Red Hat Data Grid 8.4

Hot Rod Node.JS Client Guide

Configure and use Hot Rod JS clients

Last Updated: 2024-03-26

Red Hat Data Grid 8.4 Hot Rod Node.JS Client Guide

Configure and use Hot Rod JS clients

Legal Notice

Abstract

Hot Rod JS clients provide asynchronous, event-driven access to Data Grid clusters for Node.js applications. Asynchronous operation results are represented with `Promise` instances, allowing clients to chain multiple invocations and centralize error handling.

Table of Contents

RED HAT DATA GRID

Data Grid is a high-performance, distributed in-memory data store.

Schemaless data structure

Flexibility to store different objects as key-value pairs.

Grid-based data storage

Designed to distribute and replicate data across clusters.

Elastic scaling

Dynamically adjust the number of nodes to meet demand without service disruption.

Data interoperability

Store, retrieve, and query data in the grid from different endpoints.

DATA GRID DOCUMENTATION

Documentation for Data Grid is available on the Red Hat customer portal.

- Data Grid 8.4 Documentation
- Data Grid 8.4 Component Details
- Supported Configurations for Data Grid 8.4
- Data Grid 8 Feature Support
- Data Grid Deprecated Features and Functionality

DATA GRID DOWNLOADS

Access the Data Grid Software Downloads on the Red Hat customer portal.

NOTE

You must have a Red Hat account to access and download Data Grid software.

MAKING OPEN SOURCE MORE INCLUSIVE

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see our CTO Chris Wright's message.

CHAPTER 1. INSTALLING AND CONFIGURING HOT ROD JS CLIENTS

Ensure your system meets requirements before installing the Hot Rod JS client. You can then configure Hot Rod JS clients to connect to Data Grid Server, use different media types for keys and values, and customize logging.

1.1. INSTALLING HOT ROD JS CLIENTS

Data Grid provides a distribution of the Hot Rod JS client that you can install via the NPM package manager.

Prerequisites

- Node.js version **12** or **14**.
- Data Grid Server 8.4.

Procedure

1. Add the Red Hat repository to your NPM configuration.
   You can use the **npm config** command or add the following to an **.npmrc** file in your project:

```
@redhat:registry=https://npm.registry.redhat.com
registry=https://registry.npmjs.org/
```

2. Install the Hot Rod JS client as follows:

```
npm install @redhat/infinispan
```

1.2. CONFIGURING DATA GRID CONNECTIONS

Configure Hot Rod JS clients to connect to Data Grid Server.

If you add multiple server addresses to the configuration, the Hot Rod JS client loops through them until it finds a node to which it can connect.

However, you only need to add one Data Grid Server address for the client to receive the entire cluster topology. If the Hot Rod JS client connects to a single server instance that is a member of a cluster, the client gets the address information for all nodes.

Because Hot Rod JS clients are topology aware, if a connection to one Data Grid Server breaks, the client retries any incomplete operations on other nodes in the cluster. Likewise, if client listener that is registered on one Data Grid Server fails or leaves the cluster, the client transparently migrates the listener registration to another node in the cluster so that it can continue receiving events.

Prerequisites

- Install the Hot Rod JS client.
- Have at least one running Data Grid Server instance.

Procedure

- Specify hostnames and ports for Data Grid Server in the client configuration.

```
var infinispan = require('infinispan');

var connected = infinispan.client(
 [{port: 11322, host: '127.0.0.1'}, {port: 11222, host: '127.0.0.1'}]
 {
   // Configure client connections with authentication and encryption here.
 }
);

connected.then(function (client) {

 var members = client.getTopologyInfo().getMembers();

 // Displays all members of the Data Grid cluster.
 console.log('Connected to: ' + JSON.stringify(members));

 return client.disconnect();

}).catch(function(error) {

 console.log("Got error: " + error.message);

});
```

1.2.1. Defining Data Grid clusters in client configuration

When you set up Data Grid clusters in separate data centers to perform cross-site replication, you can add connection details for the different sites to the client configuration.

