Using Node.js with Caché

Version 2018.1
2019-09-20

InterSystems Corporation 1 Memorial Drive Cambridge MA 02142 www.intersystems.com

Using Node.js with Caché
Caché Version 2018.1 2019-09-20

For Support questions about any InterSystems products, contact:

InterSystems Worldwide Response Center (WRC)
Tel: +1-617-621-0700
Tel: +44 (0) 844 854 2917
Email: support@InterSystems.com

Table of Contents

About This Book

This book describes how to install and use the cache.node module, which provides access to Caché from a Node.js application.

This book contains the following sections:

- Introduction
- Using cache.node NoSQL Methods
- NoSQL Methods for Global Nodes
- NoSQL Methods for Structured JSON Objects
- Integration with Caché Objects

There is also a detailed Table of Contents.

For general information, see Using InterSystems Documentation.

1

Introduction

This document describes an add-on module for the Node.js environment: cache.node. This module implements high performance access to data held in the Caché database for software developed in the Node.js environment.

1.1 Background

Node.js provides a JavaScript operating environment based on Google's highly optimized V8 JavaScript engine. Most will be familiar with JavaScript running on a client computer within the context of a web browser. Node.js, by contrast, is predominantly a middle-tier and/or server side operating environment.

The stated aim of the Node.js project is to provide a means through which high performance and highly scalable network infrastructure can be easily created. Node.js attempts to differentiate itself from conventional thread based networking software (such as web servers) by providing a non-blocking event based architecture.

Most modern web servers (with the notable exception of the newer event based servers such as Nginx and Lighttpd) are implemented using a hybrid multi-process/multithreaded server architecture where a thread is dedicated to serving each request. All resources associated with that thread remain allocated until the request is satisfied and a response dispatched to the client. In the Node.js architecture a single process is used per instance (of node) and threads are only used as a means through which operations and system calls that would otherwise block can be called asynchronously with respect to the primary thread of execution. Everything happens asynchronously in Node.js and a primary design feature of Node.js (and software written in Node.js) is that no function should block, but rather accept a callback function that Node.js will use to notify the initiating program that the operation has completed.

A consequence of the event based server architecture is that system resources (such as heap memory) are never consumed while waiting for some other operation to complete. Because of this, network programs developed using Node.js occupy a very small footprint in terms of system resource and memory usage and a high level of scalability is the key benefit.

Developers using Node.js need to be familiar with the following two technology areas:

- Event based architecture and asynchronous operation.
- Data represented using JavaScript Object Notation (JSON).

A key add-on requirement for the high-performance Node.js environment is a similarly high-performing database engine. This is where the Caché database comes in. The Caché database is provided by InterSystems. Caché is a mature database, the underlying technology for which has been under continual development and improvement since its launch in the late 1970s. Caché provides a multi-model database capability in that it can concurrently provide NoSQL, Relational and Object based access to the same physical data.

At its most basic level, the Caché database engine can be regarded as a NoSQL database providing B-Tree based storage, but with one important difference. Whereas conventional B-Tree databases implement one-dimensional key/value storage, Caché uses a multi-dimensional multiple key/value storage model. This provides the basis for a highly structured NoSQL data repository. In Caché terminology, a set of stored key-value pairs is known as a *global*, and each pair is a *global node*. For example:

One-dimensional Global storage:

```
MyGlobal("key1")="value1"
MyGlobal("key2")="value2"
MyGlobal("key3")="value3"
```

Multi-dimensional Global storage:

```
MyGlobal("key1")="value1"
MyGlobal("key1", "key1a")="value1a"
MyGlobal("key1", "key1a, "key1b")="value1b"
MyGlobal("key2")="value2"
MyGlobal("key2", 1)="value21"
MyGlobal("key2", 2)="value22"
MyGlobal("key3")="value3"
MyGlobal("key3", 1, 2, 3)="value3123"
MyGlobal("key3", 1, 3)="value313"
```

1.2 Installation

This section describes the procedure for installing a pre-built cache.node module.

Requirements

The following components should be installed:

- A Caché installation
- Node.js version 0.4.2 or later

Quick Installation Guide

The following instructions are provided for those already familiar with Node.js installations. Later sections in this chapter describe the installation procedure in more detail.

- *UNIX and Related Operating Systems*

  Copy the appropriate version of cache.node for the version of Node.js in use (renaming the file to cache.node) to the Node.js /lib/node/ directory. For example:

  ```
  /opt/local/node/lib/node/cache.node
  ```

- *Windows*

  Copy the appropriate version of cache.node for the version of Node.js in use (renaming the file to cache.node) to the Node.js home directory. For example:

  ```
  C:\Program Files\nodejs\cache.node
  ```

Versions

Node.js is available for a number of UNIX platforms. Native support for Windows was added in version 0.6.

The Caché module is named using the Node.js .node extension for both UNIX and Windows. In the kits supplied by InterSystems, the module to be used with a specific Node.js version is indicated by the name. For example, the module to be used with Node.js version 0.12 is named cache0120.node. The version-specific name may be changed to cache.node

before deployment to avoid having to modify the `'require'` statement in existing Node.js software. The following versions are available:

- Node.js version 0.4.x: cache.node
- Node.js version 0.6.x: cache061.node
- Node.js version 0.8.x: cache082.node
- Node.js version 0.10.x: cache0100.node
- Node.js version 0.12.x: cache0120.node
- Node.js version 4.2.x: cache421.node
- Node.js version 5.x.x: cache550.node
- Node.js version 6.x.x: cache610.node

The rest of this chapter is concerned with creating a working installation of cache.node with the hosting Node.js environment.

1.2.1 UNIX and Related Operating Systems

The module cache.node is actually a standard UNIX shared object (for example, cache.so) but with an unusual Node.js-specific file extension. Copy cache.node to the following location in the node file system:

```
.../node/lib/node/
```

For example:

```
/opt/local/node/lib/node/cache.node
```

Placing the module in the recognized library path will remove the need to specify the full path in the JavaScript files. For example, in JavaScript, it will be possible to write:

```
var cachedb = require('cache');
```

instead of a hard-coded path like this example:

```
var cachedb = require('/opt/local/build/default/cache');
```

1.2.2 Windows

The module cache.node is actually a standard Windows DLL (for example, cache.dll) but with an unusual Node.js-specific file extension. The standard Windows installer (provided by the Node.js Group) usually places the Node.js software in the following location: C:\Program Files\nodejs. The installation mechanism usually adds this location to the Windows `PATH` environment variable. If this is not the case, then add it manually. Also, the `NODE_PATH` environment variable should be set and it, too, should be set to the Node.js installation directory. For example:

```
NODE_PATH=C:\Program Files\nodejs
```

Copy cache.node to the location specified in the `NODE_PATH` environment variable:

```
C:\Program Files\nodejs
```

Placing the module in the recognized Node.js path will remove the need to specify the full path in the JavaScript files. For example, in JavaScript, it will be possible to write:

```
var globals = require('cache');
```

instead of a hard-coded path like the following examples:

```
var globals = require('/Program Files/nodejs/cache');

var globals = require('c:/Program Files/nodejs/cache');
```

2

Using cache.node NoSQL Methods

This chapter introduces some basic concepts and describes how to connect cache.node to the database. The following topics are discussed:

2.1 Comparing Globals and JSON Objects

In this section we will look at the basic relationship between JSON objects in the Node.js environment and Caché.

