FOUR Js

The Power of Simplicity

Programming with Genero BDL and NoSQL Databases

Sebastien FLAESCH, Reuben BARCLAY

February 2016

Table of Contents

1. Purpose of this white paper

Introduction to NoSQL databases (MongoDB, Informix-MongoDB, CouchDB), and show how to access a NoSQL DB from a Genero program.

Warning: In this white paper, we focus only on databases with JSON data models and APIs. NoSQL databases with other data models and APIs are not covered. Two other important aspects of NoSQL databases are scalability and security, which are not covered here.

2. Executive Summary

NoSQL (originally no sql or no relational) Databases are a form of databases that have grown in popularity in recent years. As the name suggests, they are databases modeled different from the relational databases typically used in Genero applications. Compared to a relational database, a NoSQL database is seen to have advantages in their simplicity of design, speed, ability to scale, and flexibility of data structures, but at the cost of lack of consistency and ACID transactions.

There are a number of different NoSQL database vendors out there and unlike relational databases, a lack of standards. This paper shows how Genero programs can read/write from/to 2 different NoSQL databases, MongoDB and CouchDB, using their proprietary REST APIs. In addition this paper shows how via Informix databases partnership with MongoDB, Informix SQL statements can be used to read/write to MongoDB. The paper focuses on a sub-class of NoSQL databases known as "Document stores", there are other sub-classes with names like key-value, graph that are not covered.

If you were using another NoSQL database that is not listed here, we would expect to find that they could read/write from/to them using similar REST APIs, or perhaps via JAVA or C libraries that can be imported and used by a Genero application. As long as there is an API using one of these techniques, there should be nothing stopping a Genero program from interacting with that particular NoSQL database. There are some areas where our Genero syntax could be extended to simplify coding that are listed in this document.

So there are no roadblocks stopping Genero developers working with NoSQL databases via the techniques shown in the examples in this document. The techniques used in this white paper can be used as stepping stones to a solution for other NoSQL databases.

3. What is a NoSQL database?

A NoSQL database can handle a large amount on data using a flexible schema, and servicing a lot of clients.

NoSQL database: Data flexibility + Service scalability

The NoSQL ecosystem includes several brands of database systems, using different data models (key-value, (JSON) document, column-based, graph). The original need came from companies like Facebook, Google and Amazon, for 21st Century Web demands to address the "Big Data" challenge.

NoSQL databases can be categorized into aggregate-oriented NoSQL databases (document-based, column-based, key-value-based) versus graph-oriented NoSQL databases (graph-based). Aggregate

oriented QL databases are designed to make complex queries on schema-less data (ex: map-reduce technology), while graph-oriented NoSQL databases is good at making complex queries on elements interconnected with an undetermined number of relations between them.

Some well known NoSQL databases: Cassandra, MongoDB, CouchDB, Redis, ElasticSearch, Riak, Hbase, Couchbase, OrientDB, Aerospike, Neo4j, Hypertable, Accumulo, VoltDB, Scalaris, RethinkDB ...

Resources:

- Intro to NoSQL by Martin Fowler: https://www.youtube.com/watch?v=qI_g07C_Q5I
- Brief overview of NoSQL solutions: http://kkovacs.eu/cassandra-vs-mongodb-vs-couchdb-vs-redis
- A visual guide to NoSQL Systems: http://blog.nahurst.com/visual-guide-to-nosql-systems
- IBM blog about NoSQL with Informix/MongoDB: https://www.ibm.com/developerworks/community/blogs/idsteam/resource/IFMX-12.10.xC2-WP-Overall_20131018.pdf?lang=en

4. Why use a NoSQL database?

The two main reasons you would use a NoSQL database over a more traditional relational SQL database are flexibility of schema and data structure, and speed with large data sets.

An SQL table structure enforces the same columns on every row of a database table. A NoSQL database does not enforce this, thus giving you greater flexibility. This greater flexibility not only accounts for changes in schemas over time but differences between rows. Different data structures such as key-value stores, document databases, graphs also give more flexibility about how data is arranged and accessed.

Relational databases typically scale vertically, that is, as the size of the database increases, so too must the system resources on the server. Adding servers does not increase performance proportionately. As a result, this equates to larger, more expensive hardware. A NoSQL database typically scales horizontally, that is as your data volumes increase, you can typically partition a database across several commodity servers.

5. NoSQL Databases used in this lab

5.1. MongoDB

MongoDB is a NoSQL database based on JSON Document data model.

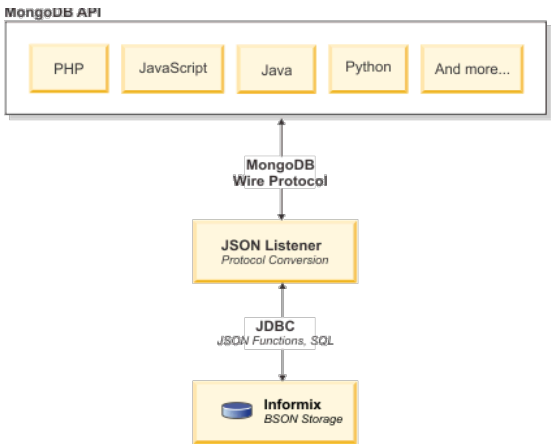
MongoDB: NoSQL DB based on JSON documents

MongoDB handles BSON/JSON documents (~= SQL records) in collections (~= SQL tables). Documents are identified by a (universal / world-wide) unique ObjectID (~= SQL primary key). For more details about MongoDB and the JSON data model, see https://docs.mongodb.org/manual/data-modeling/.

5.2. IBM Informix-MongoDB

IBM Informix IDS 12.10 introduced NoSQL support with the new JSON/BSON SQL data types and the MongoDB listener.

The Informix MongoDB listener acts as a native MongoDB server, using the IDS server for data storage with the BSON data type. The BSON data type is a binary/compacted version of the JSON data type.



Informix NoSQL jsonListener: Replace a MongoDB server

MongoDB client applications can directly access the Informix MongoDB server as if it was a native MongoDB server.

5.3. CouchDB

CouchDB is a NoSQL database based on JSON Document data model. The API to access data is HTTP/REST. Complex Map/Reduce queries are done in JavaScript. An interesting feature of CouchDB is Multi-Version Concurrency Control.

CouchDB: NoSQL DB based on JSON documents

Resource: http://couchdb.apache.org/

6. Operating system and software versions for the Lab

The OS used for this lab is a Linux Debian (Jessie). Command-line examples are Linux/UNIX specific. However, you can easily adapt the examples to use them on a computer running Microsoft Windows.

- Genero BDL **3.00.05**
- MongoDB **3.2.1**
- RESTHeart **1.1.3**
- Informix IDS **12.10.FC6**
- CouchDB **1.6.1**

Steps of a Genero program?

Below are the basic steps a Genero program would typically do with a NoSQL database:

1. Fetch JSON data from the NoSQL DB
2. Convert JSON to a RECORD as a document (or DYNAMIC ARRAY OF RECORD, as a collection)
3. View the data (DISPLAY+MENU or DISPLAY ARRAY), let the user modify the data (INPUT)
4. Convert RECORD back to JSON
5. Update data in the NoSQL DB with JSON

7. Access Informix-MongoDB data from Genero through SQL

7.1. Installing MongoDB (shell)

You need to install at least the MongoDB shell, to execute MongoDB commands (i.e. MongoDB CRUD operations) against the Informix server.

To check that the MongoDB shell / client is installed, try the following command:

```
$ mongo --help
MongoDB shell version: 3.2.1
usage ...
```

For more details about MongoDB installation, see https://docs.mongodb.org/manual/installation/

7.2. Configuring Informix for MongoDB

You need to setup your Informix IDS 12.10 server, to enable the MongoDB listener (jsonListener). If you have installed IDS 12.10 with default options, the jsonListener will be configured by default and started.

To check if the Informix jsonListener is started, log in as informix user and verify that the jsonListener.jar Java program is started:

```
$ ps -aef | grep jsonListener.jar
```

You might also want to check the jsonListener log file:

```
$ tail $INFORMIXDIR/jsonListener.log
```

You can also check the jsonListener configuration in the etc/jsonListener.properties file:

```
$ tail $INFORMIXDIR/etc/jsonListener.properties
```

If the jsonListener is not started, it needs to be configured; See Informix documentation for more details.

7.3. Create an Informix database for this Lab

A demo/test database needs to be created for this Lab in the Informix server.

Warning: Use and create the database with UTF-8 locale (CLIENT_LOCALE / DB_LOCALE).

```
$ echo $CLIENT_LOCALE
en_us.utf8
$ dbaccess - -
> create database jsontest with buffered log;
Database created.
```

By default the Informix jsonListener is configured to execute SQL database operations as the ifxjson user. Thus, you need to grant permissions for the ifxjson user on the new created database:

```
> grant connect to ifxjson;
```

```
Permission granted.
> grant resource to ifxjson;
Permission granted.
```

Test access to the new created database with the MongoDB shell:

```
$ mongo 127.0.0.1:33345/jsontest
MongoDB shell version: 3.2.1
connecting to: 127.0.0.1:21408/jsontest
> show collections;
>
```

7.4. Introduction to MongoDB CRUD

In this section, we will use the MongoDB shell to create a collection and a set of documents.