Prerequisites

- Install the Hot Rod JS client.
- Configure Data Grid for cross-site replication.

Procedure

1. Add a **clusters** definition to your configuration.
2. Add **name** and **servers** definitions for each Data Grid cluster.

```
var connected = infinispan.client({port: 11222, host: '127.0.0.1'},
 {
  clusters: [
   {
    name: 'LON',
    servers: [{port: 11222, host: 'LON-host'}]
   },
   {
    name: 'NYC',
    servers: [{port: 11222, host: 'NYC-host1'}, {port: 11322, host: 'NYC-host2'}]
```

```
    }
  ]
});
```

1.2.2. Manually switching Data Grid clusters

Change the Data Grid cluster to which the Hot Rod JS client is connectioned.

Prerequisites

- Define Data Grid clusters in the Hot Rod JS client configuration.

Procedure

1. Call the **switchToCluster(clusterName)** method to force the client to switch to a Data Grid cluster that is defined in the client configuration.
2. Call the **switchToDefaultCluster()** method to start using the initial Data Grid cluster.

```
var connected = infinispan.client({port: 11222, host: '127.0.0.1'},
 {
  clusters: [
   {
    name: 'LON',
    servers: [{port: 11222, host: 'LON-host'}]
   },
   {
    name: 'NYC',
    servers: [{port: 11222, host: 'NYC-host1'}, {port: 11322, host: 'NYC-host2'}]
   }
  ]
 });

connected.then(function (client) {

 var switchToB = client.getTopologyInfo().switchToCluster('NYC');

 switchToB.then(function(switchSucceed) {

  if (switchSucceed) {
   ...
  }

  ...

  var switchToDefault = client.getTopologyInfo().switchToDefaultCluster();

  switchToDefault.then(function(switchSucceed) {

   if (switchSucceed) {
    ...
   }

  })
```

```
  })

});
```

1.3. CONFIGURING AUTHENTICATION

Data Grid Server uses different SASL mechanisms to authenticate Hot Rod JS client connections.

Prerequisites

- Create Data Grid users.
- Add the SASL authentication mechanism to the Hot Rod connector in your Data Grid Server configuration.

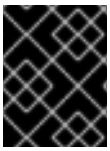
Procedure

1. Open the Hot Rod JS client configuration for editing.
2. Add an **authentication** method that sets the **enabled: true** flag.
3. Specify a value for the **saslMechanism** parameter that matches the SASL authentication mechanism for the Hot Rod connector.
4. Configure any parameters specific to the SASL authentication mechanism as appropriate.

1.3.1. SASL authentication mechanisms

Hot Rod JS clients can use the following SASL authentication mechanisms to connect to Data Grid Server.

PLAIN
Sends credentials in plain text (unencrypted) over the wire in a way that is similar to HTTP **BASIC** authentication.



IMPORTANT

To secure Data Grid credentials, you should use **PLAIN** authentication only in combination with TLS encryption.

```
var connected = infinispan.client(
  {port: 11222, host: '127.0.0.1'},
  {
    authentication: {
      enabled: true,
      saslMechanism: 'PLAIN',
      userName: 'username',
      password: 'changeme'
    }
  }
);
```

DIGEST-MD5
Uses the MD5 hashing algorithm in addition to nonces to encrypt credentials.

```
var connected = infinispan.client(
  {port: 11222, host: '127.0.0.1'},
  {
    authentication: {
      enabled: true,
      saslMechanism: 'DIGEST-MD5',
      userName: 'username',
      password: 'changeme',
      serverName: 'infinispan'
    }
  }
);
```

SCRAM
Uses salt values in addition to hashing algorithms and nonce values to encrypt credentials. Hot Rod endpoints support **SCRAM-SHA-1**, **SCRAM-SHA-256**, **SCRAM-SHA-384**, **SCRAM-SHA-512** hashing algorithms, in order of strength.

```
var connected = infinispan.client(
  {port: 11222, host: '127.0.0.1'},
  {
    authentication: {
      enabled: true,
      saslMechanism: 'SCRAM-SHA-1',
      userName: 'username',
      password: 'changeme'
    }
  }
);
```

Additional resources

- Configuring Endpoint Authentication Mechanisms

1.4. CONFIGURING ENCRYPTION

Data Grid Server can enforce different types of SSL/TLS encryption to secure Hot Rod JS client connections.

