Documentation

StrongLoop Suite documentation

StrongLoop Suite includes:

- LoopBack, an open-source mobile backend framework for Node.js.
- StrongNode, professional support for Node.js, plus cluster management and other powerful modules.
- StrongOps, a built-in monitoring and management console.

To get up and running quickly, see Getting started.

What's new

LoopBack now enables you to send push notifications to mobile apps. See Creating push notifications for details. Client SDKs have been updated to support push notifications:

- Android SDK (version 1.2)
- iOS SDK (version 1.2)

StrongLoop now supports the Digital Ocean cloud platform.

See What's new for a complete list.

StrongNode

Overview

StrongNode contains:

- Node.js version version 0.10.22.
- StrongLoop modules to support application clustering.
- A set of commonly-used modules, known as community modules or *userland modules*.
- Comprehensive toolset intended for production use. See Command-line reference (slc).

⚠ To start creating an application right away, see Getting started with StrongNode.

StrongNode modules

StrongLoop has selected the best module available in npm for commonly-used features (for example: flow control, web server). Depending on your subscription plan, StrongLoop will help you to solve any problems you may encounter while using these modules. You don't have to troubleshoot third-party code and you can focus on building applications.

For links to the modules' documentation, see Supporting module reference.

StrongNode modules fall into two categories: cluster-related and other modules. Additionally, StrongLoop Suite supports a set of community ("userland") modules.

Cluster modules

The following modules enable Node applications to create child processes that all share the same network port. This enables applications to take advantage of multi-core systems:

- Strong-cluster-control - Allows for run-time management of cluster processes.
- Strong-store-cluster - A key-value store accessible to all nodes in a node.js
- Strong-cluster-connect-store - Provides sessions for Connect and Express applications. Manages workers, ensuring the correct number of workers are available; enables you to change the worker pool size without restarting the application.
- Strong-cluster-socket.io-store - Provides an easy solution for running a socket.io server when using node cluster.
- Strong-cluster-tls-store - An implementation of TLS session store using node's native cluster messaging. Additionally, provides several different message queue implementations, including cluster-native messaging.
- Strong-mq - An abstraction layer over common message distribution patterns.

Other modules

- Strong-cli - The StrongLoop command-line tool.
- Node Inspector – Debugging interface for Node.js.
- Strong-task-emitter - Performs an arbitrary number of tasks recursively and in parallel and, using an event emitter, passes the results in an asynchronous message.
- Strong-remoting - Makes objects and data in your Node application need to be reachable from other Node processes, browsers, and mobile clients.

REVIEW COMMENT

Removed - but should we include it here?

- Strong-agent - Enables performance monitoring of your node.js application. application services. Including system usage at every moment in time to uncover and resolve issues within the application as they arise.

Community modules

StrongLoop supports a set of commonly-used community modules that provide crucial functionality:

- Express – Web application framework.
- Connect – Rich middleware framework.
- Passport – Simple, unobtrusive authentication.
- Mongoose – Elegant mongodb object modeling.
- Async – Higher-order functions and common patterns for asynchronous code.
- Q – Tool for making and composing asynchronous promises in JavaScript.
- Request – Simplified HTTP request client.
- Socket.IO – Cross-browser WebSocket for realtime apps.
- Engine.IO – Transport layer for real time data exchange.
- Reggie – Lightweight alternative to a full blown npm registry.

Getting started with StrongNode

Prerequisite

Follow the instructions in Getting started to install Node and the `slc` command-line tool.

Quick Start

1. If you have not already done so, create the StrongLoop Suite Sample App. Enter this command:

```
$ slc example
```

2. Run the sample application in cluster mode.

```
$ cd sls-sample-app
$ slc run app --size=cpus
```

The `slc run` command passes the `--size` option to the sample application, which uses it to start one Node process for each CPU in the local system. You'll see a number of messages in your terminal window: one set of messages for each CPU core in your system.

If you have not set up a StrongOps account in this project using `slc strongops` then you will see a warning message:

```
StrongOps not configured to monitor....
```

Don't worry about this right now.

3. Open another terminal window and list the current processes by entering the following command. The number of processes will be equal to the total number of cores on your machine.

```
$ cd sls-sample-app
$ slc clusterctl
```

You'll see results in your console window something like this (for a four-core system):

```
worker count: 4
worker id 1: { pid: 32438 }
worker id 2: { pid: 32439 }
worker id 3: { pid: 32440 }
worker id 4: { pid: 32442 }
```

This tells you that there are four worker processes, and the process ID of each.

4. Resize the cluster to a larger number by entering the following command. In the example below, the new number of processes will be 10.

```
$ slc clusterctl set-size 10
$ slc clusterctl status
```

You should see something like this:

```
worker count: 10
worker id 1: { pid: 32438 }
worker id 2: { pid: 32439 }
worker id 3: { pid: 32440 }
worker id 4: { pid: 32442 }
worker id 5: { pid: 32468 }
worker id 6: { pid: 32469 }
worker id 7: { pid: 32470 }
worker id 8: { pid: 32472 }
worker id 9: { pid: 32474 }
worker id 10: { pid: 32476 }
```

Terminiate your application with Ctrl-C.

Using the chat room example

Using StrongNode, you can quickly build powerful and fully-supported Node.js applications. Once you build application, you can use StrongOps a s part of the StrongLoop Suite to operationalize your application in production.

The following example walks you through a prebuilt sample application.

Application structure

The following example implements a chat room using cluster, socket.io and express. First, let us instantiate the chat example using the command line:

```
$ slc example chat my-chat --no-install
$ cd my-chat
```

Let's look at the top-level directory of the example.

```
README.md     client/       lib/
bin/          app.js        package.json
```

The above shows a typical node application structure.

- README.md - A readme file in markdown format.
- client/ - A directory with an example client implementation.
- lib/ - The source code for the chat room application.
- bin/ - Where executable scripts go.
- app.js - The application's main, it is run by `slc run .`.
- package.json - The project definition for npm.

app.js

Though you can run the application with 'slc run .', app.js only contains:

```
if (require.main === module) {
  require('./bin/simple');
} else {
  module.exports = {
    SimpleServer: require('./lib/simple'),
    WorkerServer: require('./lib/worker')
  };
}
```

Typically, the root of any application has either an app.js or index.js that requires the implemention from ./lib or ./bin. Here, if you run app.js as 'slc run app.js', the condition`require.main === module` is true and the code requires ./bin/simple, otherwise, the code makes ./lib/simple and ./lib/worker available in exports. A description of exports is in the Require section.

Also by convention, the source code for the application is in ./lib. The other source directory, ./client, illustrates an example client implementation which we won't be covering here.

package.json

The package.json describes the application for npm. The file contains:

```
{
  "name": "strongnode-chat-example",
  "version": "0.0.0",
  "description": "A pair of examples to showcase how to use `socket.io` with a static
`express` server with `async` for control flow.",
  "main": "app.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "repository": "",
  "author": "Michael Schoonmaker <michael@strongloop.com>",
  "dependencies": {
    "async": "~0.2.8",
    "express": "~3.2.4",
    "socket.io": "~0.9.14",
    "strong-agent": "~0.2.2",
    "strong-cluster-control": "~0.0.1",
    "strong-cluster-socket.io-store": "~0.1.1",
    "strong-mq": "~0.0.4"
  },
  "license": "Commercial"
}
```

Some properties - name, version, description, repository, author, license - are useful when publishing to npm. The rest have immeadiate relevance:

- main - the entry point into this application
- scripts - map to actions that are run by npm, e.g. `npm test`
- dependencies - The npm modules this application depends upon. Here, a version of "~0.2.8" means approximate equivalent, e.g. "0.2.9" would be equivalent, but "0.3.0" would not.

Running the Application

To run this application, type the following in the root directory of the application:

```
$ npm install
```

This command installs all the dependent modules. Then type this to run the app:

```
$ node app.js
```

Now, the application is running. Type control-C to exit.

Understanding the code

Look at the ./lib folder.

```
$ cd ./lib
$ ls
simple.js  worker.js
```

The file, simple.js, defines the object SimpleServer which is network chat server using TCP. The other file, worker.js, defines the WokerServer class, which uses cluster to scale across CPUs. We'll cover only simple.js.

The require() function

Simple.js begins, as most Node files do, by requiring modules. The module being required has no qualifying path information, the module is assumed to be a node core module or a module installed in the application's node_modules directory, for example:

```
var http = require('http');
var path = require('path');

var async = require('async');
var socketio = require('socket.io');
var express = require('express');
```

In the previous code example, http and path are modules from node's core while async, socket.io and express are dependencies, installed by npm and reside in ./node_modules. If you recall these modules were specified as dependencies in package.json. When you invoke `npm install` npm installs the dependencies. Require is a node-specific command.