Consider the following global node:

```
^Customer(1)="Jane K. White"
```

Note: By convention, global names are prefixed with the '^' character in Caché. However, this convention need not be followed in the corresponding JSON representation.

The equivalent JSON construct will be:

```
{global: "Customer", subscripts: [1], data: "Jane K. White"}
```

Adding further nodes to this data construct:

globals:

```
^Customer(1)="Jane K. White"
^Customer(1, "Address", 1)="London"
^Customer(1, "Address", 2)="UK"
^Customer(1, "DateOfRegistration")="1 May 2010"
```

JSON:

```
{  global: "Customer",
   subscripts: [1],
   data: "Jane K. White"
}
{  global: "Customer",
   subscripts: [1, "Address", 1],
   data: "London"
}
{  global: "Customer",
subscripts: [1, "Address", 2],
data: "UK"
}
{  global: "Customer",
subscripts: [1, "DateOfRegistration"],
data: "1 May 2010"
}
```

2.2 cache.node Methods: (Synchronous vs. Asynchronous)

All methods provided by cache.node can be invoked either synchronously or asynchronously. While synchronous operation is conceptually easier to grasp and can be useful for debugging, the expectation in the Node.js environment is that all operations should be implemented asynchronously with a completion event raised by means of a callback function. For this reason, most of the examples given in this guide are coded to run asynchronously. To run them synchronously simply omit the callback function from the arguments list.

- *Synchronous operation:* Method does not return until the operation is complete.
- *Asynchronous operation:* Method returns immediately and your program is notified when the operation is complete. In order to accept notification, you must pass a callback function as an argument to the original method call. By convention, the callback will always be the last argument in the list.

Consider the method to determine the version of the cache.node module in use:

Synchronous operation:

```
var result = mydata.version();
```

Asynchronous operation:

```
mydata.version(
   function(error, result) {
      if (error) { // error
      }
      else { // success
      }
   }
);
```

The standard convention for reporting errors in Node.js is implemented in cache.node. If an operation is successful, *error* will be false. If an error occurs then *error* will be true and the *result* object will contain the details according to the following JSON construct:

```
{
   "ErrorMessage": [error message text],
   "ErrorCode": [error code],
   "ok": [true|false]
}
```

For synchronous operation, you should check for the existence of these properties in the result object to determine whether or not an error has occurred.

2.3 Opening and Closing the Caché Database

- Connecting via the Caché API
- Connecting to Caché Via the Network
- Multiple Instances of Caché Connectivity
- Closing Access to the Caché Database

Before any other methods can be called, the cache.node module must be loaded, an instance of the Caché object created and the target Caché database opened before any data (or Caché functionality) can be accessed.

Loading the Cache.node Module

If the cache.node module has been installed in the correct location for your Node.js installation, the following line will successfully load it from the default location:

```
var cachedb = require('cache');
```

If the module is installed in some other location the full path should be specified. For example:

```
var cachedb = require('/opt/cm/node0120/build/default/cache');
```

Creating an Instance

The next task is to create an instance of the cache.node object.

```
var mydata = new cachedb.Cache();
```

There are two ways to connect to Caché. First, there is the option of API level connectivity to a local Caché instance and, second, network based connectivity to either a local or remote Caché instance.

2.3.1 Connecting Via the Caché API

Connecting to a local instance of Caché via its API offers high performance integration between Node.js and Caché.

```
mydata.open(parameters[, function(error, result){}]);
```

where:

- *parameters.path* — Path to the mgr directory of the target Caché installation.
- *parameters.username* — User name for access.
- *parameters.password* — Password for access.
- *parameters.namespace* — Target Caché namespace.

Example:

```
mydata.open({  path:"/cache20151/mgr",
               username: "_SYSTEM",
               password: "SYS",
               namespace: "USER"
            }
            [, function(error, result){}]
);
```

The Caché Principal Device

The Caché principal input and output device for the cache.node session can be specified by defining the `input_device` and `output_device` properties respectively.

Example (using the standard input/output device):

```
mydata.open({  path: '/opt/cache20151/mgr',
               username: "_system",
               password: "SYS",
               namespace: "user",
               input_device: "stdin",
               output_device: "stdout"}
            [, function(error, result){}]
);
```

The default is to use the `NULL` device for cache.node sessions.

2.3.2 Connecting to Caché Via the Network

Network based connectivity to Caché can be used to connect to either a local or remote instance of Caché. This method is particularly useful for connecting to Caché systems installed on platforms for which Node.js is not natively available.

Also, if Node.js is hosting a public facing web application then network based connectivity to Caché will provide the basis for a secure architecture in which the database is physically separated from the web server tier:

```
mydata.open(parameters[, function(error, result){}]);
```

where:

- *parameters.ip_address* — Network name or IP address of the target Caché installation.
- *parameters.tcp_port* — Superserver TCP port of the targetCaché installation.
- *parameters.username* — User name for access.
- *parameters.password* — Password for access.
- *parameters.namespace* — Target Caché namespace.

Example:

```
mydata.open({  ip_address: "127.0.0.1",
               tcp_port: 56773,
               username: "_SYSTEM",
               password: "SYS",
               namespace: "USER"
            }
            [, function(error, result){}]
);
```

2.3.3 Multiple Instances of Caché Connectivity

The cache.node module will allow multiple instances of the Caché class to be created per hosting Node.js process if (and only if) TCP based connectivity to Caché is used.

Example 1: Creating two instances of the Caché class

```
var cachedb = require('cache');
var user = new cachedb.Cache();
var samples = new cachedb.Cache();

user.open({ ip_address: "127.0.0.1",
            tcp_port: 1972,
            username: "_SYSTEM",
            password: "SYS",
            namespace: "USER",
         }
);

samples.open({ ip_address: "127.0.0.1",
               tcp_port: 1972,
               username: "_SYSTEM",
               password: "SYS",
               namespace: "SAMPLES",
            }
);

console.log("'user' instance in namespace: " + user.get_namespace());
console.log("'samples' instance in namespace: " + samples.get_namespace());

user.close();
samples.close();
```

If Caché API based connectivity is used, it is still the case that only one instance can be created in a single Node.js process. This is because the Caché executable to which the hosting Node.js process binds is inherently single threaded. However, it is possible to create one connection using API based connectivity together with one or more TCP based connections.

