Support Portal > Requests

How up Setup Redis to use RESP3

Ticket details

Jose Perez Saborio
9 hours ago

Hi Alberto,

Thank you for reaching out with your questions about RESP3 in Redis. I'm happy to clarify how Lua scripting and a JavaScript client behave under RESP3, especially in Redis 7+ and 8+. I'll address each of your points in detail below.

Returning Tables from Lua Scripts in RESP3 vs JSON Encoding

You observed that when you **manually construct a Lua table** (e.g. `{ name = "iong_dev", status = "active", score = 98 }`) and return it, the JavaScript client didn't directly get an object unless you encoded it to JSON. This is because by default, a Lua table with arbitrary string keys isn't automatically returned as a RESP3 map – it gets treated like a list (or even an empty array if there are no numeric indices) under the default conversion rules. In RESP2 mode, any **associative table keys are simply omitted** from the replyredis.io, and without special handling, the same happens under RESP3 unless you use the proper format.

How to return a Lua table as a map/object in RESP3: Redis provides a way to return a native map type to the client. The trick is to wrap your table in a `map` field. For example, instead of returning the table directly, do:

```
return { map = { name = "iong_dev", status = "active", score = 98 } }
```

When your connection is in RESP3 mode, returning a Lua table with a single field `map` (whose value is another table of your key/value pairs) will produce a **RESP3 map reply**. In a RESP3-aware JavaScript client, that map reply will be delivered as a dictionary/JS object without needing manual JSON parsing. In other words, using the `map` wrapper leverages the RESP3 protocol's ability to convey a hash/dictionary directly.
By contrast, if you return a Lua table without that wrapper (just raw keys as you did), Redis doesn't know to treat it as a map. It falls back to RESP2-style conversion, dropping the string keys, which is why you saw an empty array (`[]`). Your workaround of using `cjson.encode(...)` on the table and then `JSON.parse` on the client works, but it isn't the only way. With RESP3 and Redis 7+, you can return structured data directly as shown above.
On the other hand, when you **return the result of a Redis command** via `redis.call()` (for example, `return`

`redis.call('HGETALL', 'myHash')`), Redis will handle the reply serialization for you under RESP3. If RESP3 is active, commands like `HGETALL` naturally return a **map type** reply, so a supporting JavaScript client will indeed receive a proper object/dictionary automatically. In summary:

- **Custom Lua table:** Use the `map = {…}` pattern (or JSON encode) to ensure it comes back as an object.
- **Redis command result:** If RESP3 is enabled for that script, Redis will natively return complex types (maps, sets, booleans, etc.), and the client should get the right type (object, etc.) without extra work.

Now let’s address your specific follow-up questions:

Question 1: Placement of `redis.setresp(3)` in Lua Scripts

Your question: Should `redis.setresp(3)` be called at the very beginning of a function/script, or can it be called just before invoking a `redis.call`?

Answer: It’s best to call `redis.setresp(3)` **early in your script**, typically at the start of the function (before you call any Redis commands whose replies you want in RESP3 format). The `redis.setresp(3)` call switches the reply format of *subsequent* `redis.call()` calls to RESP3 within that script’s execution. If you call it at the top, then *all* Redis calls in that script will yield RESP3 replies (e.g. dictionaries for hash results, true/false for boolean replies, etc.).

In practice, calling it “just before” a particular `redis.call` has the same effect for that one call, but any Redis calls issued **before** `redis.setresp(3)` in the script would have used the default RESP2 behavior. So for consistency and clarity, placing it at the beginning is recommended. This ensures the script is entirely working in RESP3 mode. (It’s also how the Redis documentation demonstrates its usage in functions: they invoke `redis.setresp(3)` as the first step before calling commands like HGETALL)

A few additional notes on `redis.setresp(3)`:

- **No permanent effect:** It only affects the **context of the running script**. Each EVAL or function call starts with the default (RESP2) unless you set it. So you don’t have to worry about it changing the server or the client’s protocol setting outside the script – it’s local to that script’s execution.
- **Toggling:** In advanced cases, a script could call `redis.setresp(3)` and even switch back to `redis.setresp(2)` if needed around certain calls. But in most cases, you won’t need to toggle; you’ll call it once at the top if you want to use RESP3 features.
- **Availability:** This function became available with Redis 6.0 (when RESP3 was introduced), and it’s fully supported in Redis 7 and 8. In Redis 8.0+, RESP3 is still opt-in (not the default for connections), so using `HELLO 3` or a RESP3 client plus `redis.setresp(3)` in scripts is the way to get these richer reply types.