JavaScript constructor

The first function is the constructor function SimpleServer. It performs all the object set-up and initialization.

```
function SimpleServer(obj) {
  if (!(this instanceof SimpleServer)) {
    return new SimpleServer(obj);
  }

  obj = obj || {};

  this.router = express();
  this.server = http.createServer(this.router);
  this.io = socketio.listen(this.server);

  this.port = obj.port || 1337;
  this.messages = [];
  this.sockets = [];

  this._initExpress();
  this._initSocketIo();
}
```

module.exports

The only thing slightly different is the line with: `SimpleServer.createServer = SimpleServer;`. On the last line in the file you see:

`module.exports = SimpleServer;`

The previous line is how other modules can "see" SimpleServer. node.js limits the JavaScript global scope to the file. To share functionality, between files use module.exports. Note though, node.js has a global object, aptly named `global` which is available to every file in your application.

Look at how the file ./bin/simple is able to use SimpleServer.

```
var SimpleServer = require('../lib/simple');
var server = SimpleServer.createServer({
  port: process.argv[2]
});

server.start(function (err) {
  if (err) {
    console.error('Failed to start with:', err.message || err);
    process.exit(1);
  }

  console.log('Listening on port ' + server.port + '...');
});
```

First, the code requires the SimpleServer file which returns the contents of modules.exports. In this case that is the object SimpleServer. Then the code instantiates an instance of the object as server. Finally, server.start executes.

See how the argument to `server.start` is an anonymous JavaScript function? That's a callback. Callbacks are a common way that Node.js apps specify what to do on an asynchronous event that will happen at some time in the future. Other patterns are promises and events.

JavaScript object prototype

Going back to the source file `simple.js`, where the definition for SimpleServer server is, you'll notice the next function after the constructor is the implementation of the start method (as seen in ./bin/sample).

```
SimpleServer.prototype.start = start;
function start(callback) {
  var self = this;

  self.port = Number(process.env.PORT) || self.port;
  self.server.listen(self.port, callback);

  return self;
}
```

The start method also demonstrates how to extend the SimpleServer object in JavaScript by adding a function object onto its prototype. JavaScript uses protoype inheritance. Every object has a prototype you can extend. However, when you add something to an object's prototype, you alter all instances of that object type. The line with `SimpleServer.prototype.start = start;` extends the prototype for SimpleServer by adding the start function to a property called start.

This method has the chat server begin to listen on a socket, so it can accept connections. You can see the callback function in the argument - it's passed to this.server.listen, which once listening, will execute the callback.

JavaScript's "this"

The `var self = this;` is a common pattern. The var `this` is a reference to current instance of the object. However, don't forget that functions are objects. In the callback scope, this refers to the callback function and not the calling code in the outer function scope.

A common mistake is to use `this` in the callback when you need to refer to the calling object. The solution is define `self` as this and rely upon the JavaScript closure to pass the value to the callback.

Working with standard Node apps

Overview

StrongLoop Suite includes the `slc` command-line tool for working with applications. It provides commands for creating and working with both LoopBack applications and standard (non-LoopBack) Node applications. The following table lists some of the most important commands relevant to standard Node apps. For information on creating LoopBack apps, see Creating a LoopBack application.

Command	Sub-commands	Description
slc create	web package module cli	Create Node boilerplate application of the specified type. See Creating Node apps below for an overview.
slc example	suite chat urlsaver blog	Create StrongLoop example application of the specified type
slc debug		Debug module with Node Inspector.

For a complete command reference, see Command-line reference (slc).

Creating Node apps

The following sections provide some simple illustrations of creating Node apps.

Hello world

The following trivial one-line Node program will print `hello world` to the console and exit:

```
console.log('hello world');
```

Save the above as `hello-world.js` and run the following command:

```
$ slc hello-world
```

This will run the `hello-world.js` script as a Node application. You'll see the output:

```
hello world
```

Creating a CLI program

You can use slc to create a basic command-line interface (CLI) application. Enter the following `slc` command:

```
$ slc create cli my-console-program
```

The program will be available to your terminal as a command:

```
$ ./my-console-program/bin/my-console-program
```

This command will output:

```
hello from my-console-program
```

Creating a simple web app

The `create` command supports several program types. The following generates the boilerplate for a web app.

```
$ slc create web my-app
```

By default, the generated web app will contain the following:

- `package.json`: dependencies and other package configuration
- `app.js`: app entry point and runtime configuration
- `public/`: static assets available over http
- `routes/`: route handler functions
- `views/`: templates for rendering html

Adding modules

A module is a JavaScript file with a single distinct purpose. It should accept some form of input, and generate meaningful output preferably with an asynchronous API. Well-designed modules often require a significant amount of boilerplate code. Use the following command to generate it:

```
$ slc create module my-module
```

This will generate a module in the lib folder `./lib/my-module.js`.

This command also supports automatically generating tests.

```
$ slc create module my-module --test
```

It also allows you to supply a stream type to implement.

```
$ slc create module my-module --stream transform
```

This will generate the following methods required by the specified streams2 interface:

- readable
- writeable
- duplex
- transform

Setting up a private package registry

Any team building a private-source Node.js application soon realizes the need for a private package registry for their Node modules.

Here is short guide on how to configure one yourself.

- Setting up the server
- Publishing packages
- Specifying package dependencies

Setting up the server

The first step is to setup a Reggie instance which will act as your package registry.

1. Prepare a server machine for the registry. The machine should be accessible by all team members and has Node.js installed.
2. Install Reggie as a global application

```
$ npm install -g reggie
```

3. Create a directory where Reggie will store all packages and other run-time data.

```
$ mkdir ~/reggie-data
```

4. Start the Reggie server (at the default port 8080)

```
$ reggie-server -d ~/reggie-data
```

You might want to extend your `init.d` scripts so that the Reggie server is automatically started after reboots.

For information on how to change the default port and other settings, see the Reggie manual.

Publishing packages

Add the following line to the `package.config` file in your private module:

```
"publishConfig": { "registry": "http://{reggie-host}:8080/" }
```

This setting tells the npm client to use your private registry for publishing.

Now you can publish the package in the usual way:

```
$ npm publish
```

Specifying package dependencies

The npm client does not support multiple registries (yet), but fortunately it can download packages from any URL.

Use the following command to install a 1.0.0 version of a private package named 'private-helpers':

```
$ npm install --save http://{reggie-host}:8080/package/private-helpers/1.0.0
```

This will download the package into node_modules folder and add a dependency entry into your `package.json` file:

```
"dependencies": {
  "private-helpers": "http://{reggie-host}:8080/package/private-helpers/1.0.0"
}
```

Reggie supports version wildcards too; for more information, see the Reggie documentation.

Clustering applications

See also the Cluster control API reference.

Overview

Clustering applications means using Node's support for running a cluster of identical workers, all receiving requests on the same port. It's possible to use the cluster module directly, but in keeping with the Node core design philosophy of providing fundamental mechanism, the cluster module has very basic functionality. Here, we will use the strong-cluster-control module to start a cluster and manage it from the command line.

Once you start the cluster controller in your app, it automatically maintains a set number of workers and listens on a control port, enabling cluster configuration at run-time.

strong-cluster-control is a module for run-time management of a node cluster.

It is an extension of the node cluster module, not a replacement, and works beside it to add the following features:

- runs `size` workers (optionally), and monitors them for unexpected death
- run-time control of cluster through command line and API
- soft shutdown as well as hard termination of workers
- throttles worker restart rate if they are exiting abnormally

It can be added to an existing application using the node cluster module without modifying how that application is currently starting up or using cluster, and still make use of additional features.

You can control clusters through:

- The `slc clusterctl` command.
- API calls on the module.
- StrongOps console; see Controlling an application cluster.

Installation

Follow the steps in Getting started to install the StrongLoop command-line tool. You don't have to create the sample application, but it's helpful to understand how StrongLoop Suite works.

Enter this command to install strong-cluster-control :

```
$ npm install strong-cluster-control
```

The slc clusterctl command

Use the `slc clusterctl` command to control a cluster at run-time; it provides the following sub-commands:

- status: reports the status of the cluster workers
- set-size: set cluster size to some number of workers
- disconnect: disconnect all workers
- fork: fork one worker

The `disconnect` and `fork` commands change the cluster size, so new workers will probably be started or stopped to return the cluster to the set size. These commands are primarily for testing and development. See slc clusterctl for complete documentation.

If your Node instances are under-utilized, and having less workers makes sense, set the size of the worker pool lower without taking your application down. Or, if your Node instances are 100% CPU-bound and you have free CPUs, you can increase the number of workers at runtime..

The cluster controller enables live upgrades of your workers. If you have updated your source code and want to restart all your workers, just disconnect the workers. As they exit, the controller will fork replacement workers, running the new code.

