarely possible in a production environment. The further consequence is that changes pile up for the few opportunities available, increasing the actual risk of failure of the upgrade process.

Another problem of connecting directly to the database is that the business logic needs to be coded at least once for each platform or language supported. This creates additional support costs, and combines to the general problem of evolving the software.

In the IRMIS3 architecture, all the accesses to the database go through the data service first. The service is a REST style web service [4], is responsible for executing all the business logic and data safety. The service defines its own protocol between itself and the clients through an XML schema and a set of URLs that the clients can access.

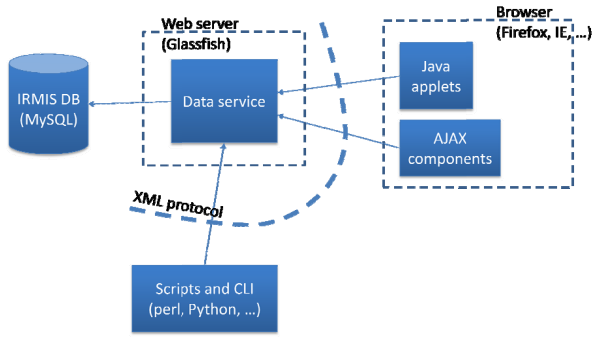

Figure 1: IRMIS3 architecture.

In this case, the service is the only component that needs to be aware of the schema change. If the change is such that the external protocol can be maintained, which is typically the case of performance driver modifications, then all the clients will never be impacted by the change. If the change is such that the protocol needs to be substantially modified, once can operate for an intermediate period with the service speaking both the old and the new protocol with the same database backend. This allows gradually moving the clients to the new protocol, while keeping the old infrastructure in place. When and if the previous protocol is no longer needed, it can be dropped. This practice is common place on most commercial websites, and allows deploying a constant stream of small improvement while retaining control and uptime of the system.

The other advantage is that the clients need only know about the XML protocol, and not the business logic. The client can be “dumb”, so to speak, because its mistakes will be caught and rejected by the service.

We want to stress out that the choice of REST, instead of SOAP or XML-RPC, has less to do with performance and ease of use than the simple fact that the service just provides access to data. SOAP and XML-RPC are the appropriate choice for remote procedure calls, so they would not be appropriate in this context regardless of any other consideration.

RETRIEVING DATA

Data can be retrieved with any standard HTTP client. The service provides a set of URLs one can use, each address representing a different query. Queries can be parameterized through a set of HTTP parameters. The service itself provides documentation of which queries and parameters are available.

The server will return an XML file that represents the data requested. The protocol is designed to minimize possible roundtrips, so it returns as much information as possible, typically de-normalizing the data returned. This

also allows using standard XML tools, such as XPath or XSLT, to search and analyze.

Standard REST conventions are only followed when appropriate. Most notably, each data elements is identified by and id which correspond to the database primary key instead of a full URL. Many data elements do not even have a specific URL associated with them. For example, there is no URL for each manufacturer, only a URL for the (possibly filtered) full list.

WRITING DATA

For writing data, we depart slightly from what is typically done in a REST service since you cannot simply use PUT/POST/DELETE on IRMIS resource URLs to make modifications. It was simply impractical to perform each operation individually, as a typical session of editing might change hundreds if not thousand of data elements.

The biggest problem, though, is that we want to keep those changes as a single operation, so that the client has a much simpler time managing updates as it does not have to clean up if one intermediate operation fails.

The way that data is modified in IRMIS3 is to prepare an XML that represents the transaction to be executed on the server. Each transaction is broken down into xml elements that represent command, such as create a component, connect a cable or delete a manufacturer. All these elements can be mixed and matched in any order as they are independent of each other. They will be executed in the same database transaction on the server, as to guarantee atomic execution.

This implicitly allows for disconnected operation: a handheld client can collect a list of changes, and then send the XML the first time it connects online. This can be done without any extra work, simply by saving the transaction to a temporary file instead of sending it directly over the network.

We also strive to keep the transaction idempotent: if the same transaction is sent twice, it will either fail, or it will have the same result. This greatly simplifies usage if network problems are encountered: if the previous request dropped out, one can safely retransmit.

The only caveat is that there is no read lock: one cannot lock the data on one read to make sure that when he will write no modifications happened in between. The cost of such feature is deemed too large for the benefits that it will introduce. The server will, in any case, save all the XML transactions it received, so that the rare collision can be at least investigated properly.

IMPLEMENTATION

The service is implemented in Java [5], under Java SE 6 and Java EE 5. It is deployed in Glassfish 2.1 [6], the open source application server sponsored by Sun.

The server itself is a Servlet which accepts all requests and dispatches them to the appropriate class for handling. Each query and each transaction element is implemented as a separate class.

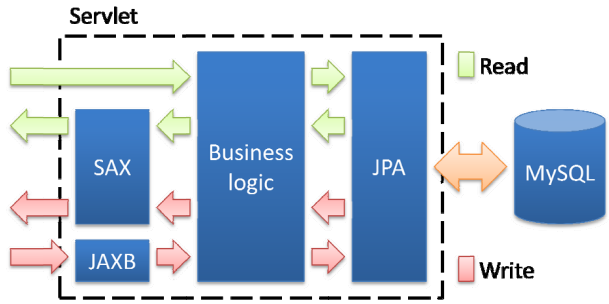

Figure 2: Service implementation.