Question 2: HGETALL Returning “Metadata” Along with Hash Data

Your question: Calling `HGETALL` (or `redis.call('HGETALL', ...)` in a script) under RESP3 returns some metadata along with the hash fields. What are these metadata, and what do they include?

Answer: What you're seeing isn't additional data stored in the hash – it's how the RESP3 protocol represents a **map reply**, and how some clients (like the Redis CLI or the Lua runtime) display it. There are a couple of aspects to clarify:

- **RESP3 Map Type:** In RESP3, commands like HGETALL return a *map* data type instead of a flat array. This means the reply distinguishes between keys and values as pairs, rather than just a list. A RESP3-aware client library (for example, a Node.js Redis client that supports RESP3) will present that to you as a native object or dictionary (with no extra fluff). In that case, you just get your fields and values as an object, which is straightforward.

- **Redis CLI formatting:** If you are using the Redis CLI (in RESP3 mode via `HELLO 3`), it prints maps in a special way. You'll see output like:

  ```
  1# "field1" => "value1"
  2# "field2" => "value2"
  ```

  Here, the numbers with `#` (e.g. `1#`, `2#`) are **not part of your data** – they are just line indices for the CLI's output. The `"field1" => "value1"` indicates a key=>value pair in the map. So the CLI is telling you that it's a map with entries: field1 = value1, field2 = value2.

  These are not extra fields or metadata stored in Redis; it's purely how the CLI displays a map reply. In other words, the "metadata" you're seeing in CLI output is just the numbering and arrow notation for maps.

- **Lua script "map" wrapper:** When you call `redis.call('HGETALL', ...)` inside a Lua script **after** doing `redis.setresp(3)`, the return is a Lua table with a single key `map`. For example, `local res = redis.call('HGETALL', 'myhash')` might produce a Lua table like `res = { map = { field1 = "value1", field2 = "value2", ... } }`. That `map` field is essentially the Lua representation of the RESP3 map reply. If you were to return `res` from the script, Redis would convert it back to a RESP3 map for the client. The presence of `res["map"]` in Lua is not an actual Redis hash field – it's a **wrapper** indicating a map structure. This allows you to manipulate the hash data in Lua (as was done in the Redis 7 function example to remove a `_last_modified_` field from the map). But again, when the final result is sent to your JavaScript client, if it's RESP3-aware it will just see the cleaned-up object (fields and values), not the wrapper.

- **No hidden fields:** By default, Redis's HGETALL does not include any hidden metadata fields like timestamps or such. All fields in the reply correspond to actual hash fields that you've set. The only exception is if you *yourself* have added some meta field (for example, the documentation's sample function added a `_last_modified_` field to the hash to track updates – but that's an application choice). Out of the box, HGETALL returns exactly the fields in the hash. Under RESP3 it's just packaged as a map. So you don't have to worry that Redis is injecting extra keys; it isn't.

If you have any more questions or need further assistance, please don't hesitate to ask. I'm always happy to help you get the most out of Redis.

Best regards,
Jose
Customer Success Engineer
Redis

Alberto Iong
1 day ago

Hi Jose,

Your reply pretty much clear out all my doubts. As far as RESP3 is concerned:

1. if I want to construct a table to be returned by Lua script, it must be encoded with "cjson.encode()"; then "JSON.parse" has to be called in javascript client.

2. If I want to return object which is obtained by a "redis.call()", the object will be serialized automatically and my javascript client will get an object.