Example 2: Creating two instances of the Caché class (one API and one TCP)

```
var cachedb = require('cache');
var user = new cachedb.Cache();
var samples = new cachedb.Cache();

// API based connection to Cache
user.open({ path:"/cache20152/mgr",
            username: "_SYSTEM",
            password: "SYS",
            namespace: "USER",
         }
);

// TCP based connection to Cache
samples.open({ ip_address: "127.0.0.1",
               tcp_port: 1972,
               username: "_SYSTEM",
               password: "SYS",
               namespace: "SAMPLES",
            }
);

console.log("'user' instance in namespace: " + user.get_namespace());
console.log("'samples' instance in namespace: " + samples.get_namespace());
```

```
user.close();
samples.close();
```

2.3.4 Closing Access to the Caché Database

The following method will gracefully close a previously opened Caché database.

```
mydata.close([function(error, result){}]);
```

2.4 Optional open() Settings

- Specifying Character Encoding
- Disabling the Caché Serialization Lock
- Enabling Debug Mode
- Specifying an Interrupt Signal Handler

2.4.1 Specifying Character Encoding

The default character encoding used in cache.node is UTF-8. Alternatively, 16-bit Unicode (UTF-16) can be used by defining this character encoding in the **open()** method.

Example:

```
mydata.open({  path:"/cache20151/mgr",
               username: "_SYSTEM",
               password: "SYS",
               namespace: "USER",
               charset: "UTF-16"
            }
               [, function(error, result){}]
);
```

2.4.2 Disabling the Caché Serialization Lock

A Caché process is inherently single threaded. Events fired asynchronously in the Node.js environment depend on the use of multiple threads to complete tasks concurrently within the context of a single process. For this reason, a serialization lock is applied to all Caché operations (invoked through cache.node) to ensure that only one command at a time is submitted to the associated Caché process.

However, locking is expensive. As a performance enhancement, the serialization lock may be disabled in the **open()** method if it can be ascertained that the Node.js application will only submit one task at a time to Caché. The lock should not be disabled if cache.node methods are used asynchronously. Even if synchronous mode is used throughout (with respect to the cache.node methods), it must be ascertained that calls to Caché fired synchronously cannot overlap as a result of events firing asynchronously elsewhere in the application. If in doubt, the lock should remain enabled. If an application crash occurs with the lock disabled then it should be re-enabled as a first step in diagnosing the problem.

Example: Disabling the serialization lock

```
mydata.open({  path:"/cache20151/mgr",
               username: "_SYSTEM",
               password: "SYS",
               namespace: "USER",
               lock: 0
            }
            [, function(error, result){}]
);
```

2.4.3 Enabling Debug Mode

A trace facility is included in the cache.node module to facilitate easier debugging when problems occur. The trace will record all calls to the Caché Call-in API, the input arguments used and the result returned. If TCP based connectivity is used the trace will record all request buffers submitted to, and response buffers received from, Caché.

The trace facility is enabled by defining the `debug` property in the **open()** method.

Example 1: Write a trace to the console (i.e. stdout)

```
mydata.open({  path:"/cache20151/mgr",
               username: "_SYSTEM",
               password: "SYS",
               namespace: "USER",
               debug: 1
            }
            [, function(error, result){}]
);
```

Example 2: Write a trace to a file (debug.log)

```
mydata.open({  path:"/cache20151/mgr",
               username: "_SYSTEM",
               password: "SYS",
               namespace: "USER",
               debug: "debug.log"
            }
            [, function(error, result){}]
);
```

Example trace:

Node.js Code:

```
var cachedb = require('cache');
var mydata = new cachedb.Cache();
mydata.open({  path:"/opt/cache20151/mgr",
               username: "_SYSTEM",
               password: "SYS",
               namespace: "USER"
               debug: "debug.log",
            });
mydata.set("Person", "1", "Jane K. White");
var data = mydata.get("Person ", 1));
mydata.close();
```

This will produce the following trace (or similar):

```
    >>> 000007FEF9E39340==fopen(debug.log, "a")
        (Debug trace file opened)
>>> cache_open
    >>> 0000000180000000==sys_dso_load(
       /opt/cache20151/bin/libcache.so)
    >>> 0==CacheSetDir(/opt/cache20151/mgr)
    >>> 0==CacheSecureStartA(0000000000126CC0(_SYSTEM),
        0000000000116CA0(SYS),
        000000000010EC90(Node.JS), 56, 15,
        0000000000106C80(//./nul), 000000000012ECD0(//./nul))
>>> cache_set
    >>> 0==CachePushGlobal(6, Person)
    >>> 00000000269E7010==CacheExStrNew(000000002682D770, 2)
    >>> 0==CachePushExStr(000000002682D770)
        >>> 1
```

```
            >>> 00000000269E7010==CacheExStrNew(000000002682D788, 11)
            >>> 0==CachePushExStr(000000002682D788)
                >>> Jane K. White
            >>> 0==CacheGlobalSet(1)
            >>> 0==CacheExStrKill(000000002682D770)
            >>> 0==CacheExStrKill(000000002682D788)

        >>> cache_get
            >>> 0==CachePushGlobal(6, Person)
            >>> 00000000269E7010==CacheExStrNew(000000002682D770, 2)
            >>> 0==CachePushExStr(000000002682D770)
                >>> 1
            >>> 0==CacheGlobalGet(1, 0)
            >>> 0==CachePopExStr(000000000013F410)
            >>> 0==CacheExStrKill(000000000013F410)
                >>> Jane K. White
            >>> 0==CacheExStrKill(000000002682D770)

        >>> cache_close
            >>> 0==CacheEnd()
```

2.4.4 Specifying an Interrupt Signal Handler

The cache.node module includes an optional signal handler to facilitate the clean interrupt of Node.js processes connected to Caché. Ideally, instead of relying on this facility, applications should include a suitable termination signal handler in the JavaScript code.

For example: to respond to the `SIGINT` signal (`Control-C`) in JavaScript code:

```
process.on( 'SIGINT', function() {
   // clean up and close down gracefully
});
```

In the absence of such an application level signal handler, the cache.node module can be instructed to register its own termination signal handler. Use the `stop_signal` property in the **open()** method to specify that cache.node should terminate execution on receiving signal `SIGINT` or `SIGTERM` (or both).

For example, to instruct cache.node to close the Node.js process on receiving either a `SIGINT` or `SIGTERM` signal:

```
user.open({path: "/cache20163/mgr",
           username: "_SYSTEM",
           password: "SYS",
           namespace: "USER",
           stop_signal: "SIGINT,SIGTERM"
          }
);
```

The signal handlers implemented in the cache.node module do no more than simply close down connectivity to Caché and halt the hosting Node.js process.

2.5 Utility Functions

- Pausing a Node.js Operation: sleep()
- Getting Version Information: version() and about()
- Invoking a Caché Function: function()

2.5.1 Pausing a Node.js Operation: sleep()

When troubleshooting Node.js applications, particularly those making extensive use of asynchronous completion techniques, it is often convenient to pause a function. The cache.node module contains a **sleep()** method to pause the current thread of execution for a period of time (in milliseconds) defined as the argument.