Prerequisites

- Create a trust store that Hot Rod JS clients can use to verify Data Grid Server identities.
- If you configure Data Grid Server to validate or authenticate client certificates, create a keystore as appropriate.

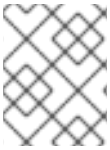
Procedure

1. Open the Hot Rod JS client configuration for editing.
2. Add an **ssl** method that sets the **enabled: true** flag.
3. Provide any other configuration specific to the type of encryption you use.

1.4.1. Encryption types

Hot Rod JS clients can use different types of encryption to negotiate secure connections with Data Grid Server.

Data Grid Server identities
For basic encryption, you can add the signing certificate, or CA bundle, for Data Grid Server certificates to your configuration as follows:



NOTE

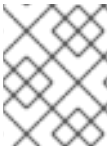
To verify certificates issued to Data Grid Server, Hot Rod JS clients require either the full certificate chain or a partial chain that starts with the Root CA.

```
var connected = infinispan.client({port: 11222, host: '127.0.0.1'},
  {
    ssl: {
      enabled: true,
      trustCerts: ['my-root-ca.crt.pem']
    }
  }
);
```

Trust stores
You can add trust stores in **PKCS12** or **PFX** format as follows:

```
var connected = infinispan.client({port: 11222, host: '127.0.0.1'},
  {
    ssl: {
      enabled: true,
      cryptoStore: {
        path: 'my-truststore.p12',
        passphrase: 'secret'
      }
    }
  }
);
```

Client certificate authentication
If you enable client certificate authentication in Data Grid Server configuration, add a keystore as in the following example:



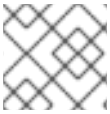
NOTE

You must configure the Hot Rod JS client with the **EXTERNAL** authentication mechanism when using client certificate authentication.

```
var connected = infinispan.client({port: 11222, host: '127.0.0.1'},
  {
    ssl: {
      enabled: true,
      trustCerts: ['my-root-ca.crt.pem'],
      clientAuth: {
```

```
        key: 'privkey.pem',
        passphrase: 'secret',
        cert:ssl 'cert.pem'
      }
    }
  }
);
```

Server Name Indication (SNI)
If you use SNI to allow Hot Rod JS clients to request Data Grid Server hostnames, set a value for the **sniHostName** parameter that matches a hostname in the Data Grid Server configuration.



NOTE

The **sniHostName** parameter defaults to **localhost**.

```
var connected = infinispan.client({port: 11222, host: '127.0.0.1'},
  {
    ssl: {
      enabled: true,
      trustCerts: ['my-root-ca.crt.pem']
      sniHostName: 'example.com'
    }
  }
);
```

TIP

Hot Rod JS clients do not allow self-signed certificates by default, which can cause issues in development or test environments where no public certificate authority (CA) key is available.

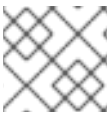
Check out the Data Grid code tutorials for an example on creating signed certificates with the Java keytool.

Additional resources

- Network interfaces and endpoints
- Encrypting Data Grid Server connections

1.5. CONFIGURING DATA FORMATS

Hot Rod JS clients can handle keys and values as native JavaScript Object Notation (JSON) objects or as String objects. By default, clients handle entries as String objects. If you want to transmit data to Data Grid Server in JSON format, then you must configure the Hot Rod JS client.



NOTE

Script operations support String key/value pairs and String parameters only.