And lastly, please permit me to ask two more questions:

1. "redis.setresp(3)" should be called in the very beginning of a function or anywhere just before invoking "redis.call"?

2. Calling "redis.HGETALL" returns meta data along with hash data. What are these meta data includes?

Best regards,

Alberto

Jose Perez Saborio
2 days ago

Hi Alberto,

Glad the deep dive sparked more questions—let me clarify:

1. What `my_hgetall` actually returns
When you register a Redis Function and call `redis.setresp(3)`, subsequent `redis.call('HGETALL', hash)` returns a RESP3 **dictionary** reply. In Lua, you see this as a special table whose associative part lives under `res.map`:

```
redis.setresp(3)
local res = redis.call('HGETALL', hash)
-- res.map is a Lua table like { field1 = "value1", field2 = "value2", … }
res.map["_last_modified_"] = nil   -- remove the metadata return res
```

When you run:

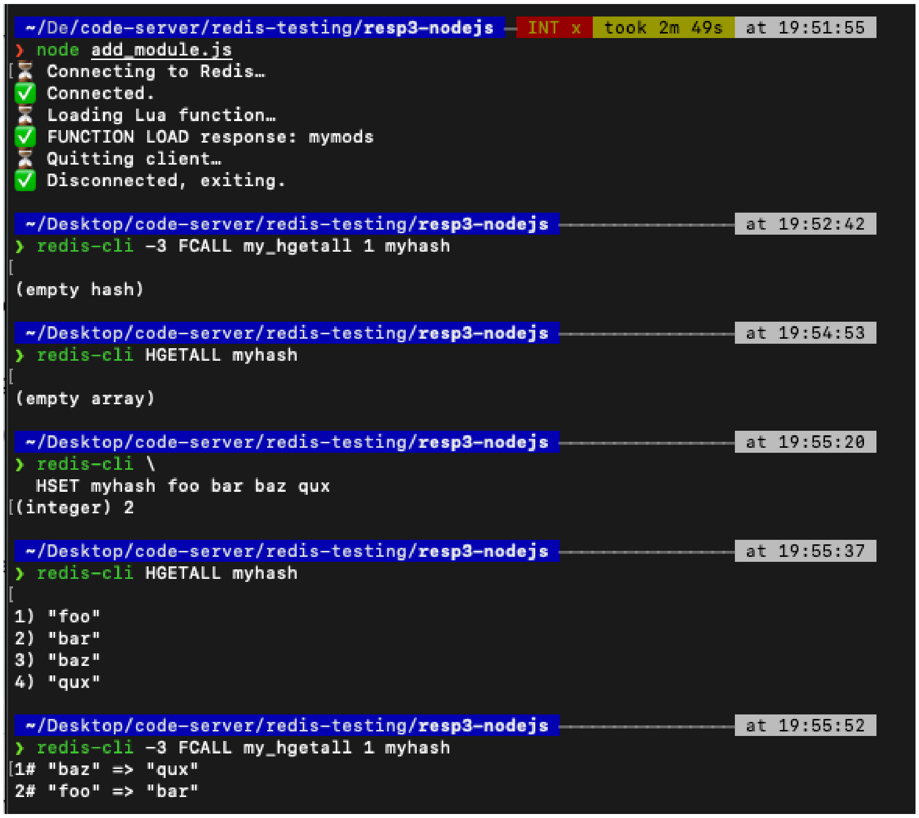
```
redis-cli -3 FCALL my_hgetall 1 myhash
```

You'll get just the remaining field/value pairs (no `_last_modified_`), see screenshot at the end.

2. Why use RESP3 rather than JSON-encode every time?
RESP3's built-in **map** type is perfect for flat key→value results—clients automatically deserialize it into objects or dictionaries without extra work. You only need to switch to `cjson.encode()` if you want to return more complex or nested Lua tables that RESP3 doesn't natively support (for example, a list of maps or deeply nested structures). For a simple filtered hash like `my_hgetall`, RESP3 alone is enough.