Example

In this example, you'll add two blocks of code to the sample blog app main file `app.js`. At the very top, after the following line:

```
, setup = require('./app-setup.js');
```

Add this code:

```
if(cluster.isMaster) {
  control.start({
    size: control.CPUS
  });
} else {
```

and at the very end of the file, as the very last line, add a single '}':

```
}
```

to match the else { you added above.

Next, install the strong-cluster-control module, and the app is ready to run:

```
$ npm install --save strong-cluster-control
$ node app
```

Now `app.js` runs as the cluster master, maintaining the cluster size. The default cluster size is determined according to the number of CPUs you have. However, do performance testing to confirm if this is the right choice for your application. If you decide you need more or fewer workers, you can change the number of workers at runtime.

In the sample-app directory, run:

```
$ slc clusterctl
worker count: 2
worker id 0: { pid: 7696 }
worker id 1: { pid: 7703 }
```

If you decide you want four workers, instead of two, enter these commands:

```
$ slc clusterctl set-size 4
$ slc clusterctl status
worker count: 4
worker id 0: { pid: 7696 }
worker id 1: { pid: 7703 }
worker id 2: { pid: 7705 }
worker id 3: { pid: 7707 }
```

For more in-depth examples, see the chat server example, and the in-source example.

Instantiating cluster control at runtime

To instantiate cluster-control:

```
var cluster = require('cluster');
var control = require('strong-cluster-control');

// global setup here...

control.start({
    size: control.CPUS
}).on('error', function(er) {
    console.error(er);
});

if(cluster.isWorker) {
    // do work here...
}
```

To control the cluster, if `my-server` is running in `/apps/`:

```
$ slc clusterctl --path /apps/my-server/clusterctl set-size 4
$ slc clusterctl --path /apps/my-server/clusterctl status
worker count: 4
worker id 0: { pid: 11454 }
worker id 1: { pid: 11471 }
worker id 2: { pid: 11473 }
worker id 3: { pid: 11475 }
```

For more in-depth examples, see the chat server example, and the in-source example.

Strong Cluster TLS Store

See also the Strong Cluster TLS Store reference.

Overview

Transport Layer Security (TLS) provides encrypted streams using sockets. *Strong Cluster TLS Store* implements a TLS session store using Node's native cluster messaging. It provides an easy solution for improving performance of Node's TLS/HTTPS server running in a cluster.

By adding hooks for the events 'newSession' and 'resumeSession' on the `tls.Server` object, Strong Cluster TLS Store enables clients to resume previous TLS sessions.

The performance of an HTTPS/TLS cluster depends on many factors:

- Node.js version (Version 0.11 implemented significant improvements to both TLS and cluster modules).
- Operating system platform.
- Whether clients support the SessionTicket TLS extension (RFC5077).
- How often the same HTTPS connection is reused for multiple requests.

You should therefore monitor the performance of your application and find out yourself how much extra speed is gained in your specific scenario (if any at all).

Installation

```
$ npm install strong-cluster-tls-store
```

Configuration

TLS server

```
var shareTlsSessions = require('strong-cluster-tls-store');

if (cluster.isMaster) {
  // Setup your master and fork workers.
} else {
  // Start a TLS server, configure it to share TLS sessions.
  var tlsOpts = { /* configure certificates, etc. */ }
  var server = tls.createServer(tlsOpts, connectionHandler);
  shareTlsSessions(server);
  server.listen(port);
  // etc.
}
```

HTTPS Server

`https.Server` implements the interface of `tls.Server`. The code to configure session sharing is the same.

```
 var shareTlsSessions = require('strong-cluster-tls-store');

if (cluster.isMaster) {
  // Setup your master and fork workers.
} else {
  // Start a TLS server, configure it to share TLS sessions.
  var httpsOpts = { /* configure certificates, etc. */ }
  var server = https.createServer(httpsOpts, requestHandler);
  shareTlsSessions(server);
  server.listen(port);
  // etc.
}
```

Connect and Express

Both Connect and Express require that the caller create an HTTP server. TLS session sharing follows the same pattern as for a plain HTTPS server.

```
 var express = require('express');
var shareTlsSessions = require('strong-cluster-tls-store');

if (cluster.isMaster) {
  // Setup your master and fork workers.
} else {
  // Start the server and configure it to share TLS sessions.

  var app = express();
  // configure the app

  var httpsOpts = { /* configure certificates, etc. */ }
  var server = https.createServer(httpsOpts, app);
  shareTlsSessions(server);

  server.listen(port);
  // etc.
}
```

Using multiple servers

To configure session sharing for multiple TLS/HTTPS servers, you must assign a unique namespace to each server.

```
shareTlsSessions(server1, 'server1');
shareTlsSessions(server2, 'server2');
```

Setting up the master process

The store requires that a shared-state server is running in the master process. The server is initialized automatically when you call `require()` for this module from the master. In the case that your master and workers have separate source files, you must explicitly require this module in your master source file. Optionally, you can call `setup()` to make it more obvious why you are loading a module that is not used anywhere else.

The following code in `master.js` configures the cluster and forks the workers.

```
var cluster = require('cluster');
require('strong-cluster-tls-store').setup();
```

Setting up the client

TLS session resumption may not occur without client configuration. For non-Node clients it is case-by-case. For example, many browsers attempt session resumption by default. With the Node.js client, session data from a successful connection must be explicitly copied to `opts.session` wh en making a new connection.

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

var opts = {
  port: 4433,
  host: 'localhost'
};

var initialConnection = tls.connect(opts, function() {
  // save the TLS session
  opts.session = this.getSession();

  // talk to the other side, etc.
});

var resumedConnection = tls.connect(opts, function() {
  // talk to the other side, etc.
});
```

As of Node.js v0.10.15 and v0.11.4, the HTTPS client reuses TLS sessions by default and the API does not provide an easy way how to enable it manually.

Using Strong Cluster TLS Store

The following example shows how to use Strong Cluster TLS Store with Cluster and Express.

```
'use strict';
var cluster = require('cluster');

if (cluster.isMaster) {
  // The cluster master executes this code

  require('strong-cluster-tls-store').setup();

  // Create a worker for each CPU
  var numCPUs = require('os').cpus().length;
  for (var i=0; i<numCPUs; i++)
    cluster.fork();

  cluster.on('online', function(worker) {
    console.log('Worker ' + worker.id + ' is online.');
  });

  cluster.on('exit', function(worker) {
    console.log('worker ' + worker.id + ' died');
  });

} else {
  // workers execute this code

  var express = require('express');
  var app = express();

  app.get('/hello', function(req, res) {
    res.json(200, {msg: 'hello'});
  });

  var fs = require('fs');
  var options = {
    key: fs.readFileSync('./private-key.pem'),
    cert: fs.readFileSync('./public-cert.pem')
  };

  var https = require('https');
  var server = https.createServer(options, app);
  require('strong-cluster-tls-store')(server, 'example-namespace');
  server.listen(8000);
}
```

TLS store API

- installSessionHandler
- setup

Module: strong-cluster-tls-store

installSessionHandler(tlsServer, namespace)

Enable TLS session resumption by installing listeners for events related to TLS sessions.

Arguments

Name	Type	Description

`tlsServer`	tls.Server	Instance of node's TLS server,
`namespace`	String	Optional namespace to distinguish between multiple TLS servers. The namespace must be unique within the application and same across all worker processes.

setup()

Documentation marker for explicit setup of the shared-state server in the master process. The initialization happens when this module is required, thus calling this function is entirely optional.

Strong Store for Cluster

See also the Strong Store for Cluster API reference.

Overview

When using a cluster, you have multiple processes with each handling requests. However, sometimes you need to keep state information locally across all the Node cluster processes. This is when you use Strong Store for Cluster.

Strong Store for Cluster enables you to share information across clustered Node processes in a key/value collection. The most common use is to implement sessions.

Session management

LoopBack, Connect, and Express applications operate over REST (Representational State Transfer) APIs. Since REST APIs are stateless, applications need a way to store state, for example, user authentication information. The solution is to store that data server side, give it an ID, and let clients know only the ID, often in a cookie. Sessions serve to identify the user's state.

How Strong Store for Cluster works

Strong Store places all data in a master process object where the key is the property name and the data is the value of the property. When you require the Strong Store Cluster module, the master process gets a different implementation of the interface than the worker. The master process API refers to the object storing the key/value pairs while the worker API uses `process.send()` to asynchronously request the key/value pair from the master process.

The API for the worker and the master is the same.

Example

```
 // require the collection, and give it a name
var collection = require('strong-store-cluster').collection('test');
var key = 'ThisIsMyKey';

// don't let keys expire, ever - values are seconds to expire keys
collection.configure({ expireKeys: 0 });

