Mogu Redis Query API

Abstract.
Decoupling the application instance (Mogu class instantiation) and allowing the use of separate Redis instances necessitate a complete overhaul of how Redis queries are handled. This document outlines this new, restructured API.

Table of Contents

1.0 Configuring the Redis Prefixes....1
  1.1 Host Configuration....2
  1.2 Port Configuration....2
  1.3 Database Number....2
  1.4 Redis AUTH Command....3
2.0 The Static Context Map....4
3.0 The ContextQuery Class....4
  3.1 Creating a ContextQuery....4
  3.2 Adding Commands to the Query List....4
  3.3 Retrieving Responses from the QuerySet....5
    3.3.1 The Execute Method....6
    3.3.2 The YieldResponse Method....7
4.0 Example Creating a Widget....8

1.0 Configuring the Redis Prefixes

Mogu operates by labeling prefixes within the Redis database, each of these (sometimes called a root keyspace), houses the information it describes. Because Mogu shall no longer assume that Redis will be running with default configurations on the same server as the application is being run, it has become necessary to include a configuration file for these prefixes. In this manner, Mogu will allow each prefix to live in a separate Redis instance.

When building Mogu, one may use:

```
DBCONFDIR=/path/to/config
```

in order to customize the location of this file. Otherwise, it shall default to:

```
/usr/share/Mogu/
```

The filename shall be: 'dbconfig.conf'

The contents of the file shall default to the following:

```
@meta
     host:      localhost
     port:      6379
     number:     0

@user
     host:       localhost
     port:       6379
     number:     1

@group
     host:       localhost
     port:       6379
     number:     2

@widgets
     host:       localhost
     port:       6379
     number:     3

@templates
     host:       localhost
     port:       6379
     number:     3

@validators
     host:       localhost
     port:       6379
     number:     4

@policies
     host:       localhost
     port:       6379
     number:     4
```

Therefore by default, Mogu's different parts shall be separated into 5 different databases on the local instance, in order to protect the data. This way, should one database be corrupted or flushed, the others can remain in tact.

1.1 Host Configuration

The host keyword shall accept as its argument the word 'localhost', or an IP address of an offsite host.

1.2 Port Configuration

The port keyword shall accept as its argument only a valid integer representing a port on which the Redis instance at 'host' is listening.

1.3 Database Number

This must be a number less than the maximum defined number of Redis databases within that instance. Redis defaults to eight by default, but more can be set by updating the redis.conf file.

1.4 Redis AUTH Command

Because the redis-py module does not currently provide support for the AUTH command, Mogu cannot currently use this. This is unfortunate and an admitted flaw, but shall be dealt with as soon as that support is provided.

2.0 The Static Context Map

The context map is static object that is created when a Mogu application is started. The map is populated with the algorithm outlined in /Redis/DatabaseConfigReader.h.

The map is declared as:

```
      std::unordered_map <Prefix, std::shared_ptr <Context>> contextMap
```

and lives in the Application namespace. All users of the application shall thus be using the same contextMap instance. This prevents the overhead from having to assemble the map when each user connects or, worse, whenever a query must be made.

If the context map is not complete, the Mogu application simply will not start, and shall present an error message to the system administrator who attempted to start it. Complete means that all prefixes are configured fully (each with a 'host','port', and 'number' parameter).

3.0 The ContextQuery Class

The ContextQuery class creates a connection to the necessary database, and allows pipelining of a list of commands to be sent to the Redis instance.

3.1 Creating a ContextQuery

The ContextQuery class extends the QuerySet class, and provides an easy abstraction to instantiate a QuerySet object by using the enumerated prefix of the data.

```
// Will connect to the correct database configured for reading
// widget constructor information.

Redis::ContextQuery querySet = Redis::ContextQuery(Prefix::widgets);
```

3.2 Adding Commands to the Query List

The ContextQuery uses the hiredis library's standard method of creating Redis commands. It should be noted that all ContextQuery commands must take an identifier as their first argument. Since all context queries will necessarily interact with some member of that keyspace.

The query will automatically prepend the requisite prefix to the identifier, deprecating the need for Mogu's programmers to do this manually.

To create a Query, you must create a shared pointer to a Query object.

```
      auto query = std::make_shared <Query>(new Query(“hget %s %d”, “widgetname”,
MoguSyntax::type));
```

```
     querySet.appendQuery(query);
```

The above example is equivalent to calling “hget widgets.widgetname type” from within a Redis interactive interface (using the contextQuery defined in 3.1 Creating a ContextQuery).

By default, Redis will yield the responses to each query, including those that write to the datbase. This may not always be desired behavior, so the QuerySet class comes with built-in flags for handling the response.

The flags are:

- IGNORE_RESPONSE
  - After the command is executed, will continue executing commands in the queue until either
  - there are no commands left or a command's response is required.
- REQUIRE_STRING
  - If the response is not ignored, requires that the response be a string. If the response is not a string, the empty string is returned.
- REQUIRE_INT
  - If the response is not ignored, requires that the response to this query be an integer. If it is not by default an integer, it will try to coerce the string into integer format. If this is not possible, the number 0 will be returned.

These flags may be turned on or off by passing them when you append the query to the ContextQuery instance. Using the above example, we know that a widget's type shall always be returned as an integer. We can add an extra layer of safety to our application by using the following instead:

```
     auto query = std::make_shared <Query> (new Query(“hget %s %d”, “widgetname”,
MoguSyntax::type));
     querySet.appendQuery(query, REQUIRE_INT);
```

Note that you may use the '|' operator to turn on multiple flags, however none of the above flags will do anything if one of the others is turned on. Undesired behavior will result.