3. Why not just do `res["_last_modified_"] = nil`?
In this Functions API, the hash fields live in `res.map`, not directly on `res`. Setting `res["_last_modified_"] = nil` won't remove anything because `res` itself has no such key, so you must nil out the metadata inside the `map` subtable.

Hope this helps! Let me know if you'd like any more examples or have other questions. I'm attaching a few samples that you can use for testing purposes. Have a great weekend!



```
 ~/De/code-server/redis-testing/resp3-nodejs — INT x  took 2m 49s  at 19:51:55
❯ node add_module.js
[⏳ Connecting to Redis…
✅ Connected.
⏳ Loading Lua function…
✅ FUNCTION LOAD response: mymods
⏳ Quitting client…
✅ Disconnected, exiting.

 ~/Desktop/code-server/redis-testing/resp3-nodejs ----------------------  at 19:52:42
❯ redis-cli -3 FCALL my_hgetall 1 myhash
[
(empty hash)

 ~/Desktop/code-server/redis-testing/resp3-nodejs ----------------------  at 19:54:53
❯ redis-cli HGETALL myhash
[
(empty array)

 ~/Desktop/code-server/redis-testing/resp3-nodejs ----------------------  at 19:55:20
❯ redis-cli \
  HSET myhash foo bar baz qux
[(integer) 2

 ~/Desktop/code-server/redis-testing/resp3-nodejs ----------------------  at 19:55:37
❯ redis-cli HGETALL myhash
[
1) "foo"
2) "bar"
3) "baz"
4) "qux"

 ~/Desktop/code-server/redis-testing/resp3-nodejs ----------------------  at 19:55:52
❯ redis-cli -3 FCALL my_hgetall 1 myhash
[1# "baz" => "qux"
2# "foo" => "bar"
```

Best,

Jose

Customer Success

Redis

📎 my_hgetall.lua

255 Bytes · Download

📎 add_module.js

872 Bytes · Download

Alberto Iong

4 days ago

Hi Jose,

Thanks for your in depth explanation... My story began here, while peeping into [Redis Function](https://redis.io/docs/latest/develop/programmability/functions-intro/) documentation, the code example

```

local function my_hgetall(keys, args)
 redis.setresp(3)
 local hash = keys[1]
 local res = redis.call('HGETALL', hash)
 res['map']['_last_modified_'] = nil
 return res
end
```

\`\`\`

> While all of the above should be straightforward, note that the my_hgetall also calls redis.setresp(3). That means that the function expects RESP3 replies after calling redis.call(), which, unlike the default RESP2 protocol, provides dictionary (associative arrays) replies. Doing so allows the function to delete (or set to nil as is the case with Lua tables) specific fields from the reply, and in our case, the _last_modified_ field.

arousd me upmost curiosity... And according to your reply, the use of "redis.setresp(3)" doesn't make "my_hgetall" return an object automatically. To be honest, I didn't test this code yet, in this case what does "my_hgetall" returns?

And if it is necessary to use "cjson.encode()" to serialize the returned object. what's point in using RESP3? It seems to me that the sole purpose is to remove the '_last_modified_' field... and if it is so why not use this?

```
res['_last_modified_'] = nil
```

Thanks for your reply and have a nice weekend.

Alberto

Jose Perez Saborio

5 days ago

Hi Alberto,

I appreciate your patience.

I want to clarify this question we often see when working with Redis RESP3 and Lua scripting, specifically for your scenario. I'll explain why certain operations return an empty array [] when accessed from JavaScript or via redis-cli and what steps you can take if your goal is to get structured, non-empty data back.

Starting from Redis 6.0, RESP3 became the new default write protocol, and it brought more explicit distinction between "empty" collections and "null" responses:

- **Empty collections** (like a list, set, or map with zero items) are returned as an empty array (`*0\r\n` in raw RESP3, displayed as `[]` in clients).
- **Nulls/absence of value** (e.g., a missing hash field or nonexistent key) are returned as RESP3 null (`_\r\n`).

This change impacts how redis-cli and modern JavaScript/Node.js clients display and interpret results.