// set a key in the current collect to the object
collection.set(key, { a: 0, b: 'Hiya', c: { d: 99}}, function(err) {
  if (err) {
    console.error('There was an error in collection.set', err);
    return;
  }

  // now get the object we just set
  collection.get(key, function(err, obj) {
    if (err) {
      console.error('There was an error in collection.get.', err);
      return;
    }

    // You now have the object
    console.log('The object: ', obj);
  });
});
```

Strong store cluster API

View docs for strong-store-cluster in GitHub

store.collection(name)

Returns a Collection object which lets you share data between node processes.

Collection class

A `Collection` instance provides access to a shared key-value store shared by multiple node instances.

How collections are named and stored is determined by the storage backend. The `strong-store-cluster` implementation stores collections in the master process (if you're using cluster), and accepts any arbitrary string as a collection name.

A `Collection` object is also an `EventEmitter`.

Methods

collection.configure([options])

- `options` (`Object`) contains configurations options to be changed
- `expireKeys` (`Number`) seconds after which keys in this collection are to be expired.

Set configuration options for the collection.

Currently only one configurable option is supported: `expireKeys`. When set to a nonzero value, keys will automatically expire after they've not been read or updated for some time. The timeout is specified in seconds. There's no guarantee that the key will be discared after exactly that number of seconds

has passed. However keys will never be automatically deleted *sooner* than what the `expireKeys` setting allows.

It is perfectly legal to call the `configure` method from multiple node processes (e.g. both in a worker and in the master process). However you should be careful to set the *same* option values every time, otherwise the effect is undefined.

collection.get(key, callback)

- `key` (`String`) key to retrieve
- `callback` (`Function`) called when the value has been retrieved

Read the value associated with a particular key. The callback is called with two arguments, `(err, value)`. When the key wasn't found in the collection, it is automatically created and it's `value` is set to `undefined`.

collection.set(key, [value], [callback])

- `key` (`String`) key to set or update
- `value` (`object`) value to associate with the key
- `callback` (`Function`) called when the value has been retrieved

Set the value associated with `key`. The `value` must be either undefined or a value that can be serialized with JSON.stringify.

When the `value` parameter is omitted or set to `undefined`, the key is deleted, so effectively it's the same as calling `collection.del(key)`.

The `callback` function receives only one argument, `err`. When the callback is omitted, the master process does not send a confirmation after updating the key, and any errors are silently ignored.

collection.del(key, [callback])

- `key` (`String`) key to delete
- `callback` (`Function`) called when the value has been retrieved

Delete a key from the collection.

This operation is the equivalent of setting the key to `undefined`.

The `callback` function receives only one argument, `err`. When the callback is omitted, the master process does not send a confirmation after deleting the key, and any errors are silently ignored.

collection.acquire(key, callback)

- `key` (`String`) key to delete
- `callback` (`Function`) called when the key has been locked

Lock a key for exclusive read and write access.

The `acquire` methods waits until it can grab an exclusive lock on the specified key. When the lock is acquired, no other process can read, write or delete this particular key. When the lock is no longer needed, it should be relinquished with `keylock.release()`.

Three parameters are passed to the `callback` function: `(err, keylock, value)`. The `keylock` argument receives a `KeyLock` class instance, which lets you read and manipulate the key's value as well as eventually release the lock. The `value` argument is set to the initial value associated with the key.

Events

'error'

- `err` (`Error`)

The error event is emitted whenever an unrecoverable error is encountered.

KeyLock class

A `KeyLock` instance represents a key that has been locked. The `KeyLock` class implements methods that lets you manipulate the key and release the lock.

Methods

keylock.get()

- Returns: (`Object`) value that's currently associated with the key

This function returns the value that's currently associated with the locked key.

Initially this is the same as the `value` argument that was passed to the `collection.acquire()` callback, but it does immediately reflect changes that are made with `keylock.set()` and `keylock.del()`.

keylock.set([value])

Updates the value associated with a locked key.

The change isn't pushed back to the master process immediately; the change is committed when the lock is released again. The change however is reflected immediately in the return value from `keylock.get()`.

After the lock has been released, the key can no longer be updated through the `KeyLock` instance. Any attempt to do so will make it throw.

Setting the value to `undefined` marks the key for deletion, e.g. it's equivalent to `keylock.del()`.

keylock.del()

Mark a locked key for deletion. See `keylock.set()`.

keylock.release([callback])

Release the lock that protects a key. If the key was updated with `keylock.set()` or `keylock.del()`, these changes are committed.

When a lock has been released, it is no longer possible to manipulate the key using `KeyLock` methods. Releasing the lock twice isn't allowed either. The `get()` method will still work but it won't reflect any value changes that were made after releasing.

The `callback` function receives only one argument, `err`. When the callback is omitted, the master process does not send a confirmation after releasing the key, and any errors are silently ignored.

Strong Cluster Connect Store

> See also the Strong Cluster Connect Store API reference.

Overview

Strong Cluster Connect Store provides an easy way to use sessions in a clustered application. Each clustered worker is a separate Node.js process with its own memory space, however Strong Cluster Connect Store enables you to store sessions in a single location with low-latency access from all members of a Node cluster.

Strong Cluster Connect Store:

- Supports both Connect and Express frameworks.
- Has no dependencies on external services.

Installation

```
$ npm install strong-cluster-connect-store
```

Configuration

Connect is a *middleware framework* for Node.js. Connect's `createServer` method returns an object inheriting an extended version of `http.Server`. Connect's extensions make it easy to add a pipeline of functions in HTTP requests.

Configuration for Connect

Use the following code to configure Strong Cluster Connect Store for use with Connect:

```
var connect = require('connect');
var ClusterStore = require('strong-cluster-connect-store')(connect);

var app = connect();
app
  .use(connect.cookieParser())
  .use(connect.session({ store: new ClusterStore(), secret: 'keyboard cat' }));
```

Configuration for Express

Express is a web application framework that uses Connect; therefore, Strong Cluster Connect Store also works with Express. Use the following code to configure Strong Cluster Connect Store for use with Express:

```
var express = require('express');
var ClusterStore = require('strong-cluster-connect-store')(express);

var app = express();
app
  .use(express.cookieParser())
  .use(express.session({ store: new ClusterStore(), secret: 'keyboard cat' }));
```

Setting up the master process

```
// The master process only executes this code
var cluster = require('cluster');
var numCPUs = require('os').cpus().length;

require('strong-cluster-connect-store').setup();

// fork the workers
for (var i = 0; i < numCPUs; i++) {
  cluster.fork();
}
// workers and master run from this point onward

// setup the workers
if (cluster.isWorker) {
  [...]
}
```

Using Strong Cluster Connect Store

The following example assumes Strong Cluster Connect Store was setup for Express and the steps in "Setup for for the Master Process" were run.

```
'use strict';
var express = require('express');
var cluster = require('cluster');
var numCPUs = require('os').cpus().length;
var ClusterStore = require('strong-cluster-connect-store')(express);

if (cluster.isMaster) {
  // The cluster master executes this code

  ClusterStore.setup();

  // Create a worker for each CPU
  for (var i=0; i<numCPUs; i++) {
    cluster.fork();
  }

  cluster.on('online', function(worker) {
    console.log('Worker ' + worker.id + ' is online.');
  });

  cluster.on('exit', function(worker, code, signal) {
    console.log('worker ' + worker.id + ' died with signal',signal);
  });
} else {
  // The cluster workers execute this code

  var app = express();
  app.use(express.cookieParser());

  app.use(express.session(
    { store: new ClusterStore(), secret: 'super-cool' }
  ));

  app.get('/hello', function(req, res) {
    var msg;
    if (req.session.visited)
      msg = {msg: 'Hello again from worker '+cluster.worker.id};
    else
      msg = {msg: 'Hello from worker '+cluster.worker.id};

    req.session.visited = '1';
    res.json(200, msg);
  });
  app.listen(8080);
}
```

Strong cluster connect store API

Module: strong-cluster-connect-store

- ClusterStore.setup

module.exports(connect)

Return the `ClusterStore` constructor.

Arguments

Name	Type	Description
`connect`	Object	connect module as returned by `require('connect')`.

ClusterStore(options)

Initialize a ClusterStore object with the given `options`. See the strong-store-cluster documentation for information on the options.

Arguments

Name	Type	Description
`options`	Object	Options for the ClusterStore object.

clusterStore.get(sid, fn)

Fetch a session by an id and receive the session in the callback.

Arguments

Name	Type	Description
`sid`	String	A string id for the session.
`fn`	Function	

fn

Name	Type	Description
`err`	Error	if present, indicates an error condition.
`The`	value	data stored in the collection, could be any type.

clusterStore.set(sid, session, fn)

Commit the given `session` object associated with the given `sid` to the session store.

Arguments

Name	Type	Description
`sid`	String	A string id identifying the session.
`session`	Object	The session object.
`fn`	Function	

fn

Name	Type	Description
`err`	Error	If defined, indicates an error occured.

clusterStore.destroy(sid, fn)

Destroy the session associated with the given `sid`.

Arguments

Name	Type	Description
`sid`	String	A String with the id of the session.
`fn`	Function	

fn

Name	Type	Description
`err`	Error	If defined, indicates an error occured.

ClusterStore.setup

Same as `setup()` (see above).

Socket IO Store for Clusters

See also the Socket IO Store for Clusters API reference.

Overview

Socket IO Store for Clusters is an implementation of socket.IO store using Node's native cluster messaging. It provides an easy solution for running a socket.IO server in a Node cluster. Key features include:

- No dependencies on external services.
- Uses *your* version of socket.io.

Installation