Procedure

1. Add a **dataFormat** configuration to your client.

2. Set the data format for keys and values as appropriate with the **keyType** and **valueType** parameters.

Keys and values can have different media types. For JSON objects, specify **application/json**. For String objects, specify **text/plain** or omit the parameter to use the default.

```
var infinispan = require('infinispan');

var connected = infinispan.client(
   {port: 11222, host: '127.0.0.1'},
   {
      dataFormat : {
         keyType: 'application/json',
         valueType: 'application/json'
      }
   }
);

connected.then(function (client) {

  var clientPut = client.put({k: 'key'}, {v: 'value'});

  var clientGet = clientPut.then(
     function() { return client.get({k: 'key'}); });

  var showGet = clientGet.then(
     function(value) { console.log("get({k: 'key'})=" + JSON.stringify(value)); });

  return showGet.finally(
     function() { return client.disconnect(); });

}).catch(function(error) {

  console.log("Got error: " + error.message);

});
```

1.6. CONFIGURING LOGGING

Hot Rod JS clients use **log4js**, which you can modify by providing configuration in JSON format.

Procedure

1. Create a logging configuration in JSON format.
   For example, the following JSON configures an appender that writes TRACE level log events to file:

```
{
  "appenders": {
    "test": {
      "type": "fileSync",
      "filename": "my-log-file.log"
    }
  },
```

```
  "categories": {
    "default": {
      "appenders": ["test"],
      "level": "trace"
    }
  }
}
```

2. Add the **var log4js = require('log4js')** statement to the Hot Rod JS client configuration.

3. Specify the path to your JSON logging configuration with the **log4js.configure()** method, as in the following example:

```
var log4js = require('log4js');
log4js.configure('path/to/my-log4js.json');
```

Additional resources

- log4js
- log4js-node examples

CHAPTER 2. USING HOT ROD JS CLIENTS

Take a look at some examples for using the Hot Rod JS client with Data Grid.

2.1. HOT ROD JS CLIENT EXAMPLES

After you install and configure your Hot Rod JS client, start using it by trying out some basic cache operations before moving on to more complex interactions with Data Grid.

2.1.1. Hello world

Create a cache named "myCache" on Data Grid Server then add and retrieve an entry.

```
var infinispan = require('infinispan');

// Connect to Data Grid Server.
// Use an existing cache named "myCache".
var connected = infinispan.client(
  {port: 11222, host: '127.0.0.1'},
  {
    cacheName: 'myCache',
    clientIntelligence: 'BASIC',
    authentication: {
      enabled: true,
      saslMechanism: 'DIGEST-MD5',
      userName: 'username',
      password: 'changeme'
    }
  }
);

connected.then(function (client) {

  console.log('Connected to `myCache`');

  // Add an entry to the cache.
  var clientPut = client.put('hello', 'world');

  // Retrieve the entry you added.
  var clientGet = clientPut.then(
      function() { return client.get('hello'); });

  // Print the value of the entry.
  var showGet = clientGet.then(
      function(value) { console.log('get(hello)=' + value); });

  // Disconnect from Data Grid Server.
  return client.disconnect();

}).catch(function(error) {

  // Log any errors.
  console.log("Got error: " + error.message);

});
```

2.1.2. Working with entries and retrieving cache statistics

Add, retrieve, remove single entries and view statistics for the cache.

```
var infinispan = require('infinispan');

var connected = infinispan.client(
  {port: 11222, host: '127.0.0.1'},
  {
    cacheName: 'myCache',
    authentication: {
        enabled: true,
        saslMechanism: 'DIGEST-MD5',
        userName: 'username',
        password: 'changeme'
    }
  }
);

connected.then(function (client) {

  var clientPut = client.put('key', 'value');

  var clientGet = clientPut.then(
      function() { return client.get('key'); });

  var showGet = clientGet.then(
      function(value) { console.log('get(key)=' + value); });

  var clientRemove = showGet.then(
      function() { return client.remove('key'); });

  var showRemove = clientRemove.then(
      function(success) { console.log('remove(key)=' + success); });

  var clientStats = showRemove.then(
    function() { return client.stats(); });

  var showStats = clientStats.then(
    function(stats) {
      console.log('Number of stores: ' + stats.stores);
      console.log('Number of cache hits: ' + stats.hits);
      console.log('All statistics: ' + JSON.stringify(stats, null, " "));
    });

  return showStats.finally(
      function() { return client.disconnect(); });

}).catch(function(error) {

  console.log("Got error: " + error.message);

});
```