Redis Lua scripting supports returning Lua tables. The way a Lua table is returned depends on its structure:

- **Numerically-indexed tables (arrays like `{1, 2, 3}`)** are correctly serialized to RESP3 arrays. These display as a non-empty array in both JavaScript and redis-cli.
- **Tables with string keys (e.g. `{ name = "iong_dev", status = "active" }`)** are "object-like" from Lua's perspective, but Redis can't serialize these directly as RESP3 maps in EVAL. Instead:
  - Redis returns an empty array (`[]`) under RESP3.
  - redis-cli shows `(empty array)`.

This response is by design, not an error.

In your screenshot, you're doing something similar to this:

```
redis-cli -3 \
  -h host \
  -p 6379 \
  EVAL "redis.setresp(3); return { name = 'iong_dev', status = 'active', score = 98 }" 0
```

Returns:

```
(empty array)
```

This is expected because the returned table is not numerically indexed.

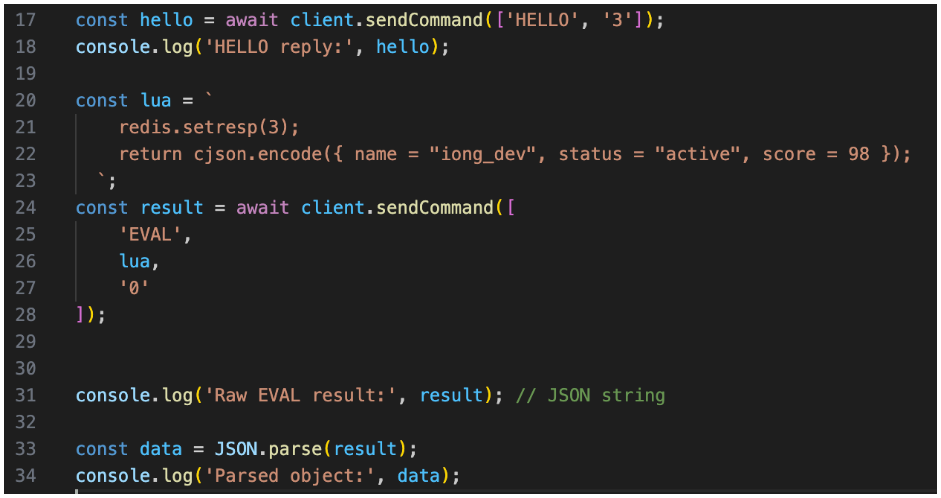
If your goal is to receive an actual map/object (not an empty array):

Solution: Encode as JSON in Lua

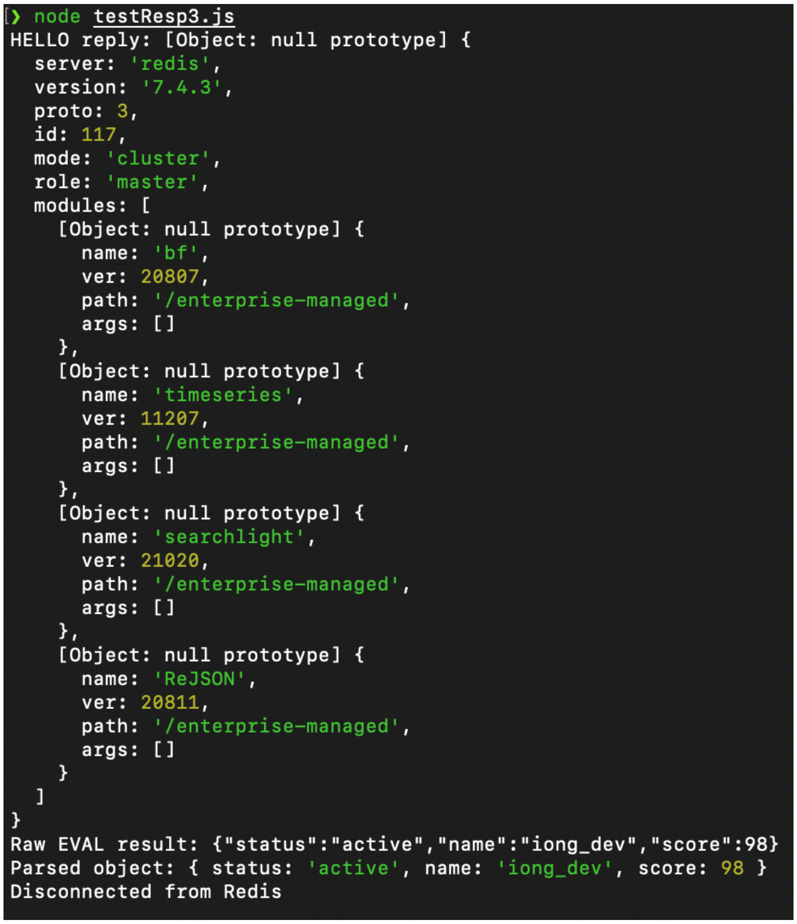
Instead of returning a Lua table directly, encode it into a JSON string within your Lua script. For example:

```
redis.setresp(3)
return cjson.encode({ name = "iong_dev", status = "active", score = 98 })
```