Note: This is just an introduction to the MongoDB query language, you can of course do much more.

First, open a session to the jsontest database:

```
$ mongo 127.0.0.1:33345/jsontest
MongoDB shell version: 3.2.1
connecting to: 127.0.0.1:21408/jsontest
>
```

Then, create a new collection by executing a search with the find() method:

```
> db.c1.find();
> show collections;
c1
```

You can see that the collection “c1” has been created.

Now create a new document in this collection:

```
> db.c1.insert({_id:1001, name:"Mike Philips", phone:"234-122-
9342" });
> db.c1.find();
{ "_id" : 1001, "name" : "Mike Philips", "phone" : "234-122-9342" }
```

You can see that when we do a search with the find() method, we get the new created BSON/JSON document.

Note: In this example we explicitly specify an _id field. If you do not specify an _id field, MongoDB will automatically produce a unique object id to identify the record.

Create another document with different data:

```
> db.c1.insert({_id:1002, name:"Ted Barber", phone:"762-331-3411" });
> db.c1.find();
{ "_id" : 1001, "name" : "Mike Philips", "phone" : "234-122-9342" }
{ "_id" : 1002, "name" : "Ted Barber", "phone" : "762-331-3411" }
```

To find a specific document by name:

```
> db.c1.find({name: "Mike Philips"});
```

To add a projection clause to extract only some fields of the collection/documents:

```
> db.c1.find({name: "Mike Philips"},{phone:1,_id:0});
```

To sort by name (1 = ascending order, -1 = descending order):

```
> db.c1.find().sort({name:-1});
```

To modify an existing document by id:

```
> db.c1.update({_id: 1001},
               {$set:{phone:"999-999-9999"}} );
```

To add a field/value to an existing document by id:

```
> db.c1.update({_id: 1001},
               {$set:{address:"5 Platter St."}} );
```

To remove a field/value from an existing document by id:

```
> db.c1.update({_id: 1001},
               {$unset:{address:""}} );
```

To delete an existing document by id:

```
> db.c1.remove({_id: 1001});
```

For more details about MongoDB CRUD operations, see https://docs.mongodb.org/manual/core/crud-introduction/

7.5. Preparing a sample collection for the demo

Before jumping to the Genero BDL side, let's first create a new MongoDB collection containing people data for a typical contact application (with phone numbers, address, etc).

This collection will be loaded from a JSON data file provided at the end of this section, by using the mongoimport command:

```
$ mongoimport --host 127.0.0.1:33345 --db jsontest \
              --collection contacts < contacts.json
```

The JSON data structure of the contacts collection looks as follows:

```
{
  _id : contact-number,
  name : contact-name,
  loc : { longitude : longitude, latitude : latitude },
  address : contact-address,
  city : contact-city-name,
  phone : contact-phone-num,
  birth : contact-birth-date
}
```

Note that:

- The “loc” field is defined as a JSON sub-record.
- The “birth” field is a string representing a date/time in ISO-8601 format, in UTC (it is not a $date nor a ISODate() MongoDB-extended-JSON date).

Here the JSON data that can be copy/pasted into a .json file for mongoimport:

```
{ "_id" : 1001, "name" : "Scott Beckham",   "loc" : {"longitude":-
95.1304080,"latitude":39.559411}, "address" : "28 Hollywhite Av.", "city" :
"ATCHISON",  "phone" : "134-263-2346", "birth" :"2001-12-23T14:55:00Z" }

{ "_id" : 1002, "name" : "John Kinsley",    "loc" : {"longitude":
-94.964713,"latitude":38.956563}, "address" : "23 Big Tod St.",    "city" : "DE
SOTO",   "phone" : "234-333-7812", "birth" :"1989-03-12T02:12:00Z" }

{ "_id" : 1003, "name" : "Philip Desmond",  "loc" : {"longitude":
-95.118759,"latitude":38.352428}, "address" : "5 Bakers St.",      "city" :
"GREELEY",   "phone" : "234-333-7812", "birth" :"1967-08-01T12:12:00Z" }

{ "_id" : 1004, "name" : "Mike Kobain",     "loc" : {"longitude":
-95.258335,"latitude":39.860752}, "address" : "213 Cartoon St.",   "city" :
"HIGHLAND",  "phone" : "345-132-8888", "birth" :"2000-11-04T00:33:00Z" }

{ "_id" : 1005, "name" : "Ted Fitzpatrick", "loc" : {"longitude":
-94.858369,"latitude":38.684463}, "address" : "82 Smallhill Av.",  "city" :
"HILLSDALE", "phone" : "999-442-2435", "birth" :"1978-10-14T23:45:00Z" }
```

7.6. Creating an SQL View

The simplest way to access Informix MongoDB BSON data from a Genero program is to prepare the database by creating a view based on the target collection:

```
CREATE VIEW contacts_view
  ( _id, name, address, city, phone, birth )
AS SELECT
  BSON_VALUE_BIGINT(data, '_id'),
  BSON_VALUE_LVARCHAR(data, 'name'),
  BSON_VALUE_LVARCHAR(data, 'address'),
  BSON_VALUE_LVARCHAR(data, 'city'),
  BSON_VALUE_LVARCHAR(data, 'phone'),
  BSON_VALUE_LVARCHAR(data, 'birth')
 FROM contacts;
```

After creating the view, the BDL program can make a regular SQL query as shown in the next example using dbaccess:

```
> SELECT * FROM contacts_view WHERE _id = 1004;
_id      1004
name     Mike Kobain
address  213 Cartoon St.
city     HIGHLAND
phone    345-132-8888
birth    2000-11-04T00:00:00Z
1 row(s) retrieved.
```

7.7. Accessing raw BSON data from Genero

Genero programs can use Informix BSON_* functions to query and modify BSON data through SQL.

The code example in Appendix A implements a program that loads the data from the contacts collection into a program array, and allows the user to browse and edit the rows with a DISPLAY ARRAY dialog:

Below is an explanation of some of the code snippets with syntax you have most likely not come across before ...

```
DECLARE c1 CURSOR FROM
    "SELECT data::JSON::LVARCHAR FROM contacts"
    || " ORDER BY BSON_VALUE_LVARCHAR(data,'name')"
LET x=0

FOREACH c1 INTO jsdata
    CALL util.JSON.parse(jsdata, r_contact)
```

The SELECT statement casts the BSON data column to ::JSON::LVARCHAR, so it can be fetched into a STRING BDL variable. The JSON string is then parsed with util.JSON.parse() to be converted to a BDL record. Note that the BDL record structure has a nested sub-record for the longitude/latitude.

```
LET json_obj = util.JSONObject.fromFGL(r_contact)
-- Convert datetime back to UTC/ISO-8601 (BDL datetime is in local time format)

CALL json_obj.put("birth", local_to_UTC_ISO8601(r_contact.birth) )
PREPARE s1 FROM "UPDATE contacts"
      || " SET data = ?::JSON::BSON"
      || " WHERE BSON_VALUE_VARCHAR(data,\"_id\") = ?"
LET json_rec = json_obj.toString()
EXECUTE s1 USING json_rec, r_contact._id
```

To modify a document, the UPDATE statement uses a ? parameter for a STRING representing the document, and casts the string value with ::JSON::BSON to convert it back to BSON data.

```
" WHERE BSON_VALUE_VARCHAR(data,\"_id\") = ?"
```

Informix-specific BSON_* functions are used to convert BSON document fields to SQL data when needed.

```
CALL json_obj.put("birth", local_to_UTC_ISO8601(r_contact.birth) )
```

JSON date/time values are represented with timezone information. When converting JSON to a BDL DATETIME, the util.JSON.parse() function makes the conversion to the local time. In order to modify the date/time in a BSON document, we need to convert the BDL (local) DATETIME value back to a ISO-8601 representation, in UTC.

Note also that the above Genero BDL program can be used unchanged, even if the original JSON collection gets more fields. For example, execute the following command with the MongoDB shell to add an email field to a document:

```
> db.contacts.update({_id:1003}, {$set:{email:"jph@topsoft.com"}});
```

Then execute the BDL program again to check that it is still compatible...

7.8. Dealing with MongoDB extended JSON

MongoDB defines JSON extensions to handle typed data, for example:

- MongoDB is able to generate unique binary document ids in a BSON document. The ObjectID("*hexadecimal-value*") function can convert an hexadecimal string to an object id value. The ObjectID("*hexadecimal-value*") notation is also used when representing the object id in a JSON string.
- MongoDB can store date/time values in a BSON document as a number of milliseconds since epoch UTC. The string representation of such date/time data will be ISODate("YYYY-MM-DDTHH:mm:ss.mmm<+/-Offset>")

For example, let's create a new collection with auto-generated document ids and a date/time field:

```
> db.c2.insert({name:"Ted Barder", crea:ISODate("2016-01-
23T11:22:33")});
```

```
> db.c2.find();
```

```
{ "_id" : ObjectId("5693b4c71b52692c8b71c2dd"), "name" : "Ted
Barder", "crea" : ISODate("2016-01-23T11:22:33Z") }
```

Since this is not standard JSON, the Genero BDL program will have to deal with this extra JSON notation and convert the ObjectID() and ISODate() expressions to BDL data.