Example: (sleep for 5 seconds)

```
mydata.sleep(5000);
```

Note: This facility should only be used for troubleshooting as Node.js is architected according to the principle that no operation should block the main thread of execution.

2.5.2 Getting Version Information: version() and about()

The **version()** and **about()** methods return basic version information about the cache.node module in use and the associated Caché database (if open).

Synchronous:

```
var result = mydata.version();
```

Or:

```
var result = mydata.about();
```

Asynchronous:

```
mydata.version(function(error, result){});
```

Or:

```
mydata.about(function(error, result){});
```

Example:

```
mydata.version(
   function(error, result) {
      if (error) { // error (see result.ErrorMessage and result.ErrorCode)
      }
      else { // success
      }
   }
);
```

Result:

If the Caché database is not open:

```
Node.js Adaptor for Cache: Version: 1.1.112 (CM)
```

If the Caché database is open:

```
Node.js Adaptor for Cache: Version: 1.1.112 (CM); Cache Version: 2016.3 build 168
```

2.5.3 Invoking a Caché Function: function()

Functions contained within the Caché environment can be called directly via the **function()** method.

Synchronous:

```
var result = mydata.function(cache_function);
```

Asynchronous:

```
var result = mydata.function(cache_function, function(error, result){});
```

The Math routine

The examples in this section will be based on calls to the following Caché routine:

```
Math            ; Math Functions
                ;
Add(X,Y)        ; Add two numbers
                Quit (X * Y)
                ;
Multiply(X,Y)   ; Add two numbers
                Quit (X * Y)
                ;
```

This is a simple Caché routine called **Math** containing functions to perform basic mathematical functions (**Add** and **Multiply**).

Example 1 (Synchronous/Non-JSON)

```
result = mydata.function("Add^Math", 3, 4);
```

result: 7

Example 2: (Synchronous/JSON)

```
result = mydata.function({function: "Add^Math", arguments: [3, 4]},
```

result:

```
{
   "function": "Add^Math",
   "arguments": [3, 4],
   "result": 7
}
```

Example 3: (Asynchronous/JSON)

```
mydata.lock(
   {function: "Add^Math", arguments: [3, 4]},
   function(error, result) {
      if (error) { // error (see result.ErrorMessage and result.ErrorCode)
      }
      else { // success
      }
   }
);
```

result:

```
{
   "ok": 1
   "function": " Add^Math ",
        "arguments": [3, 4],
        "result": 7
}
```

3

NoSQL Methods for Global Nodes

In the following subsections we will discuss the basic NoSQL-style methods which include the following operations:

Create, read, update, and delete operations

- Save a Global Node: set()
- Retrieve a Global Node: get()
- Delete a Global Node: kill()

Iteration

- Test for the Existence of a Global Node: data()
- Get the Next Global Subscript: next()
- Get the Previous Global Subscript: previous()
- Get the Next Global Node: next_node()
- Get the Previous Global Node: previous_node()

Other operations

- Get the Next Value for an Integer Held in a Global Node: increment()
- Copy a Global: merge()
- Get a List of Globals in the Directory: global_directory()
- Lock and Unlock Global Nodes: lock() and unlock()

Note: **Data used in Basic NoSQL examples**

In this chapter the examples will be based on the following simple Caché data set:

```
^Customer(1)="Jane K. White"
^Customer(1, "address")="Banstead"
^Customer(2)="John P.Jackson"
^Customer(3)="Jane Smith"
```

3.1 Save a Global Node: set()

Synchronous:

```
var result = mydata.set(node);
```

Asynchronous:

```
mydata.set(node, function(error, result){});
```

Example:

```
mydata.set(
   {global: "Customer", subscripts: [1], data: "Jane K. White"},
   function(error, result) {
      if (error) { // error (see result.ErrorMessage and result.ErrorCode)
      }
      else { // success
      }
   }
);
```

result:

```
Operation successful if no error reported.
```

3.2 Retrieve a Global Node: get()

Synchronous:

```
var result = mydata.get(node);
```

Asynchronous:

```
mydata.get(node, function(error, result){});
```

Example 1 (Synchronous)

```
result = mydata.get({global: "Customer", subscripts: [9]});
```

result:

```
{global: "Customer", subscripts: [9], data: "", defined: 0}
```

Example 2 (Asynchronous)

```
mydata.get(
   {global: "Customer", subscripts: [1]},
   function(error, result) {
      if (error) { // error (see result.ErrorMessage and result.ErrorCode)
      }
      else { // success
      }
   }
);
```

result:

```
{global: "Customer", subscripts: [1], data: "Jane K. White", defined: 1}
```

Note that the `defined` property is set to `1` if the global node is actually defined and set to `0` if it is not.

3.3 Delete a Global Node: kill()

Synchronous:

```
var result = mydata.kill(node);
```

Asynchronous:

```
mydata.kill(node, function(error, result){});
```

Example:

```
mydata.kill(
   {global: "Customer", subscripts: [1]},
   function(error, result) {
      if (error) { // error (see result.ErrorMessage and result.ErrorCode)
      }
      else { // success
      }
   }
);
```

result:

Operation successful if no error reported.

3.4 Test for the Existence of a Global Node: data()

This method will return one of the following numeric values:

- `0` — Node does not exist.
- `1` — Node contains data but has no subnodes.
- `10` — Node does not contain data but has subnodes.
- `11` — Node contains data and has subnodes.

This method will return one of the following numeric values:

- 0: Global node is undefined.
- 1: Global node points to a data value. Global(1)="Data" data(Global(1))
- 10: Global node does not point to data but is further subscripted. Global(1,1)="Data" data(Global(1))
- 11: Global node points to a data value and is further subscripted. Global(1)="Data 1", ^Global(1,1)="Data 1" data(Global(1))

Synchronous:

```
var result = mydata.data(node);
```

Asynchronous:

```
mydata.data(node, function(error, result){});
```

Example:

```
mydata.data(
   {global: "Customer", subscripts: [1]},
   function(error, result) {
      if (error) { // error (see result.ErrorMessage and result.ErrorCode)
      }
      else { // success
      }
   }
);
```

result:

```
{defined: 11}
```

3.5 Get the Next Global Subscript: next()

Get the next value in the collating sequence for a particular level of subscript.