The database access is provided through the Java Persistance API [7], which is a standard spec for objects brokers (such as Hibernate or EclipseLink). We currently use TopLink essentials as JPA provider simply because it's the one included in Glassfish.

We use JPA simply as a way to write queries more easily taking advantage of JPQL, an SQL dialect that allows to query directly on the objects and chain through relationships through the '.' notation. It also provides an abstraction on top of the database, so that we could switch from one database vendor to another with little effort.

We actually started out using JDBC with direct SQL manipulation, but the cost of maintaining the code was starting to get out of control: to de-normalize the data we have to go through many relationships and many joins, with called for queries that could be as long as 600 characters. With that length, SQL starts to become very difficult to read and understand. Use of JPQL cut the length by more than half, and made them much more readable. The cost associate to moving to JPA was about 6%, so it was not deemed high compared to the gains.

JPA also greatly simplifies writing the non-trivial business rules that IRMIS requires. And it also makes it easier to understand what needs to be changed after a schema change: the classes are regenerated from the schema, and we get a compile time error whenever something needs to be fixed. In JDBC, instead, all references would be through Strings, so we would have to use search and replace techniques which tend to be more brittle. We make sure, though, that no logic is attached to JPA objects, so that we can throw them out and regenerate them. It also makes the code less relying on JPA in case we changed our mind in the future.

To parse the incoming transaction, we use JAXB [8] which is a standard library to convert XML into Java objects. This simplifies the coding of the business logic in the same way that using JPA objects does. JAXB objects are automatically generated from the XML schema, therefore they also do not contain any business logic or rules.

To write the data out we do not use JAXB (or DOM): this would force us to create the whole document before we started to write the first byte, which would increase the delay suffered by the client. We use SAX to stream the data as it becomes available.

PERFORMANCE

We typically do our performance evaluation by using an old copy of the APS dataset which was converted to use IRMIS3. On my laptop, a Dell Precision M4300 - Intel Core Duo T7500 - 2.20 GHz, the longest query that streams all of the 31338 components and their 63911 relationships takes about 5 seconds. Most of the other queries take less than 100 ms.

Though 5 seconds is not trivial, is actually a good number as it is still acceptable at start-up of a GUI or of a script. What one gains is that no further roundtrip is needed to go through the entire dataset, so searching, filtering or providing visual cues is trivial. The other queries are well below 500 ms, which is the typical threshold of patience for a GUI user.

The main reason we are able to stream that much database rows and relationships is that the number of queries are a constant in respect to the amount of data retrieved. For example, in the case of components and their relationship, we have three queries: one for the component themselves, one for the parent relationships and on for the child relationships. The queries are returned in the same component order, so that we can read the first component from the first queries, its parents from the second and its children from the third. The entire output is created by scrolling through the three queries at the same time.

It also important to note that only half the time is spent by the database query. This means, for example, that after a pause of 2.5 seconds XML starts coming in, and the client can start parsing and interpreting the data. So the total latency seen by the client is only slightly more than 5 seconds since a good part of the parsing happens while the data arrives. This is why we used SAX (instead of JAXB or DOM) to generate the XML.

No further optimization was done as these expedients gave us a good enough performance. There are other areas that could be explored, such as hardware/os tuning, streaming XML directly without using SAX, profiling to look for unexpected bottlenecks and caching the produced XML for the next request. These can be used if in the future we need to further improve performance.

CONCLUSION

Between October 2008 and October 2009 we had 12 releases of IRMIS, 6 of which included db schema updates. We always had a production instance running, which is gathering data as it becomes available. This shows that it is possible to evolve the system with little impact on the client.

The data service layer also constitutes a good boundary on which to integrate. In particular, NSCL at Michigan State University contributed a php API for the data service, which adds to the Java API already provided. They were also able to build a simple php inventory management system, which included a handheld device working in disconnected mode: it can be used to scan barcodes; it prepares the XML transaction and then sends it when it is back online.

We encourage the reader to consider using a service layer in front of other databases or systems they may have already deployed as a way to decouple the components and allow for better separation of the business rules and simplify upgrades.

AKNOWLEDGEMENTS

This work would not have been possible without the help of Peter Beard, Eric Berryman, Bob Dalesio, Don Dohan, Debby Quock, and Doug Sheffer.

REFERENCES

[1] IRMIS; http://www.aps.anl.gov/epics/irmis/index.php.
[2] IRMIS3; ttph//irmis.sourceforge.net.
[3] An introduction to Service Oriented Architecture http://en.wikipedia.org/wiki/Service-oriented_architecture.
[4] An introduction to Representational State Transfer (REST) http://en.wikipedia.org/wiki/Representational_State_Transfer.
[5] http://java.sun.com.
[6] https://glassfish.dev.java.net/.
[7] JPA; http://java.sun.com/javaee/overview/faq/persistence.jsp.
[8] https://jaxb.dev.java.net/.