> **Warning:** If a BSON document contains extended JSON data such as MongoDB's object id or date, the JSON representation will use the ObjectID() or ISODate() notations. Since the Genero BDL util.JSON API is based on standard JSON specification, it will not be able to make an automatic conversion to/from BDL data.

Replacing the ObjectID() and ISODate() expressions in a MongoDB JSON string can be done with Genero BDL function. However, this is quick and incomplete solution (a full tokenizer should be used instead, to distinguish string and keyword tokens):

```
FUNCTION mongoJSONToStringJSON(s)
    DEFINE b base.StringBuffer
    DEFINE s STRING
    DEFINE i, j INT
    LET b = base.StringBuffer.create()
    CALL b.append(s)
    LET i = b.getIndexOf("ObjectId(", 1)
    WHILE i > 1
        CALL b.replaceAt(i, LENGTH("ObjectId("),"");
        LET j = b.getIndexOf(")", i)
        CALL b.replaceAt(j, 1, "")
        LET i = b.getIndexOf("ObjectId(", j)
    END WHILE
    LET i = b.getIndexOf("ISODate(", 1)
    WHILE i > 1
        CALL b.replaceAt(i, LENGTH("ISODate("), "");
        LET j = b.getIndexOf(")", i)
        CALL b.replaceAt(j, 1, "")
        LET i = b.getIndexOf("ISODate(", j)
    END WHILE
    RETURN b.toString()
```

```
END FUNCTION
```

Note that MongoDB can also use a “strict format”, for example with mongoexport and the HTTP/REST API. For example, if we export the c2 collection created before, we'll get { $oid : “...” } and { $date : “...” } strict notations instead of ObjectID() and ISODate() expressions:

```
$ mongoexport --host 127.0.0.1:33345 --db jsontest --collection c2
connected to: 127.0.0.1:21408
{ "_id" : { "$oid" : "5693b7981b52692c8b71c2de" }, "name" : "Ted Barder", "crea" : { "$date" : 1453548153000 } }
exported 1 records
```

For more details, see https://docs.mongodb.org/manual/reference/mongodb-extended-json

7.9. Missing Informix-MongoDB features

Some document Informix-NoSQL/MongoDB features are not implemented in IDS version 12.10.FC6 used for this lab. For example, the BSON_GET() function is unknown in 12.10.FC6. You get error 674: Routine (bson_get) can not be resolved by executing the sample code provided here: https://www-01.ibm.com/support/knowledgecenter/SSGU8G_12.1.0/com.ibm.sqls.doc/ids_sqs_2340.htm?lang=en.

8. Access (native) MongoDB NoSQL data from Genero

In this section we will show how to setup a MongoDB database, and access it from Genero BDL by using an HTTP/REST API, provided by a software component called RESTHeart.

8.1. MongoDB basics

MongoDB is a NoSQL database based on JSON documents.

Several MongoDB clients exists: MongoDB implements a documented binary protocol. In this lab, we will use the RESTHeart software component on top of the MongoDB server. RESTHeart provides the HTTP/REST API.

With MongoDB, you define databases, containing collections of JSON documents.

As stated before, this white paper does not cover replication/scalability, but keep in mind that this is an important feature of MongoDB.

For more details, see https://docs.mongodb.org/manual/

8.2. Install and setup MongoDB

Before starting with this lab, you need to setup a MongoDB database server.

The version tested here is MongoDB 3.2.1.

Get the MongoDB package and install it (see MongoDB web site for installation instructions).

Create a directory to hold database files:

```
$ cd <your-location-to-mongodb-files>
$ mkdir data
```

Create a configuration file (mongod.conf), for example:

```
processManagement:
  fork: true
net:
  bindIp: 127.0.0.1
  port: 27017
storage:
  dbPath: /opt3/dbs/mgo/3.2.1/data
systemLog:
  destination: file
  path: "/opt3/dbs/mgo/3.2.1/mongod.log"
  logAppend: true
storage:
  journal:
    enabled: true
```

(Note that the HTTP/REST interface is enabled with the `RESTInterfaceEnabled` option)

Start the MongoDb process:

```
$ mongod --config mongod.conf
about to fork child process, waiting until server is ready for
connections.
forked process: 21337
child process started successfully, parent exiting
```

You might want to inspect the mongod.log file for information.

To stop the server:

```
$ mongod --config mongod.conf –shutdown
killing process with pid: 21337
```

Test access to the server with the MongoDB shell:

```
$ mongo 127.0.0.1:27017/jsontest
MongoDB shell version: 3.2.1
connecting to: 127.0.0.1:21408/jsontest
> show collections;
>
```

Resource: http://couchdb.apache.org/

8.3. Preparing a sample collection for the demo

Import the "contact" collection (from the Informix-NoSQL lab) into the MongoDB database:

```
$ mongoimport --host 127.0.0.1:27017 --db jsontest --collection
contacts < contacts.json
2016-01-13T13:59:54.517+0100  connected to: 127.0.0.1:27017
2016-01-13T13:59:54.555+0100  imported 5 documents
```

8.4. Install and setup RESTHeart

MongoDB 3.2 comes with a HTTP/REST interface, but in 3.2 this API is dedicated to administration tasks and the doc even suggests to disable it for production servers.

In order to perform REST commands, we will use another software called RESTHeart, "The web API for MongoDB". Note that other REST API solutions are available for MongoDB, for more details see https://docs.mongodb.org/ecosystem/tools/http-interfaces/

The version tested here is RESTHeart 1.1.3.

Download and unzip the RESTHeart package (see http://www.restheart.org/)

Setup the RESTHeart configuration file (etc/restheart.yml) to enable the HTTP listener (for REST API programming tests, no authentication mechanism will be configured):

```
…
http-listener: true
http-host: 0.0.0.0
http-port: 27080                          ← port for HTTP requests
…
mongo-uri: mongodb://127.0.0.1:27018  ← URI to access MongoDB
…
```

Review the RESTHeart security settings and user definitions in the etc/security.yml file:

```
users:
…
    - userid: user
      password: fourjs
      roles: [users]
…
    - userid: admin
      password: fourjs
      roles: [users, admins]
```

Note: By default, the "users" role defined in RESTHeart security configuration file allows only reads (GET). You must cannot as a user with "admins" role to modify data (PUT/POST/PATCH/DELETE).

Start the RESTHeart daemon, passing the restheart.yml configuration file as argument:

```
$ java -server -jar restheart.jar ./etc/restheart.yml
```

```
15:30:51.363 [main] INFO  org.restheart.Bootstrapper - Starting
RESTHeart
15:30:51.366 [main] INFO  org.restheart.Bootstrapper - version 1.1.3
15:30:51.371 [main] INFO  org.restheart.Bootstrapper - Logging to
console with level INFO
…
15:31:22.481 [main] INFO  org.restheart.Bootstrapper - RESTHeart
started
```

Open a web browser and test the following URL (you need to enter a user/pswd as defined in the etc/security.yml configuration file):

```
http://127.0.0.1:27080/browser
```

The “HAL Browser” home page should appear...

Now try a CURL command from the command line (in this example we pass the result to jsonlint to beautify the JSON data):

```
$ curl -u user:fourjs -X GET
    http://127.0.0.1:27080/jsontest/contacts | jsonlint --format
…
{
  "_collection-props-cached" : false,
  "_created_on" : "2016-01-13T15:22:25Z",
  "_embedded" : { "rh:doc" : [
          {
            "address" : "82 Smallhill Av.",
            "birth" : "1978-10-14T23:45:00Z",
            "city" : "HILLSDALE",
            "loc" : {
                "latitude" : 38.684463,
                "longitude" : -94.858369
              },
            "name" : "Ted Fitzpatrick",
            "phone" : "999-442-2435",
            "_id" : 1005,
            "_links" : { "self" : { "href" :
"/jsontest/contacts/1005?id_type=NUMBER" } }
          },
…
```

To query a single document, use the URI in the form:

server/*database*/*collection*/*id-value*?id_type=NUMBER

For example:

```
$ curl -u user:fourjs -X GET
http://127.0.0.1:27080/jsontest/contacts/1002?id_type=NUMBER |
jsonlist –format

{
  "address" : "23 Big Tod St.",
  "birth" : "1989-03-12T02:12:00Z",
  "city" : "DE SOTO",
  "loc" : {
      "latitude" : 38.956563,
      "longitude" : -94.964713
    },
  "name" : "John Kinsley",
  "phone" : "234-333-7812",
  "_id" : 1002,
  "_links" : {
      "curies" : [  ],
      "self" : { "href" : "/jsontest/contacts/1002?id_type=NUMBER" }
    }
}
```

Note: RESTHeart represents resources using the "HAL format" where resources have state, **embedded resources** and links: a **GET on a collection resource, returns its documents as embedded resources.**

For more details about the HAL format see: **http://stateless.co/hal_specification.html**

8.5. Query RESTHeart/MongoDB from Genero

To query the MongoDB database we will use Genero Web Services REST API, in order to execute HTTP requests on the RESTHeart server.

We need to know the RESTHeart API to program the application. The RESTHeart API is based on the HAL specification.