Synchronous:

```
var result = mydata.order(node);
```

Asynchronous:

```
mydata.order(node, function(error, result){});
```

Example 1 (Synchronous)

```
result = mydata.order({global: "Customer", subscripts: [""]},
```

result:

```
{global: "Customer", subscripts: [1], result: "1"}}
```

Example 2 (Synchronous)

```
result = mydata.next({global: "Customer", subscripts: [3]},
```

result:

```
{global: "Customer", subscripts: [""], result: ""}}
```

Example 3 (Asynchronous)

```
mydata.next(
   {global: "Customer", subscripts: [1]},
   function(error, result) {
      if (error) { // error (see result.ErrorMessage and result.ErrorCode)
      }
      else { // success
      }
   }
);
```

result:

```
{global: "Customer", subscripts: [2], result: "2"}}
```

Example 4 (Parse at the first level)

```
key = {global: "Customer", subscripts: [""]};
while ((key = user.next(key)).result != "") {
   console.log(JSON.stringify(key, null, '\t'))
}
```

result:

```
{global: "Customer", subscripts: [1], result: "1"}}
{global: "Customer", subscripts: [2], result: "2"}}
{global: "Customer", subscripts: [3], result: "3"}}
```

If there is no further subscript defined in the sequence then the result property will be returned as empty string ("").

3.6 Get the Previous Global Subscript: previous()

Get the previous value in the collating sequence for a particular level of subscript.

Synchronous:

```
var result = mydata.previous(node);
```

Asynchronous:

```
mydata.previous(node, function(error, result){});
```

Example 1 (Synchronous)

```
result = mydata.previous({global: "Customer", subscripts: [""]},
```

result:

```
{global: "Customer", subscripts: [3], result: "3"}}
```

Example 2 (Synchronous)

```
result = mydata.previous({global: "Customer", subscripts: [2]},
```

result:

```
{global: "Customer", subscripts: ["1"], result: "1"}}
```

Example 3 (Asynchronous)

```
mydata.previous(
   {global: "Customer", subscripts: [1]},
   function(error, result) {
      if (error) { // error (see result.ErrorMessage and result.ErrorCode)
      }
      else { // success
      }
   }
);
```

result:

```
{global: "Customer", subscripts: [""], result: ""}}
```

Example 4 (Parse at the first level)

```
key = {global: "Customer", subscripts: [""]};
while ((key = user.previous(key)).result != "") {
   console.log(JSON.stringify(key, null, '\t'))
}
```

result:

```
{global: "Customer", subscripts: [3], result: "3"}}
{global: "Customer", subscripts: [2], result: "2"}}
{global: "Customer", subscripts: [1], result: "1"}}
```

If there is no further subscript defined in the sequence then the result property will be returned as empty string ("").

3.7 Get the Next Global Node: next_node()

Get the next whole global node in the collating sequence regardless of the level of subscript.

Synchronous:

```
var result = mydata.next_node(node);
```

Asynchronous:

```
mydata.next_node(node, function(error, result){});
```

Example 1 (Synchronous)

```
result = mydata.next_node({global: "Customer"},
```

result:

```
{global: "Customer", subscripts: [1], data: "Jane K. White, defined: 1}}
```

Example 2 (Synchronous)

```
result = mydata.next_node({global: "Customer", subscripts: [3]},
```

result:

```
{global: "Customer", defined: 0}}
```

Example 3 (Asynchronous)

```
mydata.next_node(
   {global: "Customer", subscripts: [1]},
   function(error, result) {
      if (error) { // error (see result.ErrorMessage and result.ErrorCode)
      }
      else { // success
      }
   }
);
```

result:

```
{
   global: "Customer",
   subscripts: [1, "address"],
   data: "Banstead",
   defined: 1
}
```

Example 4 (Parse global)

```
key = {global: "Customer"};
while ((key = user.next_node(key)).defined) {
   console.log(JSON.stringify(key, null, '\t'));
}
```

key:

```
{
   global: "Customer",
   subscripts: [1],
   data: "Jane K. White",
   defined: 1
}
{
   global: "Customer",
   subscripts: [1, "address"],
   data: "Banstead",
```

```
   defined: 1
}
{
   global: "Customer",
   subscripts: [2],
   data: "John P.Jackson",
   defined: 1
}
{
   global: "Customer",
   subscripts: [3],
   data: "Jane Smith",
   defined: 1
}
```

If there is no further node defined in the sequence then the defined property will be set to zero.

3.8 Get the Previous Global Node: previous_node()

Get the previous whole global node in the collating sequence regardless of the level of subscript.

Synchronous:

```
var result = mydata.previous_node(node);
```

Asynchronous:

```
mydata.previous_node(node, function(error, result){});
```

Example 1 (Synchronous)

```
result = mydata.previous_node({global: "Customer", subscripts: ["z"]},
```

result:

```
{global: "Customer", subscripts: [3], data: "Jane Smith", defined: 1}}
```

Example 2 (Synchronous)

```
result = mydata.previous_node({global: "Customer", subscripts: [2]},
```

result:

```
{
   global: "Customer",
   subscripts: [1, "address"],
   data: "Banstead",
   defined: 1
}
```

Example 3 (Asynchronous)

```
mydata.previous_node(
   {global: "Customer", subscripts: [1]},
   function(error, result) {
      if (error) { // error (see result.ErrorMessage and result.ErrorCode)
      }
      else { // success
      }
   }
);
```

result:

```
{global: "Customer", defined: 0}}
```

Example 4 (Parse global)

```
key = {global: "Customer", subscripts: ["z"]};
while ((key = user.previous_node(key)).defined) {
   console.log(JSON.stringify(key, null, '\t'));
}
```

result:

```
{
   global: "Customer",
   subscripts: [3],
   data: "Jane Smith",
   defined: 1
}
{
   global: "Customer",
   subscripts: [2],
   data: "John P.Jackson",
   defined: 1
}
{
   global: "Customer",
   subscripts: [1, "address"],
   data: "Banstead",
   defined: 1
}
{
   global: "Customer",
   subscripts: [1],
   data: "Jane K. White",
   defined: 1
}
```

If there is no further node defined in the sequence then the defined property will be set to zero.

3.9 Get the Next Value for an Integer Held in a Global Node: increment()

This method provides an efficient way of uniquely assigning an (incremented) integer to a process without using locking. When called, the integer value held in the global is incremented and the new value returned to the calling program. Caché will guarantee that a unique number is assigned to multiple processes that may be accessing this function simultaneously.

Synchronous:

```
var result = mydata.increment(node);
```

Asynchronous:

```
mydata.increment(node, function(error, result){});
```

Example 1 (Increment ^Counter by one)

```
mydata.increment(
   {global: "Counter"},
   function(error, result) {
      if (error) { // error (see result.ErrorMessage and result.ErrorCode)
      }
      else { // success
      }
   }
);
```

result:

If successful the integer value held in global *^Counter* will be incremented and the new value returned as the result.

Example 2 (Increment ^Counter by one – JSON based argument)

```
result = mydata.increment({global: "Counter", increment: 1})
```

Example 3 (Increment ^Counter(1) by two)

```
result = mydata.increment({global: "Counter", subscripts: [1], increment: 2})
```

3.10 Copy a Global: merge()

The **merge()** method copies either a whole global or a section of a global to another global.