- **In JavaScript**, parse this string using `JSON.parse()` to get a native object:

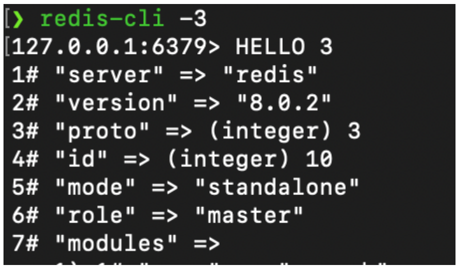


```
const hello = await client.sendCommand(['HELLO', '3']);
console.log('HELLO reply:', hello);

const lua = `
    redis.setresp(3);
    return cjson.encode({ name = "iong_dev", status = "active", score = 98 });
  `;
const result = await client.sendCommand([
    'EVAL',
    lua,
    '0'
]);


console.log('Raw EVAL result:', result); // JSON string

const data = JSON.parse(result);
console.log('Parsed object:', data);
```

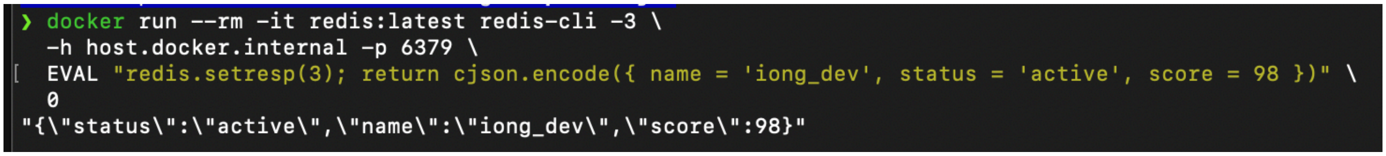


```
❯ node testResp3.js
HELLO reply: [Object: null prototype] {
  server: 'redis',
  version: '7.4.3',
  proto: 3,
  id: 117,
  mode: 'cluster',
  role: 'master',
  modules: [
    [Object: null prototype] {
      name: 'bf',
      ver: 20807,
      path: '/enterprise-managed',
      args: []
    },
    [Object: null prototype] {
      name: 'timeseries',
      ver: 11207,
      path: '/enterprise-managed',
      args: []
    },
    [Object: null prototype] {
      name: 'searchlight',
      ver: 21020,
      path: '/enterprise-managed',
      args: []
    },
    [Object: null prototype] {
      name: 'ReJSON',
      ver: 20811,
      path: '/enterprise-managed',
      args: []
    }
  ]
}
Raw EVAL result: {"status":"active","name":"iong_dev","score":98}
Parsed object: { status: 'active', name: 'iong_dev', score: 98 }
Disconnected from Redis
```