> **Warning:** RESTHeart comes with additional features on top of the standard MongoDB features, to ease Web application programming. One of these features is Etags usage (comes from HTTP Etags used for web cache). Each document created (or PATCHed) through RESTHeart will get an _etag field, that will be used to do optimistic concurrency control. In order to update a JSON document through RESTHeart (1.1), you must provide the current Etag in a If-Match header of the POST HTTP request. Problem is when you want to update JSON data that was created from another source as RESTHeart, where documents have no _etag field!
>
> See https://softinstigate.atlassian.net/wiki/display/RH/ETag
>
> Discussed with RESTHeart implementers: Version 1.1 has this constraint, but version 1.2 will make Etags usage optional. Workaround is to PATCH existing documents by changing an existing field, to get _etag fields added (despite you get a 409 HTTP error (Conflict).
>
> See https://softinstigate.atlassian.net/browse/RH-147

In order to update JSON documents through the RESTHeart API, we must add _etag fields to the original documents of the contacts collection...

So first of all we'll patch the existing documents to get a _etag fields for each of them. Just run this shell script (you need the curl utility):

```
for id in 1001 1002 1003 1004 1005
do
    curl -X PATCH \
         http://admin:fourjs@127.0.0.1:27080/jsontest/contacts/${id}?
id_type=NUMBER \
         -H "Content-Type: application/json" \
         -d "{_id:${id}}"
done
```

> **Important:** These commands return HTTP error HTTP/1.1 409 Conflict... this is expected with the PATCH method, according to RESTHeart people.

Check that the documents have got an _etag with mongo shell:

```
> db.contacts.find({},{_id:1,_etag:2});
{ "_id" : 1002, "_etag" : ObjectId("569912d32dedc53866216a13") }
{ "_id" : 1004, "_etag" : ObjectId("569912d32dedc53866216a15") }
{ "_id" : 1003, "_etag" : ObjectId("569912d32dedc53866216a14") }
{ "_id" : 1005, "_etag" : ObjectId("569912d32dedc53866216a16") }
{ "_id" : 1001, "_etag" : ObjectId("569912d32dedc53866216a12") }
```

Now we can use the data in the Genero program to implement basic CRUD operations.

Source is available in Appendix B.

Note that the _etag field has to be converted from the $oid notation:

```
"_etag" : { "$oid" : "56992c052dedc53866216a1b"}
```

This is done at load time with following code:

```
IF json_rec.has("_etag") THEN
   LET json_etag = json_rec.get("_etag")
   LET ca[x]._etag = json_etag.get("$oid")
ELSE
   ...
```

To create, update or delete a document, the “If-Match” header must be set with the appropriate Etag:

- To create a new document, we must pass the Etag returned from last GET.
- To update or delete a document, we must to pass the Etag of the document.

```
IF new THEN
   LET method = "POST"
   LET expcode = 201
   LET etag = get_etag
ELSE
   LET method = "PUT"
   LET expcode = 0
   LET etag = ca[x]._etag
END IF
LET json_rec = json_obj.toString()
CALL restheart_operation(method, contact_uri(ca[x]._id), NULL,
                         etag, json_rec, expcode)
         RETURNING err, new_etag, result
```

9. Access CouchDB NoSQL data from Genero

In this section we will show how to setup an Apache CouchDB database, and access to it from Genero BDL by using the HTTP/REST API provided by CouchDB.

9.1. Apache CouchDB basics

CouchDB is a JSON-document-based NoSQL database, that implements an HTTP/REST API: http://docs.couchdb.org/en/latest/api/basics.html

For queries, CouchDB uses map and reduce functions (implemented in JavaScript). This provides great flexibility as it can adapt to variations in document structure, and indexes for each document can be computed independently and in parallel. The combination of a map and a reduce function is called a view in CouchDB terminology.

CouchDB implements Multi-Version Concurrency Control: When modifying a document, CouchDB will actually create a new version of the document. The automatic `_rev` field identifies a revision of a document (the `Etag` HTTP header field is used to show the revision for a document, or a view). When deleting a document, CouchDB sets the automatic `_deleted` field to true with a new `_rev` revision, but the document is not really dropped.

CouchDB also provides per-document validation from within the database, using JavaScript functions called each time a document is modified, to approve or deny the update.

As stated before, this white paper does not cover replication/scalability, but keep in mind that this is an important feature of CouchDB.

For more details, see http://docs.couchdb.org/en/1.6.1/intro/consistency.html

9.2. Install and setup CouchDB

Before starting with this lab, you need to setup a CouchDB database server.

The version tested here is CouchDB 1.6.1.

Get the CouchDB package and install it (see CouchDB web site for installation instructions).

In this lab we have downloaded the sources and installed in a specific directory by doing:

```
$ cd <couchdb-sources-dir>
$ ./configure --prefix <your-install-dir>
...
$ make all install
```

CouchDB configuration files are located in `<your-install-dir>/etc/couchdb`: `default.ini` file and `local.ini` files: `default.ini` is overwritten during next installation, while `local.ini` can be used to customize CouchDB).

As we will run couchdb server as a regular user, we'll comment out the COUCHDB_USER definition in the file <*your-install-dir*>/etc/default/couchdb:

```
#COUCHDB_USER=couchdb
COUCHDB_STDOUT_FILE=/dev/null
COUCHDB_STDERR_FILE=/dev/null
COUCHDB_RESPAWN_TIMEOUT=5
COUCHDB_OPTIONS=
```

Start the couchdb server with the init.d script:

```
$ ./1.6.1/etc/init.d/couchdb start
[ ok ] Starting database server: couchdb.
```

Check that the couchdb server is running:

```
$ ./1.6.1/etc/init.d/couchdb status
Apache CouchDB is running as process 7899, time to relax.
$ curl http://localhost:5984/
{"couchdb":"Welcome","uuid":"943232ac8fba553697c9b746f9f3cd0b","versi
on":"1.6.1","vendor":{"version":"1.6.1","name":"The Apache Software
Foundation"}}
```

CouchDB includes a web-based front end called Futon which can be accessed by your browser via the following address:

```
http://localhost:5984/_utils
```

Security configuration: By default, no admin/member users are defined, thus the database is public.

9.3. Prepare the CouchDB database for Genero

First let's create a new database with the CURL utility:

```
$ curl -X PUT http://127.0.0.1:5984/jsontest
{"ok":true}
```

Check that the new database is there:

```
$ curl -X GET http://127.0.0.1:5984/_all_dbs
["_replicator","_users","jsontest"]
```

Unlike MongoDB, CouchDB does not have the concept of collection: Below a CouchDB database, everything is a document. You can however define an application `type` field, to give a type to a document and identify its structure to implement the map/reduce “views” to make queries.

First we need to load the contacts into our CouchDB database. This will be done by doing a “bulk insert/update”, using the *_bulk_docs* endpoint in the HTTP request. The JSON data file needs to be adapted to match bulk insert/update input format:

```
{
 "docs": [
  { "_id" : "1001", "type":"contacts", "name" : "Scott Beckham",   "loc" :
{"longitude":-95.1304080,"latitude":39.559411}, "address" : "28 Hollywhite Av.",
"city" : "ATCHISON",  "phone" : "134-263-2346", "birth" :"2001-12-23T14:55:00Z" },
  { "_id" : "1002", "type":"contacts", "name" : "John Kinsley",    "loc" :
{"longitude": -94.964713,"latitude":38.956563}, "address" : "23 Big Tod St.",
"city" : "DE SOTO",   "phone" : "234-333-7812", "birth" :"1989-03-12T02:12:00Z" },
  { "_id" : "1003", "type":"contacts", "name" : "Philip Desmond",  "loc" :
{"longitude": -95.118759,"latitude":38.352428}, "address" : "5 Bakers St.",
"city" : "GREELEY",   "phone" : "234-333-7812", "birth" :"1967-08-01T12:12:00Z" },
  { "_id" : "1004", "type":"contacts", "name" : "Mike Kobain",     "loc" :
{"longitude": -95.258335,"latitude":39.860752}, "address" : "213 Cartoon St.",
"city" : "HIGHLAND",  "phone" : "345-132-8888", "birth" :"2000-11-04T00:33:00Z" },
  { "_id" : "1005", "type":"contacts", "name" : "Ted Fitzpatrick", "loc" :
{"longitude": -94.858369,"latitude":38.684463}, "address" : "82 Smallhill Av.",
"city" : "HILLSDALE", "phone" : "999-442-2435", "birth" :"1978-10-14T23:45:00Z" }
 ]
}
```

Important: The "_id" values must be strings, and that each record gets a "type" field, to group all these documents as contacts, since CouchDB there is no collection concept as in MongoDB.

Execute the following command, to create the documents in our CouchDB database (note that here we use the *_bulk_docs* endpoint in the URI to create several documents in one HTTP request):

```
$ curl -X POST -H "Content-Type: application/json" -d @contacts.json
http://127.0.0.1:5984/jsontest/_bulk_docs
[{"ok":true,"id":"1001","rev":"1-95b8cbe0b7c1d2aac875db5b1579fb38"},
{"ok":true,"id":"1002","rev":"1-f7058ef914696b7df36072816cd0c36b"},
{"ok":true,"id":"1003","rev":"1-cd242d8ca9ed0d11c9e90e576b0ed339"},
{"ok":true,"id":"1004","rev":"1-8b9523ebdfd2b60d7c0d44df3d964155"},
{"ok":true,"id":"1005","rev":"1-9a95b607f5576d2b6aca78933bb4b48c"}]
```

The result of this HTTP request is a list of ok/id/rev fields for each inserted contact record.