2.1.3. Working with multiple cache entries

Create multiple cache entries with simple recursive loops.

```
var infinispan = require('infinispan');

var connected = infinispan.client(
  {port: 11222, host: '127.0.0.1'},
  {
    cacheName: 'myCache',
    authentication: {
        enabled: true,
        saslMechanism: 'DIGEST-MD5',
        userName: 'username',
        password: 'changeme'
    }
  }
);

connected.then(function (client) {
  var data = [
    {key: 'multi1', value: 'v1'},
    {key: 'multi2', value: 'v2'},
    {key: 'multi3', value: 'v3'}];

  var clientPutAll = client.putAll(data);

  var clientGetAll = clientPutAll.then(
    function() { return client.getAll(['multi2', 'multi3']); });

  var showGetAll = clientGetAll.then(
    function(entries) {
      console.log('getAll(multi2, multi3)=%s', JSON.stringify(entries));
    }
  );

  var clientIterator = showGetAll.then(
    function() { return client.iterator(1); });

  var showIterated = clientIterator.then(
    function(it) {
      function loop(promise, fn) {
        // Simple recursive loop over the iterator's next() call.
        return promise.then(fn).then(function (entry) {
          return entry.done
            ? it.close().then(function () { return entry.value; })
            : loop(it.next(), fn);
        });
      }

      return loop(it.next(), function (entry) {
        console.log('iterator.next()=' + JSON.stringify(entry));
        return entry;
      });
    }
  );
```

```
  var clientClear = showIterated.then(
    function() { return client.clear(); });

  return clientClear.finally(
    function() { return client.disconnect(); });

}).catch(function(error) {

  console.log("Got error: " + error.message);

});
```

2.1.4. Using Async and Await constructs

Node.js provides **async** and **await** constructs that can simplify cache operations.

Single cache entries

```
const infinispan = require("infinispan");

const log4js = require('log4js');
log4js.configure('example-log4js.json');

async function test() {
  await new Promise((resolve, reject) => setTimeout(() => resolve(), 1000));
  console.log('Hello, World!');

  let client = await infinispan.client({port: 11222, host: '127.0.0.1'});
  console.log(`Connected to Infinispan dashboard data`);

  await client.put('key', 'value');

  let value = await client.get('key');
  console.log('get(key)=' + value);

  let success = await client.remove('key');
  console.log('remove(key)=' + success);

  let stats = await client.stats();
  console.log('Number of stores: ' + stats.stores);
  console.log('Number of cache hits: ' + stats.hits);
  console.log('All statistics: ' + JSON.stringify(stats, null, " "));

  await client.disconnect();
}

test();
```

Multiple cache entries

```
const infinispan = require("infinispan");

const log4js = require('log4js');
log4js.configure('example-log4js.json');
```

```
async function test() {
  let client = await infinispan.client({port: 11222, host: '127.0.0.1'});
  console.log(`Connected to Infinispan dashboard data`);

  let data = [
    {key: 'multi1', value: 'v1'},
    {key: 'multi2', value: 'v2'},
    {key: 'multi3', value: 'v3'}];

  await client.putAll(data);

  let entries = await client.getAll(['multi2', 'multi3']);
  console.log('getAll(multi2, multi3)=%s', JSON.stringify(entries));

  let iterator = await client.iterator(1);

  let entry = {done: true};

  do {
    entry = await iterator.next();
    console.log('iterator.next()=' + JSON.stringify(entry));
  } while (!entry.done);

  await iterator.close();

  await client.clear();

  await client.disconnect();
}

test();
```

2.1.5. Running server-side scripts

You can add custom scripts to Data Grid Server and then run them from Hot Rod JS clients.