- **In redis-cli**, you'll see the raw JSON string, which you can read or process as needed.



```
❯ redis-cli -3
127.0.0.1:6379> HELLO 3
1# "server" => "redis"
2# "version" => "8.0.2"
3# "proto" => (integer) 3
4# "id" => (integer) 10
5# "mode" => "standalone"
6# "role" => "master"
7# "modules" =>
```



```
❯ docker run --rm -it redis:latest redis-cli -3 \
  -h host.docker.internal -p 6379 \
[ EVAL "redis.setresp(3); return cjson.encode({ name = 'iong_dev', status = 'active', score = 98 })" \
  0
"{\"status\":\"active\",\"name\":\"iong_dev\",\"score\":98}"
```

With RESP3 and JavaScript clients, an empty array ([]) on return—especially from { key = val, ... } Lua tables—is expected. To reliably send richer objects from Lua to JavaScript, always JSON-encode your return values and parse the result.

Feel free to let me know if you need additional examples or additional details on this topic!

Best regards,
Jose
Customer Success
Redis

Jose Perez Saborio
5 days ago

Hi Alberto,

Thank you for sharing the details and code samples regarding your attempts to use RESP3 with Redis Cloud and your local Redis 8 instance. I'm investigating this issue further to provide you with the most accurate guidance. I appreciate your patience and will update you as soon as I have more information.

If you have any additional logs or error details, please share them—they may help expedite the troubleshooting process.

Best regards,
Jose
Customer Success
Redis

Alberto Iong
6 days ago

Hi,

Just to add one more image using redis-cli, the protocol is confirmed as RESP3, but the return object is forced back to RESP2... I think...

resp3test-2.JPG

70 KB · Download

Alberto Iong

7 days ago

Hi,

This is the nodejs program i wrote to test resp3 with:

```
import { redis } from './redis/redis.js'

/*

  main

*/

const script = `

   redis.setresp(3)

   return { name = "iong_dev", status = 'active', score = 98 }

  `

await redis.connect()

await redis.sendCommand(['HELLO', '3']);

console.log(await redis.sendCommand(['HELLO']));

console.log('The result is', await redis.eval(script, 0))

await redis.close();

process.exit(0)
```

and the output is attached. I test with my Redis Cloud account and local instance on Redis 8.

resp3test.JPG

60 KB · Download

Jose Perez Saborio

7 days ago

Hi Alberto,

My name is Jose from the Redis Customer Success team—thanks for reaching out!

Please let me know whether you use Redis Cloud or self-managed Redis Software to help you better. Additionally, could you share more details about your issues when setting up RESP3? Any error messages, stack traces, or logs from your application would be beneficial. This information will allow us to provide more targeted guidance for your use case.

I'm looking forward to hearing back from you!

Best regards,
Jose
Customer Success
Redis

Alberto Iong
7 days ago

This is the nodejs program i wrote to test resp3 with:

~~~
```

import { redis } from './redis/redis.js'

/*

  main

*/

const script = `

   redis.setresp(3)

   return { name = "iong_dev", status = 'active', score = 98 }

 `

await redis.connect()

await redis.sendCommand(['HELLO', '3']);

console.log(await redis.sendCommand(['HELLO']));

console.log('The result is', await redis.eval(script, 0))

await redis.close();
~~~

```
process.exit(0)
```

and the output is attached.

resp3test.JPG

60 KB · Download

Parag Marjiwe

7 days ago

Hi Alberto,

Thank you for reaching out to Redis support regarding queries related to setting up redis to use RESP3.

We have forwarded your request to the relevant team, who will get back to you during local business hours. Meanwhile, we appreciate your patience and understanding.

Best Regards,
Parag
The Redis Team
Blog | X | LinkedIn

Redis Support Bot Agent

7 days ago

Hello,

This is an automated response.
One of our Support Engineers will get back to you soon.

Note that, based on our policies, we will never ask for your passwords or credentials.

Regards,
Redis Support Automated Services

University ↗ Blog ↗

Alberto Iong

7 days ago

Hi hi,

I was trying to setup Redis to response in RESP3 in Redis Function and Lua Script but failed. Is there any technical guide to teach how to setup RESP3 ? The drivers setup, redis.conf ... etc

Alberto

Add to conversation

Mark as solved

Support Portal