Synchronous:

```
var result = mydata.merge(
   {to: {destination_node},
    from: {source_node}};
);
```

Asynchronous:

```
mydata.merge(
   {{ to: {destination_node},
      from: {source_node}},
   function(error, result){}
);
```

Example 1: Copy a whole global

```
mydata.merge(
   {to: {global: "CopyOfCustomer"},
    from: {global: "Customer"}},
   function(error, result) {
      if (error) { // error (see result.ErrorMessage and result.ErrorCode)
      }
      else { // success
      }
   }
);
```

result:

If successful, the whole of global *^Customer* will be copied to *^CopyOfCustomer*.

Example 2: Copy a section of global

```
mydata.merge(
   {to: {global: "Customer", subscripts: [7, "address"]},
    from: {global: "Customer", subscripts: [1, "address"]}},
   function(error, result) {
      if (error) { // error (see result.ErrorMessage and result.ErrorCode)
      }
      else { // success
      }
   }
);
```

result:

If successful, the subsection of global contained under `"Customer(1,'address')"` will be copied to `"Customer(7,'address')"`.

3.11 Get a List of Globals in the Directory: global_directory()

This method will obtain a list of globals held in the directory. The number of names returned can be controlled using the range limiting properties `lo`, `hi` and `max`.

Synchronous:

```
var result = mydata.global_directory(range);
```

Asynchronous:

```
mydata.global_directory (range, function(error, result){});
```

Example 1: Obtain a list of all globals in the directory

```
mydata.global_directory({},
   function(error, result) {
      if (error) { // error (see result.ErrorMessage and result.ErrorCode)
      }
      else { // success
      }
   }
);
```

result:

If successful, an array containing all global names found will be returned.

Example 2: Obtain a list of all globals whose name begins with "Cust"

```
mydata.global_directory(
   {lo: "Cust", hi: "Cust~"},
   function(error, result) {
      if (error) { // error (see result.ErrorMessage and result.ErrorCode)
      }
      else { // success
      }
   }
);
```

result:

If successful, an array containing all global names beginning with `Cust` will be returned.

3.12 Lock and Unlock Global Nodes: lock() and unlock()

The Caché database contains the facility to lock either whole (for example, *^Customer*) or subsections of a global (for example, *^Customer(1)*) for exclusive use.

Synchronous:

```
var result = mydata.lock(node, timeout);
var result = mydata.unlock(node);
```

Asynchronous:

```
mydata.lock(node, timeout, function(error, result){});
mydata.unlock(node, function(error, result){});
```

Example:

```
mydata.lock(
   {global: "Customer", subscripts: [1]}, 10,
   function(error, result) {
      if (error) { // error (see result.ErrorMessage and result.ErrorCode)
      }
      else { // success
      }
   }
);
```

result:

Operation successful if no error reported. The return value will be `1` if the Global was successfully locked, or `0` if the lock request timed out.

Remember to, at some point, unlock all global nodes that your code previously locked in order to avoid deadlock situations. The lock request is placed on a wait queue if the node is already locked by another process, and will wait indefinitely (until the process terminates) if no *timeout* is specified.

Finally, to release all locks held by your program call the **unlock()** method with no node argument.

```
var result = mydata.unlock();
mydata.unlock(function(error, result){});
```

All locks held by a Node.js process will be released if and when the process terminates.

4

NoSQL Methods for Structured JSON Objects

This section describes advanced methods that perform operations that could otherwise be achieved by combining several of the basic NoSQL-style methods.

- Retrieve a List of Global Nodes: retrieve("list")
- Retrieve a List of Global Nodes Recursively: retrieve("array")
- Save a List of Global Nodes: update("array")
- Retrieve a Structured Data Object: retrieve("object")
- Save a Structured Data Object: update("object")
- Controlling the Amount of Data Returned by Retrieve Operations

The following Caché data will be used to illustrate these methods:

Data used in advanced NoSQL examples

```
^Customer(1)="Jane K. White"
^Customer(1, "Address", 1)="London"
^Customer(1, "Address", 2)="UK"
^Customer(1, "DateOfRegistration")="1 May 2010"
^Customer(2)="John P.Jackson"
^Customer(2, "Address", 1)="Reigate"
^Customer(2, "Address", 2)="UK"
^Customer(2, "DateOfRegistration")="7 May 2010"
^Customer(3)="Jane Smith"
^Customer(3, "Address", 1)="London"
^Customer(3, "Address", 2)="UK"
^Customer(3, "DateOfRegistration")="9 June 2010"
```

4.1 Retrieve a List of Global Nodes: retrieve("list")

This method will return a list of subscript values that are defined directly beneath a given level in the global.

Synchronous:

```
var result = mydata.retrieve(node, "list");
```

Asynchronous:

```
mydata.retrieve(node, "list", function(error, result){});
```

Example 1: Retrieve a list of customer numbers

```
mydata.retrieve(
   {global: "Customer"},
   "list",
   function(error, result) {
      if (error) { // error (see result.ErrorMessage and result.ErrorCode)
      }
      else { // success
      }
   }
);
```

result (three Customer records):

```
[1, 2, 3]
```

Example 2: Retrieve a list of address lines for a specific customer

```
mydata.retrieve(
   {global: "Customer", subscripts: [1, "address"]},
   "list"
   function(error, result) {
      if (error) { // error (see result.ErrorMessage and result.ErrorCode)
      }
      else { // success
      }
   }
);
```

result (two lines in the address for customer number 1):

```
[1, 2]
```

4.2 Retrieve a List of Global Nodes Recursively: retrieve("array")

This method will return a list of subscript values together with their associated data values that are defined beneath a given level in the global. The method is recursive and will retrieve all global nodes defined beneath a chosen level.

The result will be expressed as an array of global nodes: each global node expressed as a single JSON object.

Synchronous:

```
var result = mydata.retrieve(node, "array");
```

Asynchronous:

```
mydata.retrieve(node, "array", function(error, result){});
```

Example 1: Retrieve all data for a customer

```
mydata.retrieve({global: "Customer", subscripts: [1]}, "array",
   function(error, result) {
      if (error) {
         // error   (see result.ErrorMessage
         //       and result.ErrorCode)
      }
      else {
         // success
      }
   }
);
```

Result:

```
[
   {   global: "Customer",
   subscripts: [1],
   data: "Jane K. White"
   }
   {   global: "Customer",
   subscripts: [1, "Address", 1],
   data: "London"
   }
   {   global: "Customer",
   subscripts: [1, "Address", 2],
   data: "UK"
   }
   {   global: "Customer",
   subscripts: [1, "DateOfRegistration],
   data: "1 May 2010"
   }
]
```

4.3 Save a List of Global Nodes: update("array")

This method performs the opposite task to that performed by the retrieve method. In other words the update method takes an array of global nodes, each expressed as a single JSON object and writes them back to the Caché database.