Sample script

```
// mode=local,language=javascript,parameters=[k, v],datatype='text/plain; charset=utf-8'
cache.put(k, v);
cache.get(k);
```

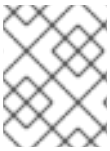
Script execution

```
var infinispan = require('infinispan');
var readFile = Promise.denodeify(require('fs').readFile);

var connected = infinispan.client(
  {port: 11222, host: '127.0.0.1'}
  {
    // Configure client connections with authentication and encryption here.
  }
);
```

```
connected.then(function (client) {

  var addScriptFile = readFile('sample-script.js').then(
    function(file) {
      return client.addScript('sample-script', file.toString());
    });

  var clientExecute = addScriptFile.then(
    function() {
      return client.execute('sample-script', {k: 'exec-key', v: 'exec-value'});
    });

  var showExecute = clientExecute.then(
    function(ret) { console.log('Script execution returned: ' + ret); });

  return showExecute.finally(
    function() { return client.disconnect(); });

}).catch(function(error) {

  console.log("Got error: " + error.message);

});
```

2.1.6. Registering event listeners

Event listeners notify Hot Rod JS clients when cache updates occur, including when entries are created, modified, removed, or expired.



NOTE

Events for entry creation and modification notify clients about keys and values. Events for entry removal and expiration notify clients about keys only.

Event listener registration

```
var infinispan = require('infinispan');

var connected = infinispan.client(
  {port: 11222, host: '127.0.0.1'}
  {
    // Configure client connections with authentication and encryption here.
  }
);

connected.then(function (client) {

    var clientAddListenerCreate = client.addListener('create', onCreate);

    var clientAddListeners = clientAddListenerCreate.then(
        function(listenerId) {
            // Associate multiple callbacks with a single client-side listener.
            // To do this, register listeners with the same listener ID.
            var clientAddListenerModify =
```

```
            client.addListener('modify', onModify, {listenerId: listenerId});

        var clientAddListenerRemove =
            client.addListener('remove', onRemove, {listenerId: listenerId});

        return Promise.all([clientAddListenerModify, clientAddListenerRemove]);
    });

var clientCreate = clientAddListeners.then(
    function() { return client.putIfAbsent('eventful', 'v0'); });

var clientModify = clientCreate.then(
    function() { return client.replace('eventful', 'v1'); });

var clientRemove = clientModify.then(
    function() { return client.remove('eventful'); });

var clientRemoveListener =
    Promise.all([clientAddListenerCreate, clientRemove]).then(
        function(values) {
            var listenerId = values[0];
            return client.removeListener(listenerId);
        });

return clientRemoveListener.finally(
    function() { return client.disconnect(); });

}).catch(function(error) {

    console.log("Got error: " + error.message);

});

function onCreate(key, version) {
    console.log('[Event] Created key: ' + key +
        ' with version: ' + JSON.stringify(version));
}

function onModify(key, version) {
    console.log('[Event] Modified key: ' + key +
        ', version after update: ' + JSON.stringify(version));
}

function onRemove(key) {
    console.log('[Event] Removed key: ' + key);
}
```

You can tune notifications from event listeners to avoid unnecessary roundtrips with the **key-value-with-previous-converter-factory** converter. This allows you to, for example, find out values associated with keys within the event instead of retrieving them afterwards.