To check that our documents have been created, go to the FUTON console and verify that the documents have been created ...

To GET a single document, just pass the _id as URI endpoint:

```
$ curl -X GET http://127.0.0.1:5984/jsontest/1001 | jsonlint --format
{
  "address" : "28 Hollywhite Av.",
  "birth" : "2001-12-23T14:55:00Z",
  "city" : "ATCHISON",
  "loc" : {
      "latitude" : 39.559410999999997216,
      "longitude" : -95.130408000000002744
    },
  "name" : "Scott Beckham",
  "phone" : "134-263-2346",
  "type" : "contacts",
  "_id" : "1001",
  "_rev" : "1-004add87be732772b3a1c53fcfa6d09d"
}
```

To GET several documents, use the *_all_docs* endpoint with the *?include_docs=true* as query parameter.

Warning: This query will return ALL documents of a CouchDB database! You need to create a CouchDB view to select only the contacts documents with type=”contacts”.

```
$ curl -X GET http://127.0.0.1:5984/jsontest/_all_docs?
include_docs=true
{
  "offset" : 0,
  "rows" : [
      {
        "doc" : {
            "address" : "23 Big Tod St.",
            "birth" : "1989-03-12T02:12:00Z",
            "city" : "DE SOTO",
            "loc" : {
                "latitude" : 38.956563000000002717,
                "longitude" : -94.964713000000003262
              },
            "name" : "John Kinsley",
            "phone" : "234-333-7812",
            "type" : "contacts",
            "_id" : "1002",
            "_rev" : "1-7af927c7c55c4b80e8a42289fdb6b680"
          },
        "id" : "1002",
        "key" : "1002",
        "value" : { "rev" : "1-7af927c7c55c4b80e8a42289fdb6b680" }
      },
      …
```

```
]}
```

To create a single document, use a POST HTTP request, passing the id as URI endpoint:

```
$ curl -v -X POST -H "Content-Type: application/json" -d
'{"_id":"1006","name":"New contact","phone":"999-999-9999"}'
http://127.0.0.1:5984/jsontest | jsonlint -format
{
  "id" : "1002",
  "ok" : true,
  "rev" : "1-e76537606aa276677c176469c4593e81"
}
```

To modify a single document, use a PUT HTTP request, passing the id as URI endpoint and the revision in the query string:

```
$ curl -X PUT -H "Content-Type: application/json" -d '{"phone":"999-
999-9999"}' http://127.0.0.1:5984/jsontest/1002?rev=1-
7af927c7c55c4b80e8a42289fdb6b680 | jsonlint -format
{
  "id" : "1002",
  "ok" : true,
  "rev" : "2-b625bbfb0d503f416a567239534b9e94"
}
```

Note that if you retry now the same command without changing to the last revision, you'll get an update conflict:

```
$ curl -X PUT -H "Content-Type: application/json" -d '{"phone":"999-
999-9999"}' http://127.0.0.1:5984/jsontest/1002?rev=1-
7af927c7c55c4b80e8a42289fdb6b680 | jsonlint -format
{
  "error" : "conflict",
  "reason" : "Document update conflict."
}
```

To delete a document, use a DELETE HTTP request pass the id in the URI and the revision in the query string:

```
$ curl -X DELETE http://127.0.0.1:5984/jsontest/1001?rev=1-
004add87be732772b3a1c53fcfa6d09d
{"ok":true,"id":"1001","rev":"2-3703681757bf20bae8c920d2cee89798"}
```

(remember the document is not definitively dropped, it just gets a _deleted flag set to true...)

If needed, CouchDB is able to generate UUIDs with the __uuids URI endpoint:

```
curl -X GET http://127.0.0.1:5984/_uuids
{"uuids":["af59a1d39a9c5745207613cfa50304c3"]}
```

You can then use this unique id to create a new document.

Then fetch several records according to a list of ids, use a POST request and pass the list of ids as data with a “keys” field:

```
$ curl -X GET -d '{"keys":["1001","1002"]}'
http://127.0.0.1:5984/jsontest/_all_docs?include_docs=true
{
  "offset" : 0,
  "rows" : [
      {
        "doc" : null,
        "id" : "1001",
        "key" : "1001",
        "value" : {
            "deleted" : true,      <= You get also rows marked as deleted!
            "rev" : "2-3703681757bf20bae8c920d2cee89798"
          }
      },
      {
        "doc" : {
            "address" : "23 Big Tod St.",
            "birth" : "1989-03-12T02:12:00Z",
            "city" : "DE SOTO",
            "loc" : {
                "latitude" : 38.956563000000002717,
                "longitude" : -94.964713000000003262
              },
            "name" : "John Kinsley",
            "phone" : "234-333-7812",
            "type" : "contacts",
            "_id" : "1002",
            "_rev" : "1-7af927c7c55c4b80e8a42289fdb6b680"
          },
        "id" : "1002",
        "key" : "1002",
        "value" : { "rev" : "1-7af927c7c55c4b80e8a42289fdb6b680" }
      }
    ],
  "total_rows" : 4          <= 4 rows??? we can see 2 => CouchDB Bug?
}
```

For more details about CouchDB's HTTP API for documents, see
http://docs.couchdb.org/en/1.6.1/intro/api.html#documents

9.4. CouchDB views

Unlike traditional SQL databases where you can do ad-hoc SQL queries, CouchDB uses predefined map and reduce (JavaScript) functions in a style known as MapReduce, in order to filter and sort (i.e. map) documents and potentially produce aggregates/summary (i.e. reduce). The combination of a map and a reduce function is called a view in CouchDB terminology.

Basically:

- Map function: Takes a document as parameter and emit() one or more view rows as key/value pairs from it. Map functions must not depend on any information outside of the document: This independence is what allows CouchDB views to be generated incrementally and in parallel.
- Reduce function: Takes the output from a map function as input and combines those data tuples into a smaller set of tuples.

The essence of Map/Reduce technology is to be able execute queries on a very large amount of data, doing parallel processing.

In CouchDB, Map/Reduce functions (i.e. views), are stored in the database as documents, in the `_design` database.

Since this technique requires to predefine views the CouchDB database, in order to make queries. Thus it is not possible to make any sort of ad-hoc query from a Genero program with this technology.

The easiest way to create views in CouchDB is to use the FUTON interface. Open your favorite browser and enter the following address:

```
http://localhost:5984/_utils
```

Select the jsontest database, and in the View: field select “Temporary view” and enter following JavaScript code in the Map function block:

```
function(doc) {
  if (doc.type == "contacts") {
    if (doc.loc) {
      var key = [doc._id, doc.name];
      emit(key, doc.loc);
    }
  }
}
```

In this Map function we filter on the document type field to get only “contacts” document of the database, and we extract the [_id,name] and the location data as key/value peers for the source documents. Note that we test that the document contains the fields to be returned (otherwise CouchDB shuts off its indexing).

Test the Map function with the [Run] button. You should see the result list with id/name peers on the left and longitude/latitude data on the right.

Now save the temporary view with the [Save as] button, in the design document “contacts” using the

view name “coordinates”. You can now access the view with following URI:

```
http://localhost:5984/_design/contacts/_views/coordinates
```

For example, using CURL:

```
$ curl -X GET
http://localhost:5984/jsontest/_design/contacts/_view/coordinates
{"total_rows":3,"offset":0,"rows":[
{"id":"1003","key":["1003","Philip Desmond"],"value":{"longitude":-
95.118758999999997172,"latitude":38.352428000000003294}},
{"id":"1004","key":["1004","Mike Kobain"],"value":{"longitude":-
95.258335000000002424,"latitude":39.86075199999997963}},
{"id":"1005","key":["1005","Ted Fitzpatrick"],"value":{"longitude":-
94.858368999999996163,"latitude":38.684463000000000932}}
]}
```

To get coordinates for a given contact, add a query string on the [id/name] key (note that you must quote the URI in this case to avoid shell script interpretation):

```
$ curl -X GET
"http://localhost:5984/jsontest/_design/contacts/_view/coordinates?
key=\[\"1005\",\"Ted%20Fitzpatrick\"\]"
{"total_rows":3,"offset":2,"rows":[
{"id":"1005","key":["1005","Ted Fitzpatrick"],"value":{"longitude":-
94.858368999999996163,"latitude":38.684463000000000932}}
]}
```

The important things to keep in mind are that map functions give you an opportunity to sort your data using any key you choose, and that CouchDB’s design is focused on providing fast, efficient access to data within a range of keys.

We will not implement a reduce function in the lab.

For an introduction to CouchDB views, see http://docs.couchdb.org/en/1.6.1/couchapp/views/intro.html

9.5. Access CouchDB data from Genero

Before starting with Genero program, we need to create a CouchDB view to select only the contacts documents of the database. This will be done by filtering on the type field. Go to FUTON and create the following temporary view:

```
function(doc) {
  if (doc.type == "contacts") {
      emit(doc._id, doc);
  }
}
```

This map function returns a key/value list with doc ids as keys and the corresponding contact record as values.

