ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-TUPHA038

A GENERIC REST API SERVICE FOR CONTROL DATABASES*

Wenge Fu†, Ted D'Ottavio, Seth Nemesure
Brookhaven National Laboratory, Upton, NY 11793, USA

Abstract

Accessing database resources from accelerator controls servers or applications with JDBC/ODBC and other dedicated programming interfaces have been common for many years. However, availability and performance limitations of these technologies were obvious as rich web and mobile communication technologies became more mainstream. The HTTP Representational State Transfer (REST) services have become a more reliable and common way for easy accessibility for most types of data resources include databases. Commercial products to quickly setup database REST services have become available in recent years, each with their own pros and cons. This paper presents a simple way for setting up a generic HTTP REST database service with technology that combines the advantages of application servers (such as Glassfish/Payara), JDBC drivers, and REST API technology to make major RDBMS systems easy to access and handle data in a secure way. This allows database clients to retrieve data (user data or meta data) in standard formats such as XML or JSON.

INTRODUCTION

As a common resource of data, the usage of databases are essential for accelerator control systems. The way to access databases from all parts of the controls system, locally or remotely, can greatly affect the overall performance of the accelerator controls system. Traditional ways of accessing databases with JDBC or ODBC from different programming languages works fine in most cases. However, the rising and widespread use of mobile and web applications requires more sophisticated ways of accessing database resources for network-based programs. The Representational State Transfer (REST) API paves a universal way for this purpose. While several commercial products (such as RESTIFYDB[1], Drupal[2], Firebase[3] etc.) are available on the market, most of these REST API building products are based on JPA or Hibernate technology, and each product has its own advantages with rich functionality. However, there are also some disadvantages, requiring a constant and significant amount of maintenance work for dynamically changing control database systems such as the ones we are using.

In our controls system, we have several MySQL database servers and SAP Sybase ASE database servers, encompassing several hundred databases. Some databases are for storing the control system configuration data and others are for use to store real-time controls data. In addition, we have many controls application developers with different programming language backgrounds (C. Java, Matlab etc.), that work on different projects on different OS platforms, some locally and others remotely. This requires an easy and generic way to universally expose different kinds of database resource data to all kinds of clients.

Based on our database resource types and user environments, we developed a simple REST API service for our users to access controls databases. In the API design, we try to avoid the shortcomings of JPA/Hibernate technology on frequently changing database structures (such as dynamically mapping database objects into Java objects), while keeping the full syntax, flexibility and power of SQL language. This ensures that users and program developers can always get whatever database resource data they want, no matter how a database's structure changes overtime.

The API presents the resource data in either XML or JSON format along with the meta data, so users get everything they needed in a single API call. The data security issue of the REST APIs are also taken into the consideration in the API design.

APPLICATION SERVER SETUP

REST API services are normally delivered by an application server system. Our server system setup is shown in Figure 1 below. Here are some details:

- The main database servers and application servers are located inside our network firewall. The firewall is primarily responsible for basic system security and only allows authenticated users to access the REST API service.
- We make use of several MySQL (V5.1.73) and SAP Sybase ASE (V15.7) back end database servers. Since the core of this REST API is using the standard JDBC to make database server connections, and doing all CRUD database operations through JDBC underneath, this API can be extended to any JDBC supported database servers.
- The application server is running on a Red Hat Linux V6.5 OS. The application software is a version of the open source Payara application server form the Payara Foundation. The Payara server is derived from GlassFish, and provides developer support. [4]
- An internal reverse proxy server (NGINX) provides an HTTP gateway for all internal users behind a local firewall to access the REST API services available on the application server.
- An external reverse proxy (NGINX) server provides a secure HTTP gateway to allow authenticated users to access the REST API service inside the firewall.

* Work supported by Brookhaven Science Associates, LLC under Contract No. DE-SC0012704 with the U.S. Department of Energy.
† Email: fu@bnl.gov