Remote event converter

```
var infinispan = require('infinispan');

var connected = infinispan.client(
```

```
    {port: 11222, host: '127.0.0.1'}
    , {
        dataFormat : {
            keyType: 'application/json',
            valueType: 'application/json'
        }
    }
);

connected.then(function (client) {
    // Include the remote event converter to avoid unnecessary roundtrips.
    var opts = {
        converterFactory : {
            name: "key-value-with-previous-converter-factory"
        }
    };

    var clientAddListenerCreate = client.addListener('create', logEvent("Created"), opts);

    var clientAddListeners = clientAddListenerCreate.then(
        function(listenerId) {
            // Associate multiple callbacks with a single client-side listener.
            // To do this, register listeners with the same listener ID.
            var clientAddListenerModify =
                client.addListener('modify', logEvent("Modified"), {opts, listenerId: listenerId});

            var clientAddListenerRemove =
                client.addListener('remove', logEvent("Removed"), {opts, listenerId: listenerId});

            return Promise.all([clientAddListenerModify, clientAddListenerRemove]);
        });

    var clientCreate = clientAddListeners.then(
        function() { return client.putIfAbsent('converted', 'v0'); });

    var clientModify = clientCreate.then(
        function() { return client.replace('converted', 'v1'); });

    var clientRemove = clientModify.then(
        function() { return client.remove('converted'); });

    var clientRemoveListener =
        Promise.all([clientAddListenerCreate, clientRemove]).then(
            function(values) {
                var listenerId = values[0];
                return client.removeListener(listenerId);
            });

    return clientRemoveListener.finally(
        function() { return client.disconnect(); });

}).catch(function(error) {

    console.log("Got error: " + error.message);

});
```

```
function logEvent(prefix) {
   return function(event) {
      console.log(prefix + " key: " + event.key);
      console.log(prefix + " value: " + event.value);
      console.log(prefix + " previous value: " + event.prev);
   }
}
```

TIP

You can add custom converters to Data Grid Server. See the Data Grid documentation for information.

2.1.7. Using conditional operations

The Hot Rod protocol stores metadata about values in Data Grid. This metadata provides a deterministic factor that lets you perform cache operations for certain conditions. For example, if you want to replace the value of a key if the versions do not match.

Use the **getWithMetadata** method to retrieve metadata associated with the value for a key.

```
var infinispan = require('infinispan');

var connected = infinispan.client(
  {port: 11222, host: '127.0.0.1'}
  {
    // Configure client connections with authentication and encryption here.
  }
);

connected.then(function (client) {

  var clientPut = client.putIfAbsent('cond', 'v0');

  var showPut = clientPut.then(
     function(success) { console.log(':putIfAbsent(cond)=' + success); });

  var clientReplace = showPut.then(
     function() { return client.replace('cond', 'v1'); } );

  var showReplace = clientReplace.then(
     function(success) { console.log('replace(cond)=' + success); });

  var clientGetMetaForReplace = showReplace.then(
     function() { return client.getWithMetadata('cond'); });

  // Call the getWithMetadata method to retrieve the value and its metadata.
  var clientReplaceWithVersion = clientGetMetaForReplace.then(
     function(entry) {
       console.log('getWithMetadata(cond)=' + JSON.stringify(entry));
       return client.replaceWithVersion('cond', 'v2', entry.version);
     }
  );

  var showReplaceWithVersion = clientReplaceWithVersion.then(
```

```
    function(success) { console.log('replaceWithVersion(cond)=' + success); });

  var clientGetMetaForRemove = showReplaceWithVersion.then(
    function() { return client.getWithMetadata('cond'); });

  var clientRemoveWithVersion = clientGetMetaForRemove.then(
    function(entry) {
      console.log('getWithMetadata(cond)=' + JSON.stringify(entry));
      return client.removeWithVersion('cond', entry.version);
    }
  );

  var showRemoveWithVersion = clientRemoveWithVersion.then(
    function(success) { console.log('removeWithVersion(cond)=' + success)});

  return showRemoveWithVersion.finally(
    function() { return client.disconnect(); });

}).catch(function(error) {

  console.log("Got error: " + error.message);

});
```