Synchronous:

```
var result = mydata.update(node, "array");
```

Asynchronous:

```
mydata.update(node, "array", function(error, result){});
```

Example 1: Create or update a record for a customer

```
mydata.update([
   {   global: "Customer",
   subscripts: [1],
   data: "Jane K. White"
}
   {   global: "Customer",
   subscripts: [1, "Address", 1],
   data: "London"
   }
   {   global: "Customer", subscripts: [1, "Address", 2],
   data: "UK"
   }
   {   global: "Customer",
   subscripts: [1, "DateOfRegistration],
   data: "1 May 2010"
   }
   ],
   "array",
   function(error, result) {
      if (error) {
         // error   (see result.ErrorMessage
         //       and result.ErrorCode)
      }
      else {
         // success
      }
   }
);
```

Result:

Operation successful if no error reported.

4.4 Retrieve a Structured Data Object: retrieve("object")

This method will retrieve a structured data object defined beneath a given level in a global. This method provides a structured alternative to the array of global nodes described previously.

The result will be expressed as a single JSON object.

Synchronous:

```
var result = mydata.retrieve(node, "object");
```

Asynchronous:

```
mydata.retrieve(node, "object", function(error, result){});
```

Example: Retrieve all data for a customer

Consider the following structured Caché global:

```
^Customer(1, "Name")="Jane K. White"
^Customer(1, "Address", 1)="London"
^Customer(1, "Address", 2)="UK"
^Customer(1, "DateOfRegistration")="1 May 2010"
```

In Node.js:

```
mydata.retrieve({global: "Customer", subscripts: [1]}, "object"
   function(error, result) {
      if (error) {
         // error    (see result.ErrorMessage
         //       and result.ErrorCode)
      }
      else {
         // success
      }
   }
);
```

Result:

```
{
   node: {   global: "Customer",
      subscripts: [1]
   }
   object: { Name: "Jane K. White",
      Address: {"1": "London", "2": "UK"}
      DateOfRegistration: "1 May 2010",
   }
}
```

4.5 Save a Structured Data Object: update("object")

This method performs the opposite task to that performed by the retrieve method. In other words it writes the contents of a structured JSON object back to the Caché database.

Synchronous:

```
var result = mydata.update(node, "object");
```

Asynchronous:

```
mydata.update(node, "object", function(error, result){});
```

Example 1: Create or update a record for a customer

```
mydata.update({   node: {global: "Customer",
                         subscripts: [1]
                  },
           Object: {Name: "Jane K. White",
                  DateOfRegistration: "1 May 2010",
                  Address: {"1": "London", "2": "UK"}
                  },
              "object",
           function(error, result) {
           if (error) {
              // error    (see result.ErrorMessage
              //       and result.ErrorCode)
           }
           else {
              // success
           }
     }
);
```

Result:

Operation successful if no error reported. If successful, the following set of global nodes will be created for this example:

```
^Customer(1, "Address", 1)="London"
^Customer(1, "Address", 2)="UK"
^Customer(1, "DateOfRegistration")="1 May 2010"
^Customer(1, "Name")="Jane K. White"
```

4.6 Controlling the Amount of Data Returned by Retrieve Operations

Clearly the retrieve operations are capable, depending on the underlying global structure, of returning enormous volumes of data to the Node.js environment. The following properties can be added to the requesting JSON object (i.e. the object defining the root global node) to limit the amount of data returned.

- *max* — The maximum number of global nodes to return.
- *lo* — The lowest global subscript to return.
- *hi* — The highest global subscript to return.

If all three properties are defined, the retrieve operation will terminate when the *max* number of global nodes have been returned, even if the global subscript defined in *hi* has not been reached.

Consider the following Caché global:

```
^Customer(1, "Address", 1)="London"
^Customer(1, "Address", 2)="UK"
^Customer(1, "DateOfRegistration")="1 May 2010"
^Customer(1, "Name")="Jane K. White"
^Customer(1, "Orders", 20100501, "Reference")="Order001"
^Customer(1, "Orders", 20100503, "Reference")="Order002"
^Customer(1, "Orders", 20100507, "Reference")="Order003"
^Customer(1, "Orders", 20100509, "Reference")="Order004"
^Customer(1, "Orders", 20100510, "Reference")="Order005"
^Customer(1, "Orders", 20100520, "Reference")="Order006"
^Customer(1, "Orders", 20100527, "Reference")="Order007"
^Customer(1, "Orders", 20100906, "Reference")="Order008"
^Customer(1, "Orders", 20110104, "Reference")="Order011"
^Customer(1, "Orders", 20110203, "Reference")="Order012"
^Customer(2, "Address", 1)="Reigate"
^Customer(2, "Address", 2)="UK"
^Customer(2, "DateOfRegistration")="7 May 2010"
^Customer(2, "Name")="John P.Jackson"
^Customer(2, "Orders", 20101204, "Reference")="Order009"
```

```
^Customer(2, "Orders", 20101206, "Reference")="Order010"
^Customer(3, "Address", 1)="London"
^Customer(3, "Address", 2)="UK"
^Customer(3, "DateOfRegistration")="9 June 2010"
^Customer(3, "Name")="Jane Smith"
^Customer(4, "Name")="Jim Cooper"
^Customer(5, "Name")="Eve Simpson"
^Customer(6, "Name")="John Cotton"
^Customer(7, "Name")="Alison Clarke"
^Customer(8, "Name")="Paul Francis"
^Customer(9, "Name")="Susan Reed"
^Customer(10, "Name")="Mary Dodds"
```

Note: The third subscript in the `Orders` section is the date in `YYYYMMDD` format.

Example 1: List the registered Customer numbers between 2 and 7 (inclusive)

```
mydata.retrieve(
   {global: "Customer", lo: 2, hi: 7},
   "list",
   function(error, result) {
      if (error) { // error (see result.ErrorMessage and result.ErrorCode)
      }
      else { // success
      }
   }
);
```

result (six Customer records):

```
[2, 3, 4, 5, 6, 7]
```

Example 2: List the Orders placed by Customer number 1 between 7 May 2010 and 24 December 2010 (inclusive)

```
mydata.retrieve(
   {  global: "Customer",
      subscripts: [1, "Orders"],
      lo: 20100507,
      hi: 20101224
   },
   "list",
   function(error, result) {
      if (error) { // error (see result.ErrorMessage and result.ErrorCode)
      }
      else { // success
      }
   }
);
```

result (six Customer Order records):

```
[20100507, 20100509, 20100510, 20100520, 20100527, 20100906]
```

Example 3: List the full details for Orders placed by Customer number 1 in September 2010

```
mydata.retrieve(
   {  global: "Customer",
      subscripts: [1, "Orders"],
      lo: 20100901,
      hi: 20100930
   },
   "array",
   function(error, result) {
      if (error) { // error (see result.ErrorMessage and result.ErrorCode)
      }
      else { // success
      }
   }
);
```

result (one Customer Order record):

```
[
   {  global: "Customer",
      subscripts: [1, "Orders", 20100906, "Reference"]
      data: "Order008"
   }
]
```

Example 4: List first three Customers registered (by Customer number) in the database

```
mydata.retrieve(
   {global: "Customer", max: 3},
   "list",
   function(error, result) {
      if (error) { // error (see result.ErrorMessage and result.ErrorCode)
      }
      else { // success
      }
   }
);
```

Result (three Customer records):

```
[1, 2, 3]
```

5

Integration with Caché Objects

The cache.node module provides methods to allow Node.js applications to integrate directly with Caché objects. (instances of classes defined in the Caché class library).