Run the view to see if it works and save it in the “contacts” design doc, as view “all”, to be accessed from the jsontest/_design/contacts/_view/all URI. This is the URI we will use to load the contact list in the Genero program (try with CURL to check/understand the JSON structure):

```
$ curl -X GET
http://127.0.0.1:5984/jsontest/_design/contacts/_view/all | jsonlint
--format
{
  "offset" : 0,
  "rows" : [
      {
        "id" : "1002",
        "key" : "1002",
        "value" : {
            "address" : "23 Big Tod St.",
            "birth" : "1989-03-12T02:12:00Z",
            "city" : "DE SOTO",
            "loc" : {
                "latitude" : 38.956563000000002717,
                "longitude" : -94.964713000000003262
              },
            "name" : "John Kinsley",
            "phone" : "234-333-7812",
            "type" : "contacts",
            "_id" : "1002",
            "_rev" : "1-7af927c7c55c4b80e8a42289fdb6b680"
          }
          …
   ],
  "total_rows" : 4
}
```

Source code is available in Appendix C

Note that:

The record definition contains the type and _rev fields to hold respectively the “contacts” value and current record version.

```
      TYPE t_contact RECORD
              _id STRING, -- Must be a string
              type STRING,
              name STRING,
              loc RECORD
                      longitude DECIMAL(20,10),
                      latitude  DECIMAL(20,10)
                  END RECORD,
              address STRING,
              city STRING,
              phone STRING,
              birth DATETIME YEAR TO SECOND,
              _rev STRING
          END RECORD
```

The load_contacts() function uses the _design/contacts/_view/all CouchDB view created previously, and can filter the result set from a list of ids.

```
CALL couchdb_operation(IIF(filter IS NULL,"GET","POST"),
                         SFMT("%1/_design/contacts/_view/all", COUCHDB_DATABASE),
                         NULL, filter)
                  RETURNING err, load_etag, json_data
```

When creating a new contact, we need to remove the “_rev:null” field created by LET json_obj = util.JSONObject.fromFGL(r_contact)

```
            IF new THEN
              LET method = "POST"
              LET uri = COUCHDB_DATABASE
              LET rev = NULL
              CALL json_obj.remove("_rev") -- Avoid "_rev":null
```

When delete or modifying a contact, we pass the current revision in the query string. If another client has modified the record since last fetch we will get a HTTP 409 conflict error.

```
            IF new THEN
              LET method = "POST"
              LET uri = COUCHDB_DATABASE
              LET rev = NULL
              CALL json_obj.remove("_rev") -- Avoid "_rev":null
            ELSE
              LET method = "PUT"
              LET uri = SFMT("%1/%2", COUCHDB_DATABASE, ca[x]._id)
              LET rev = SFMT("rev=%1",ca[x]._rev)
            END IF
            CALL couchdb_operation(method, uri, rev, json_obj.toString())
```

…

```
CALL couchdb_operation("DELETE",
                               SFMT("%1/%2", COUCHDB_DATABASE, ca[x]._id),
                               SFMT("rev=%1",ca[x]._rev), NULL)
```

With the default CouchDB configuration there is no authentication needed.

10. Open questions

1. What is (or will be) the standard for JSON-DBs REST-APIs?
    - JSON API http://jsonapi.org/ ?,
    - RESTHeart http://restheart.org + HAL http://stateless.co/hal_specification.html ?
    - Couch DB REST API http://docs.couchdb.org/en/latest/api/ ?
    - IBM Informix proprietary http://www-01.ibm.com/support/knowledgecenter/SSGU8G_12.1.0/com.ibm.json.doc/ids_json_051.htm ?)

11. Appendix A: Genero sample for Informix/MongoDB

Form file:

```
LAYOUT
GRID
{
<TABLE t1                                                          >
 Name           Lon.        Lat.        City    Phone   Birth Date
[c1            |c2         |c3         |c4     |c5     |c6         ]
<                                                                  >
}
END
END
ATTRIBUTES
PHANTOM FORMONLY._id;
c1 = FORMONLY.name, SCROLL;
c2 = FORMONLY.loc_longitude;
c3 = FORMONLY.loc_latitude;
PHANTOM FORMONLY.address;
c4 = FORMONLY.city, SCROLL;
c5 = FORMONLY.phone, SCROLL;
c6 = FORMONLY.birth;
END
INSTRUCTIONS
SCREEN RECORD sr(FORMONLY.*);
END
```

Program file:

```
IMPORT util
TYPE t_contact RECORD
              _id STRING,
              name STRING,
              loc RECORD
                      longitude DECIMAL(20,10),
                      latitude  DECIMAL(20,10)
                  END RECORD,
              address STRING,
              city STRING,
              phone STRING,
              birth DATETIME YEAR TO SECOND
          END RECORD
DEFINE ca DYNAMIC ARRAY OF t_contact
MAIN
    DATABASE jsontest
    CALL load_contacts(ca)
    OPEN FORM f1 FROM "contacts_list"
    DISPLAY FORM f1
    DISPLAY ARRAY ca TO sr.* ATTRIBUTES(UNBUFFERED, DOUBLECLICK=update)
        ON UPDATE
           LET int_flag = edit_contact(ca, arr_curr(), scr_line())
    END DISPLAY
END MAIN

FUNCTION load_contacts(ca)
    DEFINE ca DYNAMIC ARRAY OF t_contact
```

```
    DEFINE x INTEGER,
           jsdata STRING,
           r_contact t_contact
    DECLARE c1 CURSOR FROM
      "SELECT data::JSON::LVARCHAR FROM contacts"
      || " ORDER BY BSON_VALUE_LVARCHAR(data,'name')"
    LET x=0
    FOREACH c1 INTO jsdata
        CALL util.JSON.parse(jsdata, r_contact)
        LET ca[x:=x+1].* = r_contact.*
    END FOREACH
    FREE c1
END FUNCTION
FUNCTION edit_contact(ca, x, r)
    DEFINE ca DYNAMIC ARRAY OF t_contact, x, r INT
    DEFINE r_contact t_contact,
           json_obj util.JSONObject,
           json_rec STRING
    LET r_contact.* = ca[x].*
    INPUT r_contact.* WITHOUT DEFAULTS FROM sr[r].*
    IF NOT int_flag THEN
       TRY
           LET json_obj = util.JSONObject.fromFGL(r_contact)
           -- Convert datetime back to UTC/ISO-8601 (BDL datetime is in local time
format)
           CALL json_obj.put("birth", local_to_UTC_ISO8601(r_contact.birth) )
           PREPARE s1 FROM "UPDATE contacts"
                        || " SET data = ?::JSON::BSON"
                        || " WHERE BSON_VALUE_VARCHAR(data,\"_id\") = ?"
           LET json_rec = json_obj.toString()
           EXECUTE s1 USING json_rec, r_contact._id
           LET ca[x].* = r_contact.*
       CATCH
           ERROR "Could not update collection..."
           LET int_flag = TRUE
       END TRY
    END IF
    RETURN int_flag
END FUNCTION

FUNCTION local_to_UTC_ISO8601(dt)
    DEFINE dt DATETIME YEAR TO SECOND
    DEFINE r VARCHAR(40)
    -- First convert to UTC and STRING
    LET r = util.Datetime.toUTC(dt)
    -- Then set T and Z chars
    LET r[11,11] = "T"
    LET r[20,20] = "Z"
    RETURN r
END FUNCTION
```

12. Appendix B: Genero sample for native MongoDB

Form file:

```
LAYOUT
GRID
{
<GROUP g1                                                          >
 REST query string:[qs                              ][b1           ]
 Query ETag:       [et                                           ]
<                                                                  >
<TABLE t1                                                          >
 Name      Lon.        Lat.        City    Phone   Birth   ETag
[c1       |c2         |c3         |c4     |c5     |c6     |c7       ]
<                                                                  >
[                          :b2           :b3           :b4         ]
}
END
END
ATTRIBUTES
GROUP g1: group1, TEXT="Query";
EDIT qs = FORMONLY.query, SCROLL;
EDIT et = FORMONLY.etag, SCROLL, NOENTRY;
BUTTON b1: load, TEXT="Load";
PHANTOM FORMONLY._id;
c1 = FORMONLY.name, SCROLL;
c2 = FORMONLY.loc_longitude;
c3 = FORMONLY.loc_latitude;
PHANTOM FORMONLY.address;
c4 = FORMONLY.city, SCROLL;
c5 = FORMONLY.phone, SCROLL;
c6 = FORMONLY.birth;
c7 = FORMONLY._etag, SCROLL;
BUTTON b2: append, TEXT="Append";
BUTTON b3: delete, TEXT="Delete";
BUTTON b4: close, TEXT="Close";
END
INSTRUCTIONS
SCREEN RECORD sr(
   FORMONLY._id,
   FORMONLY.name,
   FORMONLY.loc_longitude,
   FORMONLY.loc_latitude,
   FORMONLY.address,
   FORMONLY.city,
   FORMONLY.phone,
   FORMONLY.birth,
   FORMONLY._etag
  );
END
```