2.1.8. Working with ephemeral data

Use the **getWithMetadata** and **size** methods expire cache entries.

```
var infinispan = require('infinispan');

var connected = infinispan.client(
  {port: 11222, host: '127.0.0.1'}
  {
    // Configure client connections with authentication and encryption here.
  }
);

connected.then(function (client) {

  var clientPutExpiry = client.put('expiry', 'value', {lifespan: '1s'});

  var clientGetMetaAndSize = clientPutExpiry.then(
    function() {
      // Compute getWithMetadata and size in parallel.
      return Promise.all([client.getWithMetadata('expiry'), client.size()]);
    });

  var showGetMetaAndSize = clientGetMetaAndSize.then(
    function(values) {
      console.log('Before expiration:');
      console.log('getWithMetadata(expiry)=' + JSON.stringify(values[0]));
      console.log('size=' + values[1]);
    });

  var clientContainsAndSize = showGetMetaAndSize.then(
```

```
    function() {
      sleepFor(1100); // Sleep to force expiration.
      return Promise.all([client.containsKey('expiry'), client.size()]);
    });

  var showContainsAndSize = clientContainsAndSize.then(
    function(values) {
      console.log('After expiration:');
      console.log('containsKey(expiry)=' + values[0]);
      console.log('size=' + values[1]);
    });

  return showContainsAndSize.finally(
    function() { return client.disconnect(); });

}).catch(function(error) {

  console.log("Got error: " + error.message);

});

function sleepFor(sleepDuration){
  var now = new Date().getTime();
  while(new Date().getTime() < now + sleepDuration){ /* Do nothing. */ }
}
```

2.1.9. Working with queries

Use the **query** method to perform queries on your caches. You must configure Hot Rod JS client to have **application/x-protostream** data format for values in your caches.

```
const infinispan = require('infinispan');
const protobuf = require('protobufjs');
// This example uses async/await paradigma
(async function () {
  // User data protobuf definition
  const cacheValueProtoDef = `package awesomepackage;
  /**
   * @TypeId(1000044)
   */
  message AwesomeUser {
      required string name = 1;
      required int64 age = 2;
      required bool isVerified =3;
  }`
  try {
    // Creating clients for two caches:
    // - ___protobuf_metadata for registering .proto file
    // - queryCache for user data
    const connectProp = { port: 11222, host: '127.0.0.1' };
    const commonOpts = {
      version: '3.0',
      authentication: {
        enabled: true,
        saslMechanism: 'DIGEST-MD5',
```

```
      userName: 'admin',
      password: 'pass'
    }
  };
  const protoMetaClientOps = {
    cacheName: '___protobuf_metadata',
    dataFormat: { keyType: "text/plain", valueType: "text/plain" }
  }
  const clientOps = {
    dataFormat: { keyType: "text/plain", valueType: "application/x-protostream" },
    cacheName: 'queryCache'
  }
  var protoMetaClient = await infinispan.client(connectProp, Object.assign(commonOpts,
protoMetaClientOps));
  var client = await infinispan.client(connectProp, Object.assign(commonOpts, clientOps));

  // Registering protobuf definition on server
  await protoMetaClient.put("awesomepackage/AwesomeUser.proto", cacheValueProtoDef);

  // Registering protobuf definition on protobufjs
  const root = protobuf.parse(cacheValueProtoDef).root;
  const AwesomeUser = root.lookupType(".awesomepackage.AwesomeUser");
  client.registerProtostreamRoot(root);
  client.registerProtostreamType(".awesomepackage.AwesomeUser", 1000044);

  // Cleanup and populating the cache
  await client.clear();
  for (let i = 0; i < 10; i++) {
    const payload = { name: "AwesomeName" + i, age: i, isVerified: (Math.random() < 0.5) };
    const message = AwesomeUser.create(payload);
    console.log("Creating entry:", message);
    await client.put(i.toString(), message)
  }
  // Run the query
  const queryStr = `select u.name,u.age from awesomepackage.AwesomeUser u where u.age<20
order by u.name asc`;
  console.log("Running query:", queryStr);
  const query = await client.query({ queryString: queryStr });
  console.log("Query result:");
  console.log(query);
 } catch (err) {
  handleError(err);
 } finally {
  if (client) {
    await client.disconnect();
  }
  if (protoMetaClient) {
    await protoMetaClient.disconnect();
  }
 }
})();

function handleError(err) {
 if (err.message.includes("'queryCache' not found")) {
  console.log('*** ERROR ***');
  console.log(`*** This example needs a cache 'queryCache' with the following config:
```

```
    {
      "local-cache": {
        "statistics": true,
        "encoding": {
        "key": {
          "media-type": "text/plain"
        },
        "value": {
          "media-type": "application/x-protostream"
}}}}`)
  } else {
    console.log(err);
  }
}
```

See Querying Data Grid caches for more information.