```
$ npm install strong-cluster-socket.io-store
```

Using Socket IO Store

Socket.io's implementation has a race condition that allows the client to send the websocket request to another worker before that worker has processed the notification about a successful handshake. See socket.io#952.

You must enable session affinity (sticky sessions) in your load-balancer to get your socket.io server working in the cluster.

Configuration

The following code illustrates how to configure Socket IO Store for Clusters:

```
var io = require('socket.io');
var ClusterStore = require('strong-cluster-socket.io-store')(io);

if (cluster.isMaster) {
  // Setup your master and fork workers.
} else {
  // Start a socket.io server, configure it to use ClusterStore.
  io.listen(port, { store: new ClusterStore() });
  // etc.
}
```

Setting up the master process

The store requires that a shared-state server is running in the master process. The server is initialized automatically when you `require()` this

module from the master. In the case that your master and workers have separate source files, you must explicitly require this module in your master source file. Optionally, you can call `setup()` to make it more obvious why you are loading a module that is not used anywhere else.

```
// master.js

var cluster = require('cluster');
// etc.

require('strong-cluster-socket.io-store').setup();

// configure your cluster
// fork the workers
// etc.
```

Socket IO store API

- setup
- module.exports
- ClusterStore
- ClusterStore.setup

Module: strong-cluster-socket.io-store

setup()

Documentation marker for explicit setup of the shared-state server in the master process. The initialization happens when this module is required, thus calling this function is entirely optional.

module.exports(io)

Return the `ClusterStore` constructor.

Arguments

Name	Type	Description
`io`	Object	socket.io module as returned by `require('socket.io')`

ClusterStore(options)

Initialize ClusterStore with the given `options`.

ClusterStore is an implementation of socket.io store using node's native cluster messaging.

Arguments

Name	Type	Description
`options`	Object	

ClusterStore.setup

Same as `setup()` (see above).

Cluster controller API

Overview

The controller is exported by the `strong-cluster-control` module.

View docs for strong-cluster-control in GitHub

```
control = require('strong-cluster-control')
```

Methods

control.start([options],[callback])

Start the controller.

- `options`: {Object} An options object, see below for supported properties, no default, and options object is not required.
- `callback`: {Function} A callback function, it is set as a listener for the `'start'` event.

The options are:

- `size`: {Integer} Number of workers that should be running, the default is to *not* control the number of workers, see `setSize()`
- `env`: {Object} Environment properties object passed to `cluster.fork()` if the controller has to start a worker to resize the cluster, default is null.
- `addr`: {String or Integer} Address to listen on for control, defaults to first of `options.path`, `options.port`, or `control.ADDR` that is defined.
- `path`: {String} Path to listen on for control, no default.
- `port`: {Integer} Localhost port to listen on for control, may be necessary to use on Windows, no default.
- `shutdownTimeout`: {Milliseconds} Number of milliseconds to wait after shutdown before terminating a worker, the default is 5 seconds, see `.shutdown()`
- `terminateTimeout`: {Milliseconds} Number of milliseconds to wait after terminate before killing a worker, the default is 5 seconds, see `.terminate()`
- `throttleDelay`: {Milliseconds} Number of milliseconds to delay restarting workers after they are exiting abnormally. Abnormal is defined as as *not* suicide, see `worker.suicide` in cluster docs

For convenience during setup, it is not necessary to wrap `.start()` in a protective conditional `if(cluster.isMaster) {control.start()}`, when called in workers it quietly does nothing but call its callback.

The ‘start’ event is emitted after the controller is started.

control.stop([callback])

Stop the controller, after stopping workers (if the size is being controlled, see `setSize()`).

Remove event listeners that were set on the `cluster` module, and stop listening on the control port.

- `callback`: {Function} A callback function, it is set as a listener for the `'stop'` event.

The ‘stop’ event is emitted after the controller is stopped.

When there are no workers or listeners, node will exit, unless the application has non-cluster handles open. Open handles can be closed in the ‘stop’ event callback to allow node to shutdown gracefully, or `process.exit()` can be called, as appropriate for the application.

control.restart()

Restart workers one by one, until all current workers have been restarted.

This can be used to do a rolling upgrade, if the underlying code has changed.

Old workers will be restarted only if the last worker to be restarted stays alive for more than `throttleDelay` milliseconds, ensuring that the current workers will not all be killed while the new workers are failing to start.

control.loadOptions([defaults])

Load options from configuration files, environment, or command line.

- `defaults`: {Object} Default options, see `start()` for description of supported options.

An options object is returned that is suitable for passing directly to `start()`. How you use it is up to you, but it can be conveniently used to implement optionally clustered applications, ones that run unclustered when deployed as single instances, perhaps behind a load balancer, or that can be deployed as a node cluster.

Here is an example of the above usage pattern:

```
// app.js
var control = require('strong-cluster-control');
var options = control.loadOptions();
if(options.clustered && options.isMaster) {
    return control.start(options);
}

// Server setup... or any work that should be done in the master if
// the application is not being clustered, or in the worker if it
// is being clustered.
```

The options values are derived from the following configuration sources:

- command line arguments (parsed by optimist): if your app ignores unknown options, options can be set on the command line, ex. `node app.js --size=2`
- environment variables prefixed with `cluster_`: note that the variable must be lower case, ex. `cluster_size=2 node app.js`
- `.clusterrc` in either json or ini format: can be in the current working directory, or any parent paths
- `defaults`: as passed in, if any

This is actually a subset of the possible locations, the rc module is used with an `appname` of "cluster", see it's documentation for more information.

Supported values are those described in `start()`, with a few extensions for `size`, which may be one of:

1. `0` - `N`, a positive integer, cluster size to maintain, as for `start()`
2. `"default"`, or a string containing `"cpu"`, this will be converted to the number of cpus, see `control.CPUS`
3. `"off"`, or anything else that isn't one of the previous values will indicate a preference for *not* clustering

The returned options object will contain the following fields that are not options to `start()`:

- clustered: false if size is 0 (after above conversions), true if a cluster size was specified
- isMaster, isWorker: identical to the properties of the same name in the cluster module

The combination of the above three allows you to determine if you are a worker, or not, and if you are a master, if you should start the cluster control module, or just start the server if clustering was not requested.

control.status()

Returns the current cluster status similar to what's sent to `clusterctl status`. Its properties include:

- `master`: {Object}
- `pid`: The pid of the Master process.
- `workers`: {Array} An Array of Objects containining the following properties:
    - `id`: The id of the Worker within the Master.
    - `pid`: The pid of the Worker process.

control.setSize(N)

Set the size of the cluster.

- `N`: {Integer or null} The size of the cluster is the number of workers that should be maintained online. A size of `null` clears the size, and disables the size control feature of the controller.

The cluster size can be set explicitly with `setSize()`, or implicitly through the `options` provided to `.start()`, or through the control channel.

The size cannot be set until the controller has been started, and will not be maintained after the cluster has stopped.

Once set, the controller will listen on cluster `fork` and `exit` events, and resize the cluster back to the set size if necessary. After the cluster has been resized, the 'resize' event will be emitted.

When a resize is necessary, workers will be started or stopped one-by-one until the cluster is the set size.

Cluster workers are started with `cluster.fork(control.options.env)`, so the environment can be set, but must be the same for each worker. After a worker has been started, the 'startWorker' event will be emitted.

Cluster workers are stopped with `.shutdown()`. After a worker has been stopped, the 'stopWorker' event will be emitted.

control.shutdown(id)

Disconnect worker `id` and take increasingly agressive action until it exits.

- `id` {Number} Cluster worker ID, see `cluster.workers` in cluster docs

The effect of disconnect on a worker is to close all the servers in the worker, wait for them to close, and then exit. This process may not occur in a timely fashion if, for example, the server connections do not close. In order to gracefully close any open connections, a worker may listen to the `SHUTDOWN` message, see `control.cmd.SHUTDOWN`.

Sends a `SHUTDOWN` message to the identified worker, calls `worker.disconnect()`, and sets a timer for `control.options.shutdownTimeout`. If the worker has not exited by that time, calls `.terminate()` on the worker.

control.terminate(id)

Terminate worker `id`, taking increasingly aggressive action until it exits.

- `id` {Number} Cluster worker ID, see `cluster.workers` in cluster docs

The effect of sending SIGTERM to a node process should be to cause it to exit. This may not occur in a timely fashion if, for example, the process is ignoring SIGTERM, or busy looping.

Calls `worker.kill("SIGTERM")` on the identified worker, and sets a timer for `control.options.terminateTimeout`. If the worker has not exited by that time, calls `worker.("SIGKILL")` on the worker.

Properties

control.options

A copy of the options set by calling `.start()`.

It will have any default values set in it, and will be kept synchronized with changes made via explicit calls, such as to `.setSize()`.

Visible for diagnostic and logging purposes. Do *not* modify the options directly.

control.cmd.SHUTDOWN

- {String} `'CLUSTER_CONTROL_shutdown'`

The `SHUTDOWN` message is sent by `.shutdown()` before disconnecting the worker, and can be used to gracefully close any open connections before the `control.options.shutdownTimeout` expires.

All connections will be closed at the TCP level when the worker exits or is terminated, but this message gives the opportunity to close at a more application-appropriate time, for example, after any outstanding requests have been completed.

The message format is:

```
{ cmd: control.cmd.SHUTDOWN }
```

It can be received in a worker by listening for a `'message'` event with a matching `cmd` property:

```
process.on('message', function(msg) {
    if(msg.cmd === control.cmd.SHUTDOWN) {
        // Close any open connections as soon as they are idle...
    }
});
```

control.ADDR

`'clusterctl'`, the default address of the control server, if none other are specified through `options`.

control.CPUS

The number of CPUs reported by node's `os.cpus().length`, this is a good default for the cluster size, in the absence of application specific analysis of what would be an optimal number of workers.

Events

'start'

Event emitted after control has started. Control is considered started after the 'listening' event has been emitted.

Starting of workers happens in the background, if you are specifically interested in knowing when all the workers have started, see the 'resize' event.

'stop'

Event emitted after control has stopped, see `.stop()`.

'error'

- {Error Object}

Event emitted when an error occurs. The only current source of errors is the control protocol, which may require logging of the errors, but should not effect the operation of the controller.

'setSize'

- {Integer} size, the number of workers requested (will always be the same as `cluster.options.size`)

Event emitted after `setSize()` is called.

'resize'

- {Integer} size, the number of workers now that resize is complete (will always be the same as `cluster.options.size`)

Event emitted after a resize of the cluster is complete. At this point, no more workers will be forked or shutdown by the controller until either the size is changed or workers fork or exit, see `setSize()`.

'startWorker'

- `worker` {Worker object}

Event emitted after a worker which was started during a resize comes online, see the node API documentation for description of `worker` and "online".

'startRestart'

- `workers` {Array of worker IDs} Workers that are going to be restarted.

Event emitted after `restart()` is called with array of worker IDs that will be restarted.

'restart'

Event emitted after after all the workers have been restarted.

'stopWorker'

- `worker` {Worker object}
- `code` {Number} the exit code, if it exited normally.
- `signal` {String} the name of the signal if it exited due to signal

Event emitted after a worker which was shutdown during a resize exits, see the node API documentation for a description of `worker`.

The values of `code` and `signal`, as well as of `worker.suicide`, can be used to determine how gracefully the worker was stopped. See `.terminate()`.

'listening'

Event emitted when the controller has been bound after calling `server.listen`.

'connection'

- {Socket object} The connection object

Event emitted when a new connection is made. `socket` is an instance of `net.Socket`.

Strong Task Emitter

See also the Strong Task Emitter API reference.

Overview

Strong Task Emitter enables you to perform a number of tasks recursively and in parallel. For example, reading all the files in a nested set of directories. It has built-in support for domains by inheriting directly from `EventEmitter`.

Installation

```
$ npm install strong-task-emitter
```

Examples

The following example shows the basic API for a TaskEmitter.

```
var TaskEmitter = require('../');
var request = require('request');
var results = [];

var te = new TaskEmitter();

te
  .task('request', request, 'http://google.com')
  .task('request', request, 'http://yahoo.com')
  .task('request', request, 'http://apple.com')
  .task('request', request, 'http://youtube.com')
  // listen for when a task completes by providing the task name
  .on('request', function (url, res, body) {
    results.push(Buffer.byteLength(body));
  })
  .on('progress', function (status) {
    console.log(((status.total - status.remaining) / status.total) * 100 + '%',
'complete');
  })
  .on('error', function (err) {
    console.log('error', err);
  })
  .on('done', function () {
    console.log('Total size of all homepages', results.reduce(function (a, b) {
      return a + b;
    }), 'bytes');
  });
```

The next example highlights how to use TaskEmitter to simplify recursive asynchronous operations. The following code recursively walks a social network over HTTP. All requests run in parallel.

```
var TaskEmitter = require('../');
var request = require('request');
var socialNetwork = [];

var te = new TaskEmitter();

te
  // specify a task name
  .task('friends', fetch, 'me')
  .on('friends', function (user, url) {
    if(url !== 'me') {
      socialNetwork.push(user);
    }

    user.friendIds.forEach(function (id) {
      this.task('friends', fetch, 'users/' + id)
    }.bind(this));
  })
  .on('done', function () {
    console.log('There are a total of %n people in my network', socialNetwork.length);
  });

function fetch(url, fn) {
  request({
    url: 'http://my-api.com/' + url,
    json: true,
    method: 'GET'
  }, fn);
}
```

Extending TaskEmitter

`TaskEmitter` is designed to be a base class which you can easily extend with sub-classes. The following example shows a class that inherits from `TaskEmitter` and provides recursive directory walking and file loading.

```
var TaskEmitter = require('../');
var fs = require('fs');
var path = require('path');
var inherits = require('util').inherits;

function Loader() {
  TaskEmitter.call(this);

  this.path = path;
  this.files = {};

  this.on('readdir', function (p, files) {
    files.forEach(function (f) {
      this.task(fs, 'stat', path);
    }.bind(this));
  });

  this.on('stat', function (file, stat) {
    if(stat.isDirectory()) {
      this.task(fs, 'readdir', file);
    } else {
      this.task(fs, 'readFile', file, path.extname(file) === '.txt' ? 'utf-8' : null);
    }
  });

  this.on('readFile', function (path, encoding, data) {
    this.files[path] = data;
  });
}

inherits(Loader, TaskEmitter);

Loader.prototype.load = function (path, fn) {
  if(fn) {
    // error events are handled if a task callback ever is called
    // with a first argument that is not falsy
    this.on('error', fn);

    // once all tasks are complete the done event is emitted
    this.on('done', function () {
      fn(null, this.files);
    });
  }

  this.task(fs, 'readdir', path);
}

// usage
var l = new Loader();

l.load('sample-files', function (err, files) {
  console.log(err || files);
});
```

Strong MQ

Overview

Strong MQ is an abstraction layer over common message distribution patterns, and several different message queue implementations, including cluster-native messaging.

It allows applications to be written against a single message queue style API, and then deployed either singly, or as a cluster, with deploy-time configuration of the messaging provider. Providers include native node clustering, allowing no-dependency deployment during test and development. Support for other providers is on-going, and 3rd parties will be able to add pluggable support for new message queue platforms.

Message Patterns

- **Work queue**: published messages are delivered to a single subscriber, common when distributing work items that should be processed by a single worker
- **Topic**: published messages are delivered to all subscribers, each message is associated with a "topic", and subscribers can specify the topic patterns they want to receive
- **RPC**: published messages are delivered to a single subscriber, and a associated response is returned to the original publisher (TBD)

Installation

```
$ npm install strong-mq
```

Example

An example of connecting to a server and listening on a work queue:

```
var connection = require('strong-mq')
    .create('amqp://localhost')
    .open();

var push = connection.createPushQueue('todo-items');
push.publish({job: 'clean pool'});

var pull = connection.createPullQueue('todo-items');
pull.subscribe(function(msg) {
    console.log('TODO:', msg);
    connection.close();
});
```

Known error

If you get the **"Multiple Versions of strong-mq Being Initialized"** assert during require of strong-mq about multiple versions being initialized, then some of the modules you are depending on use strong-mq, but do not specify it as a peerDependency. See strongloop/strong-cluster-connect-store as an example of how to correctly specify a dependency on strong-mq in a module. An application can depend on strong-mq with a normal dependency.

Event: 'error'

Errors may be emitted as events from either a connection or a queue. The nature of the errors emitted depends on the underlying provider.

Messages

Message objects can be either an `Object` or `Array`, transmitted as JSON, or a `String` or `Buffer`, transmitted as data.

Queues

Queues are closed when they are empty and have no users. They might or might not be persistent across restarts of the queue broker, depending on the provider.

Messaging protocol providers

Strong MQ supports the following protocol providers:

- Native
- AMQP
- STOMP

Native

The NativeConnection uses the built-in cluster module to facilitate the strong-mq API. It's designed to be the first adapter people use in early development, before they get whatever system they will use for deployment up and running.

It has no options.

The URL format is:

```
native:[//]
```

AMQP

Support for RabbitMQ using the AMQP protocol. This provider is based on the node-amqp module, see its documentation for more information.

The options (except for `.provider`) or url is passed directly to node-amqp, supported options are:

- `host` {String} Hostname to connect to, defaults to `'localhost'`
- `port` {String} Port to connect to, defaults to `5672`
- `login` {String} Username to authenticate as, defaults to `'guest'`
- `password` {String} Password to authenticate as, defaults to `'guest'`
- `vhost` {String} Vhost, defaults to `'/'`

The URL format for specifying the options above is:

```
amqp://[login][:password][@]host[:port][/vhost]
```

Note that the `host` is mandatory when using a URL.

Note that node-amqp supports RabbitMQ 3.0.4, or higher. In particular, it will *not* work with RabbitMQ 1.8.1 that is packaged with Debian 6, see the upgrade instructions.

STOMP

Support for ActiveMQ using the STOMP protocol. This provider is based on the node-stomp-client module.

The options are:

- `host` {String} Hostname to connect to, defaults to `'127.0.0.1'`
- `port` {String} Port to connect to, defaults to `61613`
- `login` {String} Username to authenticate as, defaults to none
- `password` {String} Password to authenticate as, defaults to none

The URL format for specifying the options above is:

```
stomp://[login][:password][@]host[:port]
```

Note that the `host` is mandatory when using a URL.

ActiveMQ ships with an example configuration sufficient to run the strong-mq unit tests.

Note that node-stomp-client has been tested only with Active MQ 5.8.0. It can be installed from apache, and run as:

```
activemq console xbean:activemq-stomp.xml
```

Strong MQ API

Message Patterns

View docs for strong-mq in GitHub

- work queue: published messages are delivered to a single subscriber, common when distributing work items that should be processed by a single worker
- topic: published messages are delivered to all subscribers, each message is associated with a “topic”, and subscribers can specify the topic patterns they want to receive
- rpc: published messages are delivered to a single subscriber, and a associated response is returned to the original publisher (TBD)

Event: ‘error’

Errors may be emitted as events from either a connection or a queue. The nature of the errors emitted depends on the underlying provider.

Connections

slmq.create([options|url])

Returns a connection object for a specific provider, configuration can be created using a options object, or a url:

- `options` {Object}
- `url` {provider://...}

If `create()` is called with no arguments, the native provider will be used.

Supported providers are:

- `'amqp'`: RabbitMQ
- `'native'`: Cluster-native messaging

Supported options, other than `provider`, depend on the provider:

- `provider` {String} Mandatory name of provider, such as `'amqp'`
- `host` {String} Name of host to connect to (if supported by provider)
- `port` {String} Port to connect to (if supported by provider)
- `...` As supported by the provider

Example of creating an amqp connection, using an options object:

```
connection = clustemq.create({
    provider: 'amqp',
    host: 'localhost',
    user: 'guest',
});
```

Example of declaring amqp, using a URL:

```
connection = clusermq.create('amqp://guest@localhost');
```

connection.provider {String}

Property is set to the name of the provider.

connection.open()

Opens a connection.

Example:

```
connection.open().on('error', function () {
  // ... handle error
});
```

connection.close([callback])

Callsback when connection has been closed.

Work queues (push/pull)

Queue life-time

Queues are closed when they are empty and have no users. They might or might not be persistent across restarts of the queue broker, depending on the provider.

Messages

Message objects can be either an `Object` or `Array`, transmitted as JSON, or a `String` or `Buffer`, transmitted as data.

connection.createPushQueue()

Return a queue for publishing work items.

push.publish(msg)

Publish a msg to a push queue.

- `msg` {Object} Message to publish to the queue

connection.createPullQueue()

Return a queue for subscribing to work items.

pull.subscribe([listener])

Listen for messages on a work queue.

`listener` is optional, it will be added as a listener for the `'message'` event if provided.

queue.close()

Close the queue.

queue.name {String}

Name used to create queue.

queue.type {String}

Either ‘push’, or ‘pull’.

Event: ‘message’

Event is emitted when a subcribed pull queue receives a message.

- `msg` {Object} Message pulled off the queue

Topic queue (pub/sub)

Topics are dot-separated alphanumeric (or `'_'`) words. Subscription patterns match leading words.

connection.createPubQueue()

Return a queue for publishing on topics.

pub.publish(msg, topic)

- `msg` {Object} Message to publish onto the queue
- `topic` {String} Topic of message, default is `''`

connection.createSubQueue()

Return a queue for subscribing to topics.

sub.subscribe(pattern[, listener])

Listen for messages matching pattern on a topic queue.

- `pattern` {String} Pattern of message, may contain wildcards, default is `''`

`listener` is optional, it will be added as a listener for the `'message'` event if provided. Add your listener to the `'message'` event directly when subscribing multiple times, or all your listeners will be called for all messages.

Example of subscribing to multiple patterns:

```
sub.subscribe('that.*')
  .subscribe('this.*')
  .on('message', function (msg) { ... });
```

Example of subscribing to a single pattern, and providing a listener:

```
sub.subscribe('other.*', function (msg) { ... });
```

queue.close()

Close the queue.

queue.name {String}

Name used to create queue.

queue.type {String}

Either ‘pub’, or ‘sub’.

Event: ‘message’

Event is emitted when a subcribed pull queue receives a message.

- `msg` {Object} Message pulled off the queue

Provider: NATIVE

The NativeConnection uses the built-in cluster module to facilitate the strong-mq API. It’s designed to be the first adapter people use in early development, before they get whatever system they will use for deployment up and running.

It has no options.

The URL format is:

```
native:[//]
```

Provider: AMQP

Support for RabbitMQ using the AMQP protocol. This provider is based on the node-amqp module, see its documentation for more information.

The options (except for `.provider`) or url is passed directly to node-amqp, supported options are:

- `host` {String} Hostname to connect to, defaults to `'localhost'`
- `port` {String} Port to connect to, defaults to `5672`
- `login` {String} Username to authenticate as, defaults to `'guest'`

- `password` {String} Password to authenticate as, defaults to `'guest'`
- `vhost` {String} Vhost, defaults to `'/'`

The URL format for specifying the options above is:

```
amqp://[login][:password][@]host[:port][/vhost]
```

Note that the `host` is mandatory when using a URL.

Note that node-amqp supports RabbitMQ 3.0.4, or higher. In particular, it will *not* work with RabbitMQ 1.8.1 that is packaged with Debian 6, see the upgrade instructions.

Provider: STOMP

Support for ActiveMQ using the STOMP protocol. This provider is based on the node-stomp-client module.

The options are:

- `host` {String} Hostname to connect to, defaults to `'127.0.0.1'`
- `port` {String} Port to connect to, defaults to `61613`
- `login` {String} Username to authenticate as, defaults to none
- `password` {String} Password to authenticate as, defaults to none

The URL format for specifying the options above is:

```
stomp://[login][:password][@]host[:port]
```

Note that the `host` is mandatory when using a URL.

ActiveMQ ships with an example configuration sufficient to run the strong-mq unit tests.

Note that node-stomp-client has been tested only with Active MQ 5.8.0. It can be installed from apache,
and run as:

```
activemq console xbean:activemq-stomp.xml
```

Strong Remoting

See also the Strong Remoting API reference.

Overview

Objects (and, therefore, data) in Node applications commonly need to be accessible by other Node processes, browsers, and even mobile clients. Strong remoting:

- Makes local functions remotable, exported over adapters
- Supports multiple transports, including custom transports
- Manages serialization to JSON and deserialization from JSON
- Supports multiple client SDKs, including mobile clients

Client SDK support

For higher-level transports, such as REST and Socket.IO, existing clients will work well. If you want to be able to swap out your transport, use one of our supported clients. The same adapter model available on the server applies to clients, so you can switch transports on both the server and

all clients without changing your application-specific code.

Installation

```
$ npm install strong-remoting
```

Quick start

The following example illustrates how to set up a basic `strong-remoting` server with a single remote method, `user.greet`.

```
// Create a collection of remote objects.
var remotes = require('strong-remoting').create();

// Export a `user` object.
var user = remotes.exports.user = {
  greet: function (str, callback) {
    callback(null, str + ' world');
  }
};

// Share the `greet` method.
user.greet.shared = true;
user.greet.accepts = { arg: 'str' };
user.greet.returns = { arg: 'msg' };

// Expose it over the REST transport.
require('http')
  .createServer(remotes.handler('rest'))
  .listen(3000);
```

Then, invoke `user.greet()` easily with `curl` (or any HTTP client)!