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-TUPHA038

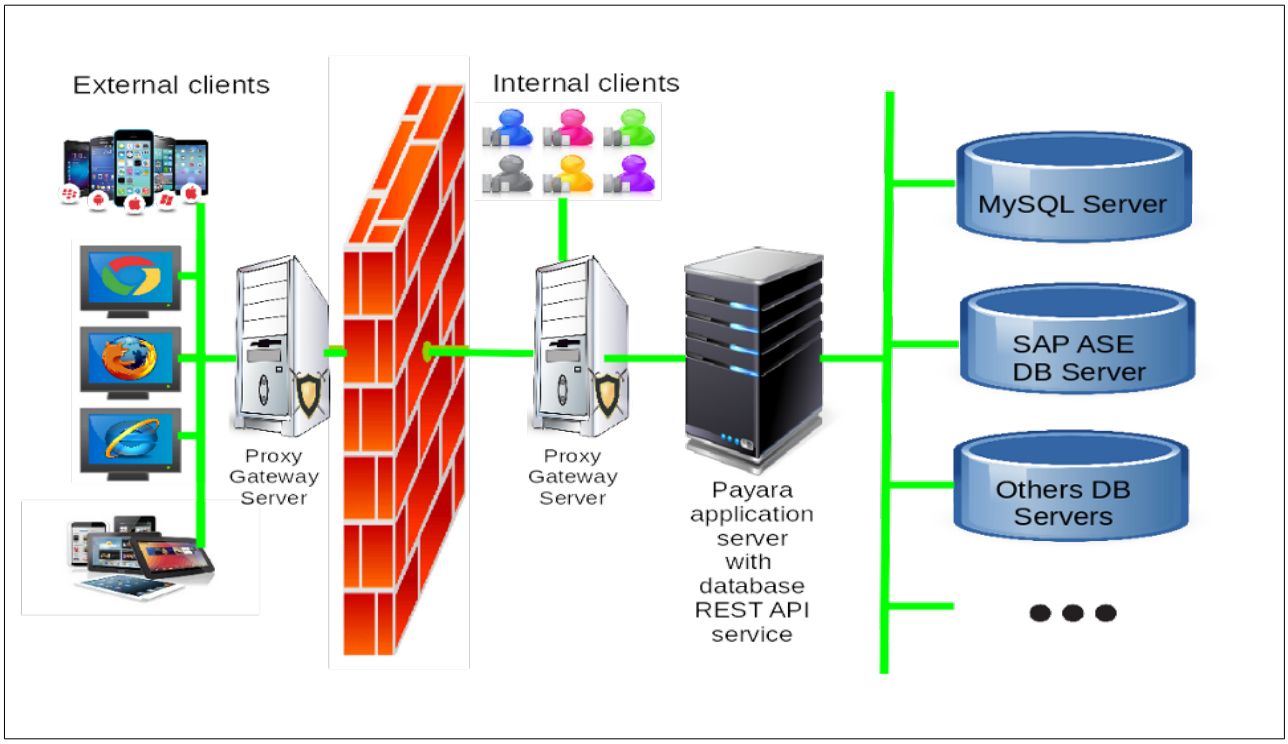

Figure 1: Diagram of Database REST API Service System.

In addition to serving the database REST API service, the Payara application server also provides several other REST API services as well. The system network firewall provides basic security checking for any possible illegal access of the system. The DB REST API has additional security measures to protect back end database resources.

THE DATABASE REST API

This database REST API attempts to cover all user requirements for normal database operation without the need to worry about changes to the database structure over time. This is accomplished by implementing all REST API features and functionality with standard JDBC connections at the back end.

This REST API service includes the following features:

- Query a database on meta data for a target database object (GET)
- Database CRUD operations through the HTTP protocol based operations (GET, PUT, POST and DELETE):
  - Create/Insert - (PUT/POST) insert new data into database
  - Read/Query - (GET) Query a database with desired data
  - Update - (PUT/POST) Update database data
  - Delete - (DELETE) Delete database data
- User authentication on CRUD operations. This is an additional security measure on the database server layer to secure the database resources.
- Service call error handling.
- Database change history logging for all successful database changes through the REST API calls.
- Format the returned data as XML or JSON.

All APIs share a common base HTTP URL path. Assuming this base path is defined as:

BASEPATH = http://host_domain_name:port/DBServer/

Below are three examples of the REST API service call URLs:

- **Query/GET - Query a database with desired data**
  e.g. BASEPATH/api/query?server=<server>&db=<db>&sql=<sql>&maxRecords=<maxrow>&page=<pageNumber>
- **Insert/POST insert new data into a database**
  e.g. BASEPATH/api/insert?server=<server>&db=<db>&table=<table>&datapairs=<column-value data pairs in JSON>&pid=<client_pid>&procname=<client_proc_name>
- **SQL/POST - Data change with SQL statement directly**
  e.g BASEPATH/api/sql?server=<server>&db=<db>&table=<table>&sql=<sql>&pid=<client_pid>&procname=<client_proc_name>

There are also more other auxiliary REST APIs server wide data such as list of all databases, all tables and objects in a database, etc.

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-TUPHA038

The API parameters used in above APIs are listed in Table 1 below.