Program file:

```
IMPORT util
IMPORT com
IMPORT xml

CONSTANT RESTHEART_PROT = "http"
CONSTANT RESTHEART_HOST = "127.0.0.1"
CONSTANT RESTHEART_PORT = 27080
CONSTANT RESTHEART_USER = "admin"
CONSTANT RESTHEART_PSWD = "fourjs"

CONSTANT MONGODB_DATABASE = "jsontest"

TYPE t_contact RECORD
               _id INTEGER,
               name STRING,
               loc RECORD
                       longitude DECIMAL(20,10),
                       latitude  DECIMAL(20,10)
                   END RECORD,
               address STRING,
               city STRING,
               phone STRING,
               birth DATETIME YEAR TO SECOND,
               _etag STRING
          END RECORD

MAIN
    DEFINE ca DYNAMIC ARRAY OF t_contact,
           query STRING,
           r SMALLINT,
           etag STRING
    LET query = 'filter={"name":{"$regex":".*"}}'
    CALL load_contacts(ca,query) RETURNING r, etag
    IF r == 1 THEN
       DISPLAY "ERROR: Run script to set _etag fields if not yet done"
       EXIT PROGRAM
    END IF
    OPEN FORM f1 FROM "contacts_list"
    DISPLAY FORM f1
    DIALOG ATTRIBUTES(UNBUFFERED)
        INPUT BY NAME query, etag ATTRIBUTES(WITHOUT DEFAULTS)
        ON ACTION load
           CALL load_contacts(ca,query) RETURNING r, etag
        END INPUT
        DISPLAY ARRAY ca TO sr.* ATTRIBUTES(DOUBLECLICK=update)
        ON APPEND
           LET int_flag = edit_contact(TRUE, etag, ca, arr_curr(), scr_line())
        ON UPDATE
           LET int_flag = edit_contact(FALSE, etag, ca, arr_curr(), scr_line())
        ON DELETE
           LET int_flag = delete_contact(ca, arr_curr())
        END DISPLAY
        ON ACTION close
           EXIT DIALOG
    END DIALOG
```

```
END MAIN

FUNCTION restheart_operation(method, uri_path, query, etag, data)
    DEFINE method, uri_path, query, etag, data STRING
    DEFINE uri, err, result STRING,
           req com.HTTPRequest,
           resp com.HTTPResponse,
           res_etag STRING,
           code INTEGER
    LET uri = SFMT("%1://%2:%3/%4",
                   RESTHEART_PROT,
                   RESTHEART_HOST,
                   RESTHEART_PORT,
                   uri_path
                  )
    IF query IS NOT NULL THEN
       LET uri = uri || "?" || query
    END IF
    TRY
        LET req = com.HTTPRequest.Create(uri)
        CALL req.setConnectionTimeout(10)
        CALL req.setTimeout(60)
        CALL req.setCharset("UTF-8")
        CALL req.setAuthentication(RESTHEART_USER,RESTHEART_PSWD,"Basic",NULL)
        CALL req.setMethod(method)
        IF etag IS NOT NULL THEN
           CALL req.setHeader("If-Match",etag) -- Mandatory for updates.
        END IF
        IF data IS NULL THEN
           CALL req.doRequest()
        ELSE
           CALL req.setHeader("Content-Type","application/json")
           CALL req.doTextRequest(data)
        END IF
        LET resp=req.getResponse()
        LET code = resp.getStatusCode()
        IF code>=200 AND code<=299 THEN
           LET result = resp.getTextResponse()
           LET res_etag = resp.getHeader("ETag")
           LET err = NULL
        ELSE
           LET err = SFMT("(%1) : HTTP request status description: %2 ",
                          code, resp.getStatusDescription())
        END IF
    CATCH
        LET err = SFMT("HTTP request error: STATUS=%1 (%2)",STATUS,SQLCA.SQLERRM)
    END TRY
    RETURN err, res_etag, result
END FUNCTION

FUNCTION load_contacts(ca,query)
    DEFINE ca DYNAMIC ARRAY OF t_contact,
           query STRING
    DEFINE err, json_data STRING,
           json_obj util.JSONObject,
           json_emb util.JSONObject,
           json_arr util.JSONArray,
```

```
           json_rec util.JSONObject,
           json_etag util.JSONObject,
           x, r INTEGER,
           load_etag STRING
    CALL restheart_operation("GET", SFMT("%1/contacts", MONGODB_DATABASE),
                             query, NULL, NULL)
                  RETURNING err, load_etag, json_data
    IF err IS NOT NULL THEN
       ERROR err
       RETURN -1, NULL
    END IF
    LET r = 0
    CALL ca.clear()
    TRY
       LET json_obj = util.JSONObject.parse(json_data)
       IF json_obj.has("_embedded") THEN
          LET json_emb = json_obj.get("_embedded")
          LET json_arr = json_emb.get("rh:doc")
          FOR x=1 TO json_arr.getLength()
              LET json_rec = json_arr.get(x)
              CALL json_rec.toFGL(ca[x])
              IF json_rec.has("_etag") THEN
                 LET json_etag = json_rec.get("_etag")
                 LET ca[x]._etag = json_etag.get("$oid")
              ELSE
                 LET r = 1
              END IF
          END FOR
       ELSE
          MESSAGE "No record found..."
       END IF
    CATCH
       ERROR "Could not extract JSON information."
       RETURN -1, NULL
    END TRY
    RETURN r, load_etag
END FUNCTION

FUNCTION contact_uri(_id)
    DEFINE _id INTEGER
    RETURN SFMT("%1/contacts/%2?id_type=NUMBER", MONGODB_DATABASE, _id)
END FUNCTION

FUNCTION my_uuid()
    DEFINE v VARCHAR(30), id INTEGER
    LET v = CURRENT FRACTION TO FRACTION(5)
    LET v = v[2,5] -- remove dot
    LET id = (10000 + v)
    RETURN id
END FUNCTION

FUNCTION edit_contact(new, get_etag, ca, x, r)
    DEFINE new BOOLEAN,
           get_etag STRING,
           ca DYNAMIC ARRAY OF t_contact,
           x, r INT
    DEFINE r_contact t_contact,
```

```
            json_obj util.JSONObject,
            json_rec STRING,
            method, etag STRING,
            err, new_etag, result STRING
    IF new THEN
       LET r_contact._id = my_uuid()
       LET r_contact.name = "New contact"
    ELSE
       LET r_contact.* = ca[x].*
    END IF
    INPUT r_contact.* WITHOUT DEFAULTS FROM sr[r].*
    IF NOT int_flag THEN
       TRY
           LET json_obj = util.JSONObject.fromFGL(r_contact)
           CALL json_obj.put("birth", local_to_UTC_ISO8601(r_contact.birth) )
           IF new THEN
              LET method = "POST"
              LET etag = get_etag
           ELSE
              LET method = "PUT"
              LET etag = ca[x]._etag
           END IF
           LET json_rec = json_obj.toString()
           CALL restheart_operation(method, contact_uri(ca[x]._id), NULL,
                                    etag, json_rec)
                    RETURNING err, new_etag, result
           IF err IS NULL THEN
              LET ca[x].* = r_contact.*
              LET ca[x]._etag = new_etag
           ELSE
              ERROR "HTTP Request failed: ", err
              LET int_flag = TRUE
           END IF
       CATCH
           ERROR "Could not update contact..."
           LET int_flag = TRUE
       END TRY
    END IF
    RETURN int_flag
END FUNCTION

FUNCTION delete_contact(ca, x)
    DEFINE ca DYNAMIC ARRAY OF t_contact, x INT
    DEFINE err, new_etag, result STRING
    TRY
        CALL restheart_operation("DELETE", contact_uri(ca[x]._id), NULL,
ca[x]._etag, NULL)
                         RETURNING err, new_etag, result
        IF err IS NULL THEN
           LET int_flag = FALSE
        ELSE
           ERROR "DELETE failed: ", err
           LET int_flag = TRUE
        END IF
    CATCH
        ERROR "Could not delete contact..."
        LET int_flag = TRUE
```

```
    END TRY
    RETURN int_flag
END FUNCTION

FUNCTION local_to_UTC_ISO8601(dt)
    DEFINE dt DATETIME YEAR TO SECOND
    DEFINE r VARCHAR(40)
    IF dt IS NULL THEN RETURN NULL END IF
    LET r = util.Datetime.toUTC(dt)
    LET r[11,11] = "T"
    LET r[20,20] = "Z"
    RETURN r
END FUNCTION
```

13. Appendix C: Genero sample for CouchDB

Form file:

```
LAYOUT
GRID
{
<GROUP g1                                                    >
 Key filter: [qs                                  ][b1       ]
 Query ETag:       [et                                      ]
<                                                            >
<TABLE t1                                                    >
 Name      Lon.        Lat.        City    Phone   Birth   Revision
[c1       |c2         |c3         |c4     |c5     |c6     |c7        ]
<                                                            >
[                          :b2           :b3           :b4         ]
}
END
END
ATTRIBUTES
GROUP g1: group1, TEXT="Query";
EDIT qs = FORMONLY.filter, SCROLL;
EDIT et = FORMONLY.etag, SCROLL, NOENTRY;
BUTTON b1: load, TEXT="Load";
PHANTOM FORMONLY._id;
PHANTOM FORMONLY.type;
c1 = FORMONLY.name, SCROLL;
c2 = FORMONLY.loc_longitude;
c3 = FORMONLY.loc_latitude;
PHANTOM FORMONLY.address;
c4 = FORMONLY.city, SCROLL;
c5 = FORMONLY.phone, SCROLL;
c6 = FORMONLY.birth;
c7 = FORMONLY._rev, SCROLL;
BUTTON b2: append, TEXT="Append";
BUTTON b3: delete, TEXT="Delete";
BUTTON b4: close, TEXT="Close";
END
INSTRUCTIONS
SCREEN RECORD sr(
   FORMONLY._id,
   FORMONLY.type,
   FORMONLY.name,
   FORMONLY.loc_longitude,
   FORMONLY.loc_latitude,
   FORMONLY.address,
   FORMONLY.city,
   FORMONLY.phone,
   FORMONLY.birth,
   FORMONLY._rev
  );
END
```