You may add commands to the query list at any time, before or after you begin retrieving their responses. This is useful especially when another query relies on a previous response. More on this shall be covered below.

3.3 Retrieving Responses from the QuerySet

All responses of the QuerySet are stateful, meaning a string response shall be stored until another *string* response is received; the same goes for integers and arrays.

3.3.1 The Execute Method

You may use two different methods to obtain responses. The first method is by using the execute() method, which will run all queries. This can be useful if you are only writing to the database, not reading from it, and don't care what the database has to say about these attempts.

```
      querySet.execute();
```

This will exhaust the queue of queries, and the last found of each *type* of response shall be preserved, and can be retrieved fro the querySet. You may also effectively retrieve the very last response gotten from the database.

Note that responses shall only be saved IF the 'IGNORE_RESPONSE' flag is not set.

Let's look at the following situations:

```
hset foo bar 123  =>    IGNORE_RESPONSE
hset foo baz 456  =>    IGNORE_RESPONSE
hset foo bop 789
hincr foo bar
```

When executing these commands in order, only the third and fourth responses shall be preserved. 'hset foo bop' will return either 1 (if the write was successfully made and did not replace any other data value), or 0 (if the write was unsuccessful or replaced another data value).

'hincr foo bar' will return `124` as a string, since all values in redis are inherently stored as strings.

Therefore, if the above were staged in an actual QuerySet instance:

```
      querySet.execute();
      querySet.replyInt();          // returns result of 'hset foo bop 789'
      querySet.replyString();       // returns “124”
```

If instead, we had set the following flags:

```
hset foo bar 123  =>    IGNORE_RESPONSE
hset foo baz 456  =>    IGNORE_RESPONSE
hset foo bop 789
hincr foo bar     =>    REQUIRE_INT
```

The 'hset foo bop' command response (1 or 0) will be overwritten by the number 124 (since require_int will coerce the string “124” into integer format).

Therefore:

```
      querySet.execute();
      int foobar = querySet.replyInt(); // foobar == 124
```

3.3.2 The YieldResponse Method

If more interaction is to take place between the application and the database, the yield method will undoubtedly be desired instead. Calling the 'yieldResponse' member function shall execute commands in the queue until a command is found that does *not* have the 'IGNORE_RESPONSE' flag set. It shall then return the result of that response.

YieldResponse is a *templated* function, and it can take one of four data types as its template argument:

- int
- std::string
- std::vector <int>
  - If a multi bulk reply is gotten from Redis, and the first element of this reply is an integer,
  - all responses shall be in integral format. This will be *extremely* rare.
- std::vector <std::string>
  - If a multi bulk reply is gotten from Redis, and the first element of this reply is a string,
  - all responses shall be in string format.

```
      int result = querySet.yieldResponse <int>();
      std::string result = querySet.yieldResponse <std::string> ();
      auto result querySet.yieldResponse <std::vector <int>>();
      auto result querySet.yieldResponse <std::vector <std::string>>();
```

Let's take a look at the following queue of commands and their flags:

```
hgetall widgetdata
incr somedatapoint      =>    IGNORE_RESPONSE
lrange somelist 0 4
get a_name              =>    REQUIRE_STRING
```

For the sake of the example, let's pretend that we want to get the contents of the 'text' field from 'widgetdata' and see whether or not that matches the text stored in 'a_name', and whether or not the same name is located somewhere in 'somelist'.

Assuming we've appended all of those queries and their flags to the querySet object, we can do the following:

```
      auto widgetdata_response = querySet.yieldResponse <std::vector
<std::string>>();
      auto somelist_response = querySet.yieldResponse <std::vector
<std::string>>();
      auto name = querySet.yieldResponse <std::string>();
```

In the above example, 'incr somedatapoint' happens at the call of the second 'yieldResponse'. However, since the 'IGNORE_RESPONSE' flag was set, that query was executed silently exactly when it was supposed to.

Both 'hgetall' and 'lrange' commands will yield a vector of strings (in the 'hgetall' example, the vector will be in the form of [key0,val0,key1,val1,key2,val2...]).

4.0 Example Creating a Widget

Putting together all of the above, we could do something like this:

```
      Redis::ContextQuery querySet(Prefix::widgets);

      querySet.appendQuery(
            std::make_shared<Query>(new Query("hget %s %d", widget_name,
MoguSyntax::type), REQUIRE_INTEGER);

      int widget_type = querySet.yieldResponse <int>();

      if (widget_type == (int) MoguSyntax::image)
      {
            querySet.appendQuery(
                  std::make_shared<Query>(new Query("hget %s %d", widget_name,
MoguSyntax::source), REQUIRE_STRING);

            querySet.appendQuery(
                  std::make_shared<Query>(new Query("hget %s %d", widget_name,
Mogusyntax::text), REQUIRE_STRING);

            querySet.appendQuery(
                  std::make_shared<Query>(new Query("llen %s.events", widget_name),
REQUIRE_INTEGER);

            auto image_src = querySet.yieldResponse<std::string>();
            auto image_alt = querySet.yieldResponse<std::string>();
            auto num_events = querySet.yieldResponse<int>();

            if (num_events > 0)
            {
                  querySet.appendQuery("lrange %s.events %d %d", widget_name, 0,
num_events));

                  auto event_triggers = querySet.yieldResponse
<std::vector<std::string>>();
            }
            //... and so on...
      }
```