The examples in this chapter are based on the contrived, but simple, Caché %Persistent class listed below:

Example: Class User.customer

```
Class User.customer Extends %Persistent
{
   Property number As %Integer;

   Property name As %String;

   ClassMethod MyClassMethod(x As %Integer) As %Integer
   {
      Quit x * 2
   }

   Method MyMethod(x As %Integer) As %Integer
   {
      Quit x * 3
   }
}
```

5.1 NoSQL Based Object-Oriented Development Methodology

The examples shown earlier in this book have illustrated the use of the Caché database as a raw NoSQL engine. The code in this chapter illustrates how the cache.node methods together with the JSON notation in JavaScript can be used to create a more object orientated development methodology.

Example:

```
var cachedb = require('cache');
var user = new cachedb.Cache();
user.open({
path:"/cache20102/mgr",
username: "_SYSTEM",
password: "SYS",
namespace: "USER"});

var customers = new Customers();
var customer = customers.getCustomer(1);

console.log("customer number 1 is " + customer.data);
customers.setCustomer(9, "Tom Smith");
user.close();

function Customers() {
   this.global = "Customer";
   this.subscripts = new Array();

   this.getCustomer = function(id) {
      this.subscripts[0] = id;
      var person = user.get(this);
      return person;
   }

   this.setCustomer = function(id, name) {
      this.subscripts[0] = id;
      this.data = name;
      var person = user.set(this);
      return;
   }
}
```

5.2 Invoke a Class Method: invoke_classmethod()

A method is an executable element defined by a class. Caché supports two types of methods: instance methods (keyword `Method`) and class methods (keyword `ClassMethod`). An instance method is invoked from a specific instance of a class and typically performs some action related to that instance. A class method is a method that can be invoked without reference to any instance; these are called static methods in other languages. The term method usually refers to an instance method. The more specific phrase class method is used to refer to class methods.

Synchronous:

```
var result = mydata.invoke_classmethod(class_method);
```

Asynchronous:

```
mydata.invoke_classmethod(class_method, function(error, result){});
```

Example:

```
var result = mydata.invoke_classmethod(
   {class: "User.customer",
    method: "MyClassMethod",
    arguments: [7]}
);
```

result:

```
{
   "ok": 1,
   "class": "User.customer",
   "method": "MyClassMethod",
   "arguments": ["7"],
   "result": "14"
}
```

5.3 Create a New Caché Object: create_instance()

Synchronous:

```
var instance = mydata.create_instance(class);
```

Asynchronous:

```
mydata.create_instance(class, function(error, instance){});
```

Example:

```
var instance = mydata.create_instance({class: "User.customer"});
```

instance:

```
{
   "ok": 1,
   "class": "User.customer",
   "method": "%New",
   "oref": 1
}
```

5.4 Open an Existing Caché Object: open_instance()

Synchronous:

```
var instance = mydata.open_instance(class);
```

Asynchronous:

```
mydata.open_instance(class, function(error, instance){});
```

Example:

```
var instance = mydata.open_instance({class: "User.customer", arguments: [1]});
```

instance:

```
{
   "ok": 1,
   "class": "User.customer",
   "method": "%OpenId",
   "arguments": ["1"],
   "oref": 1
}
```

5.5 Set the Value for a Property: set_property()

Synchronous:

```
var result = mydata.set_property(instance, property_value);
```

Asynchronous:

```
mydata.set_property(instance, property_value, function(error, result){});
```

Example 1:

```
var result = mydata.set_property(
   instance,
   {property: "number",
    value: 100000}
 );
```

result:

```
{
   "ok": 1,
   "oref": 1,
   "property": "number",
   "value": 100000
}
```

Example 2:

```
var result = mydata.set_property(
   instance,
   {property: "name",
   value: "Chris Munt"}
);
```

result:

```
{
   "ok": 1,
   "oref": 1,
   "property": "name",
   "value": "Chris Munt"
}
```

5.6 Get the Value for a Property: get_property()

Synchronous:

```
var result = mydata.get_property(instance, property);
```

Asynchronous:

```
mydata.get_property(instance, property, function(error, result){});
```

Example:

```
var get_property = mydata.get_property(instance, {property: "name"});
```

get_property:

```
{
   "ok": 1,
   "oref": 1,
   "property": "name",
   "value": "Chris Munt"
}
```

5.7 Invoke an Instance Method: invoke_method()

Synchronous:

```
result = mydata.invoke_method(instance, method);
```

Asynchronous:

```
mydata.invoke_method(instance, method, function(error, result){});
```

Example:

```
var result = mydata.invoke_method(instance, {method: "MyMethod", arguments: [3]});
```

result:

```
{
   "ok": 1,
   "oref": 1,
   "method": "MyMethod",
   "arguments": ["3"],
   "result": "9"
}
```

5.8 Save a Caché Object: save_instance()

Synchronous:

```
var result = mydata.save_instance(instance);
```

Asynchronous:

```
mydata.save_instance(instance, function(error, result){});
```

Example:

```
var result = mydata.save_instance(instance);
```

result:

```
{
   "ok": 1,
   "oref": 1
}
```

5.9 Close a Caché Object: close_instance()

Synchronous:

```
var result = mydata.close_instance(instance);
```

Asynchronous:

```
mydata.close_instance(instance, function(error, result){});
```

Example:

```
var result = mydata.close_instance(instance);
```

result:

```
{
   "ok": 1,
   "oref": 1
}
```

5.10 Putting it All Together

Adopting an object oriented approach based on Caché objects.

Example:

```
var new_customer = new customer();

new_customer.create();
new_customer.number = 100000;
new_customer.name = "Chris Munt";
new_customer.save();
new_customer.close();


function customer() {
   this.class = "User.Customer";
   this.instance = {};
   this.number = 0;
   this.name = "";

   this.create = function() {
      this.instance = user.create_instance({class: this.class});
      return this.instance;
   }

   this.save = function() {
      var set_number = user.set_property(this.instance,
                       {property: "number", value: this.number});
      var set_name = user.set_property(this.instance,
                       {property: "name", value: this.name});
      var save = user.save_instance(this.instance);
      return save;
   }

   this.close = function() {
      var close = user.close_instance(this.instance);
      return close;
   }
}
```