Program file:

```
IMPORT util
IMPORT com
IMPORT xml

CONSTANT COUCHDB_PROT = "http"
CONSTANT COUCHDB_HOST = "127.0.0.1"
CONSTANT COUCHDB_PORT = 5984
-- No authentication required by default
--CONSTANT COUCHDB_USER = "admin"
--CONSTANT COUCHDB_PSWD = "fourjs"

CONSTANT COUCHDB_DATABASE = "jsontest"

TYPE t_contact RECORD
              _id STRING, -- Must be a string
              type STRING,
              name STRING,
              loc RECORD
                      longitude DECIMAL(20,10),
                      latitude  DECIMAL(20,10)
                  END RECORD,
              address STRING,
              city STRING,
              phone STRING,
              birth DATETIME YEAR TO SECOND,
              _rev STRING
          END RECORD

MAIN
    DEFINE ca DYNAMIC ARRAY OF t_contact,
           filter STRING,
           r SMALLINT,
           etag STRING
    LET filter = '{"keys":["1001","1002","1003"]}'
    CALL load_contacts(ca,filter) RETURNING r, etag
    IF r == 1 THEN
       DISPLAY "ERROR: Run script to set _etag fields if not yet done"
       EXIT PROGRAM
    END IF
    OPEN FORM f1 FROM "contacts_list"
    DISPLAY FORM f1
    DIALOG ATTRIBUTES(UNBUFFERED)
        INPUT BY NAME filter, etag ATTRIBUTES(WITHOUT DEFAULTS)
        ON ACTION load
           CALL load_contacts(ca,filter) RETURNING r, etag
        END INPUT
        DISPLAY ARRAY ca TO sr.* ATTRIBUTES(DOUBLECLICK=update)
        ON APPEND
           LET int_flag = edit_contact(TRUE, ca, arr_curr(), scr_line())
        ON UPDATE
           LET int_flag = edit_contact(FALSE, ca, arr_curr(), scr_line())
        ON DELETE
           LET int_flag = delete_contact(ca, arr_curr())
        END DISPLAY
        ON ACTION close
```

```
            EXIT DIALOG
        END DIALOG
    END MAIN

    FUNCTION couchdb_operation(method, uri_path, query, data)
        DEFINE method, uri_path, query, data STRING
        DEFINE uri, err, result STRING,
               req com.HTTPRequest,
               resp com.HTTPResponse,
               res_etag STRING,
               code INTEGER
        LET uri = SFMT("%1://%2:%3/%4",
                       COUCHDB_PROT,
                       COUCHDB_HOST,
                       COUCHDB_PORT,
                       uri_path
                      )
        IF query IS NOT NULL THEN
           LET uri = uri || "?" || query
        END IF
        TRY
            LET req = com.HTTPRequest.Create(uri)
            CALL req.setConnectionTimeout(10)
            CALL req.setTimeout(60)
            CALL req.setCharset("UTF-8")
            --CALL req.setAuthentication(COUCHDB_USER,COUCHDB_PSWD,"Basic",NULL)
            CALL req.setMethod(method)
            IF data IS NULL THEN
               CALL req.doRequest()
            ELSE
               CALL req.setHeader("Content-Type","application/json")
               CALL req.doTextRequest(data)
            END IF
            LET resp=req.getResponse()
            LET code = resp.getStatusCode()
            IF code>=200 AND code<=299 THEN
               LET result = resp.getTextResponse()
               LET res_etag = resp.getHeader("ETag")
               LET err = NULL
            ELSE
               LET err = SFMT("(%1) : HTTP request status description: %2 ",
                              code, resp.getStatusDescription())
            END IF
        CATCH
            LET err = SFMT("HTTP request error: STATUS=%1 (%2)",STATUS,SQLCA.SQLERRM)
        END TRY
        RETURN err, res_etag, result
    END FUNCTION

    FUNCTION load_contacts(ca,filter)
        DEFINE ca DYNAMIC ARRAY OF t_contact,
               filter STRING
        DEFINE data, err, json_data STRING,
               json_obj util.JSONObject,
               json_rows util.JSONArray,
               json_row util.JSONObject,
               json_rec util.JSONObject,
```

```
        json_etag util.JSONObject,
        x, r INTEGER,
        load_etag STRING
    CALL couchdb_operation(IIF(filter IS NULL,"GET","POST"),
                           SFMT("%1/_design/contacts/_view/all", COUCHDB_DATABASE),
                           NULL, filter)
                  RETURNING err, load_etag, json_data
    IF err IS NOT NULL THEN
       ERROR err
       RETURN -1, NULL
    END IF
    LET r = 0
    CALL ca.clear()
    TRY
       LET json_obj = util.JSONObject.parse(json_data)
       IF json_obj.get("total_rows") > 0 THEN
          LET json_rows = json_obj.get("rows")
          FOR x=1 TO json_rows.getLength()
              LET json_row = json_rows.get(x)
              LET json_rec = json_row.get("value")
              CALL json_rec.toFGL(ca[x])
          END FOR
       ELSE
          MESSAGE "No record found..."
       END IF
    CATCH
       ERROR "Could not extract JSON information."
       RETURN -1, NULL
    END TRY
    -- Remove quotes around etag
    LET load_etag = load_etag.subString(2,load_etag.getLength()-1)
    RETURN r, load_etag
END FUNCTION

FUNCTION my_uuid()
    DEFINE v VARCHAR(30), id INTEGER
    LET v = CURRENT FRACTION TO FRACTION(5)
    LET v = v[2,5] -- remove dot
    LET id = (10000 + v)
    RETURN id
END FUNCTION

FUNCTION edit_contact(new, ca, x, r)
    DEFINE new BOOLEAN,
           ca DYNAMIC ARRAY OF t_contact,
           x, r INT
    DEFINE r_contact t_contact,
           json_obj util.JSONObject,
           json_res util.JSONObject,
           method, uri, rev STRING,
           err, new_etag, result STRING
    IF new THEN
       LET r_contact._id = my_uuid()
       LET r_contact.type = "contacts"
       LET r_contact.name = "New contact"
    ELSE
       LET r_contact.* = ca[x].*
```

```
    END IF
    INPUT r_contact.* WITHOUT DEFAULTS FROM sr[r].*
    IF NOT int_flag THEN
       TRY
           LET json_obj = util.JSONObject.fromFGL(r_contact)
           CALL json_obj.put("birth", local_to_UTC_ISO8601(r_contact.birth))
           IF new THEN
              LET method = "POST"
              LET uri = COUCHDB_DATABASE
              LET rev = NULL
              CALL json_obj.remove("_rev") -- Avoid "_rev":null
           ELSE
              LET method = "PUT"
              LET uri = SFMT("%1/%2", COUCHDB_DATABASE, ca[x]._id)
              LET rev = SFMT("rev=%1",ca[x]._rev)
           END IF
           CALL couchdb_operation(method, uri, rev, json_obj.toString())
                    RETURNING err, new_etag, result
           IF err IS NULL THEN
              LET ca[x].* = r_contact.*
              LET json_res = util.JSONObject.parse(result)
              LET ca[x]._rev = json_res.get("rev")
           ELSE
              ERROR "HTTP Request failed: ", err
              LET int_flag = TRUE
           END IF
       CATCH
           ERROR "Could not update contact..."
           LET int_flag = TRUE
       END TRY
    END IF
    RETURN int_flag
END FUNCTION

FUNCTION delete_contact(ca, x)
    DEFINE ca DYNAMIC ARRAY OF t_contact, x INT
    DEFINE err, new_etag, result STRING
    TRY
        CALL couchdb_operation("DELETE",
                               SFMT("%1/%2", COUCHDB_DATABASE, ca[x]._id),
                               SFMT("rev=%1",ca[x]._rev), NULL)
                     RETURNING err, new_etag, result
        IF err IS NULL THEN
           LET int_flag = FALSE
        ELSE
           ERROR "DELETE failed: ", err
           LET int_flag = TRUE
        END IF
    CATCH
        ERROR "Could not delete contact..."
        LET int_flag = TRUE
    END TRY
    RETURN int_flag
END FUNCTION

FUNCTION local_to_UTC_ISO8601(dt)
    DEFINE dt DATETIME YEAR TO SECOND
```

```
    DEFINE r VARCHAR(40)
    IF dt IS NULL THEN RETURN NULL END IF
    LET r = util.Datetime.toUTC(dt)
    LET r[11,11] = "T"
    LET r[20,20] = "Z"
    RETURN r
END FUNCTION
```