Table 1: Description of Parameters used in all REST APIs

Name	Description	Example
server	The name of the database server : host name and port	host_name:5000
db	Name of a database to be queried	testdb
sql	SQL statement **NOTE:** The SQL statement is verified on server side before execution for security and avoid possible SQL injection attacks.	select * from authors;
table	Name of a table	authors
datapairs	Column vs value data pairs in JSON format	{"data": [{"field1":"value 1"}, {"field2":"value2 "}, {"field3":"value3 "},...]" }
where	The where clause for an update or delete SQL statement.	"where last='Smith' "
pid	client PID	"23456"
procname	client proc name	"UIDemo"

When a REST API call is made, all data is returned as JSON or XML string data. By default, the data is returned using JSON format. Users can explicitly specify the data format by setting the request header to either "Accept: application/json" or "Accept: application/xml".

On the server side, client requests are authenticated to prevent data resources from receiving denial of service (DOS) attacks or SQL injection attacks.

The returned data consists of three parts:

- Meta data provides the properties of the requested data.
- Data page information provides information such as page number, page row count, etc.
- Result data contains the requested data

THE REST API SECURITY CONTROL

The REST API service provides a convenient way to access database data. However, security is a significant concern. Although the API is designed to be used inside the normal network firewall, an additional API layer supporting user authentication is crucial to protect against unauthorized access.

Several technologies exist to enhance the REST API security. These include the use of API keys, openID Connect/OAuth2/SAML (A Security Assertion Markup Language), and session state management. Each system has its own pros and cons, and each adds additional complexity to the REST API system.

In this API design, we use HTTP Basic Authentication[5], enhanced with an API key, data concatenation patterns, and dynamical double data encoding. The client side and server side share the same secrete API key and secrete data processing knowledge, and the right procedure for data parsing, encoding and decoding.

The procedure used to implement these security measures is outlined below.

On client side:

Step1: Prepare user credential data that includes:

- **userID** (OS layer user id. In most cases, this is normally known after a user passed the firewall layer security.)
- **dbID** (Database server layer user ID)
- **dbPassword** (Database user password)

Concatenate these 3 parts into a single string with a designed concatenation pattern (shared by client and server only) and encode the concatenated string into string #1.

Step 2: Get the system current time stamp string and encode it into string #2.

Step 3: Concatenate the secrete API key (shared only by client and server), string#1 and string#2 with a designed concatenation pattern (shared by client and server) , and encode it into string#3.

Step 4: Send the REST API call request along with above encoded user info in HTTP Request Header**.**

"Authorization" = "Basic " + string#3

The above procedure can be easily implemented in any programming language. The double encoded user information can be sent over the network and can not be decoded easily without knowing the API key, string concatenate patterns, double data encoding and data parsing procedure.

On the server side, the users credential data and client submitted time stamp data can be decoded back by reversing the procedure. These decoded data are verified again on server side, and additional security controls can be applied on server side to prevent DOS attacks or throttle the server working loads. Client IP filter checking on the server side can also be used to further limit the

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-TUPHA038

clients if necessary. The OS level user verification on the server side also ensures the users are among the legal audience. In this design, using simple Base64 [6] encoding/decoding technology, we can reach a reasonably high level of database protection.

Client side Base64 based encoding is available for almost all programming languages such as Java, Perl, PHP, Python, Bash, XQuery, openSSL etc.

REST API ERROR HANDLING

HTTP errors (communication issues, time outs, etc.) are returned as standard HTTP error codes. This REST API follows the generic HTTP return status code settings:

- 200 indicates an OK status;
- Statuses >= 400 indicates a generic HTTP problem.
- A 500 status represents database SQL errors, and the errors are returned in the HTTP response header as a string with the title "DB-Server-Error".
- Client should catch the exception and check the HTTP response header to find out the details of the error message from the database server.

SUMMARY

REST API services provide a convenient way to make data resources of many kinds available through the HTTP protocol over the network with standard data exchange format such as XML and JSON. This bridges the gaps between different OS systems and platforms, and makes data operations especially easy through mobile and web applications. This API design takes advantage of the REST API capability and combines the power of other technology such as JDBC. With the Database REST API service, we are able to communicate with different database systems using a common method, eliminating the need to know how to connect to each database server. This makes database data retrieval and CRUD operations much easier from systems like mobile and web applications, Matlab applications, and other accelerator control applications using different programming languages. With a little more effort, this REST API can be applied to all JDBC supported database systems.

It is also worth noting that this convenience comes with a cost due to the additional layer between clients and the data resources. Resource intensive database operations could lead to a performance overhead, forcing developers to use judgement when selecting between a REST API service or direct communication with the database resources.

REFERENCES

[1] restifydb, http://restifydb.com/

[2] Drupal, https://www.drupal.org/

[3] Firebase Database REST API, https://firebase.google.com/docs/reference/rest/database/

[4] Payara Server Data Sheet. http://info.payara.fish/payara-server-data-sheet

[5] Basic access authentication. https://en.wikipedia.org/wiki/Basic_access_authentication

[6] Base64, https://en.wikipedia.org/wiki/Base64