```
$ curl http://localhost:3000/user/greet?str=hello
{
  "msg": "hello world"
}
```

Concepts

Remote objects

Most Node applications expose a remotely-available API. Strong-remoting enables you to build your app in vanilla JavaScript and export remote objects over the network the same way you export functions from a module. Since they're just plain JavaScript objects, you can always invoke methods on your remote objects locally in JavaScript, whether from tests or other, local objects.

Remote object collections

Collections that are the result of `require('strong-remoting').create()` are responsible for binding their remote objects to transports, allowing you to swap out the underlying transport without changing any of your application-specific code.

Adapters

Adapters provide the transport-specific mechanisms to make remote objects (and collections thereof) available over their transport. The REST adapter, for example, handles an HTTP server and facilitates mapping your objects to RESTful resources. Other adapters, on the other hand, might provide a less opionated, RPC-style network interface. Your application code doesn't need to know what adapter it's using.

Hooks

Hooks enable you to run code before remote objects are constructed or methods on those objects are invoked. For example, you can prevent actions based on context (HTTP request, user credentials, and so on).

```
// Do something before our `user.greet` example, earlier.
remotes.before('user.greet', function (ctx, next) {
  if((ctx.req.param('password') || '').toString() !== '1234') {
    next(new Error('Bad password!'));
  } else {
    next();
  }
});

// Do something before any `user` method.
remotes.before('user.*', function (ctx, next) {
  console.log('Calling a user method.');
  next();
});

// Do something before a `dog` instance method.
remotes.before('dog.prototype.*', function (ctx, next) {
  var dog = this;
  console.log('Calling a method on "%s".', dog.name);
  next();
});

// Do something after the `speak` instance method.
// NOTE: you cannot cancel a method after it has been called.
remotes.after('dog.prototype.speak', function (ctx, next) {
  console.log('After speak!');
  next();
});

// Do something before all methods.
remotes.before('**', function (ctx, next, method) {
  console.log('Calling:', method.name);
  next();
});

// Modify all returned values named `result`.
remotes.after('**', function (ctx, next) {
  ctx.result += '!!!';
  next();
});
```

See the before-after example for more info.

Streams

strong-remoting supports methods that expect or return `Readable` and `Writeable` streams. This allows you to stream raw binary data such as files over the network without writing transport-specific behaviour.

Example

The following example exposes a method of the `fs` Remote Object,`fs.createReadStream`, over the REST adapter:

```
// Create a Collection.
var remotes = require('strong-remoting').create();

// Share some fs module code.
var fs = remotes.exports.fs = require('fs');

// Specifically export the `createReadStream` function.
fs.createReadStream.shared = true;

// Describe the arguments.
fs.createReadStream.accepts = {arg: 'path', type: 'string'};

// Describe the stream destination.
fs.createReadStream.http = {
  // Pipe the returned `Readable` stream to the response's `Writable` stream.
  pipe: {
    dest: 'res'
  }
};

// Expose the Collection over the REST Adapter.
require('http')
  .createServer(remotes.handler('rest'))
  .listen(3000);
```

Then you can invoke `fs.createReadStream()` using `curl` as follows:

```
$ curl http://localhost:3000/fs/createReadStream?path=some-file.txt
```

Strong remoting API

Module: strong-remoting

RemoteObjects(options)

Create a new `RemoteObjects` with the given `options`.

```
var remoteObjects = require('strong-remoting').create();
```

Arguments

Name	Type	Description
`options`	Object	

remoteObjects.adapter(name)

Get an adapter by name.

Arguments

Name	Type	Description
`name`	String	The adapter name

remoteObjects.classes()

Get all classes.

remoteObjects.findMethod(methodString)

Find a method by its string name.

Example Method Strings:

- `MyClass.prototype.myMethod`
- `MyClass.staticMethod`
- `obj.method`

Arguments

Name	Type	Description
`methodString`	String	

remoteObjects.methods()

List all methods.

remoteObjects.toJSON()

Get as JSON.

remoteObjects.before(methodMatch, hook)

Execute the given function before the matched method string.

Examples:

```
// Do something before our `user.greet` example, earlier.
remotes.before('user.greet', function (ctx, next) {
  if((ctx.req.param('password') || '').toString() !== '1234') {
    next(new Error('Bad password!'));
  } else {
    next();
  }
});

// Do something before any `user` method.
remotes.before('user.*', function (ctx, next) {
  console.log('Calling a user method.');
  next();
});

// Do something before a `dog` instance method.
remotes.before('dog.prototype.*', function (ctx, next) {
  var dog = this;
  console.log('Calling a method on "%s".', dog.name);
  next();
});
```

Arguments

Name	Type	Description
`methodMatch`	String	The glob to match a method string
`hook`	Function	

hook

Name	Type	Description
ctx	Context	The adapter specific context
next	Function	Call with an optional error object
method	SharedMethod	The SharedMethod object

remoteObjects.after(methodMatch, hook)

Execute the given hook function after the matched method string.

Examples:

```
// Do something after the `speak` instance method.
// NOTE: you cannot cancel a method after it has been called.
remotes.after('dog.prototype.speak', function (ctx, next) {
  console.log('After speak!');
  next();
});

// Do something before all methods.
remotes.before('**', function (ctx, next, method) {
  console.log('Calling:', method.name);
  next();
});

// Modify all returned values named `result`.
remotes.after('**', function (ctx, next) {
  ctx.result += '!!!';
  next();
});
```

Arguments

Name	Type	Description
methodMatch	String	The glob to match a method string
hook	Function	

hook

Name	Type	Description
ctx	Context	The adapter specific context
next	Function	Call with an optional error object
method	SharedMethod	The SharedMethod object

remoteObjects.invokeMethodInContext(, ,)

Invoke the given shared method using the supplied context. Execute registered before/after hooks.

Arguments

Name	Type	Description
	ctx	
	method	
	cb	

Running Node Inspector

This article will explain how to debug the sample blog application that comes as a part of StrongLoop Suite distribution.

Run the sample application blog system

For information on creating and running the sample application, see Sample application: blog system.

Starting the debugger

Start the blog application with the following command, from the root directory of the blog application (where package.json resides):

```
$ node app.js
```

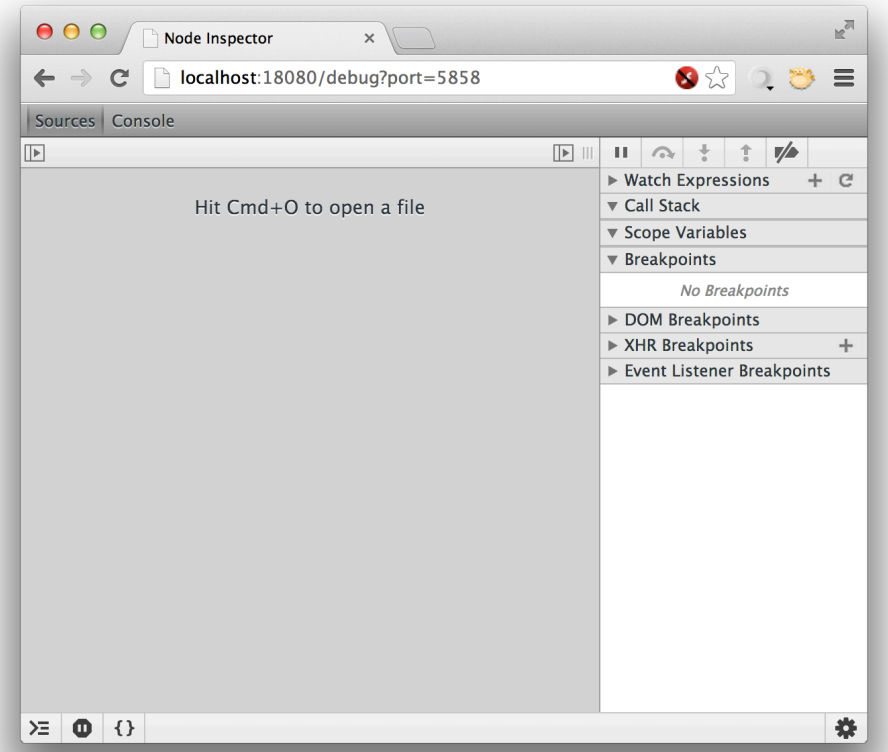
You can run the application in a debugger as follows:

```
$ slc debug app.js
```

This command does the following:

1. Starts the application in debug mode.
2. Starts Node Inspector.
3. Opens Node Inspector in your default browser.

Node Inspector works only in the Google Chrome browser at the moment. If you are using a different browser, you will have to reopen Node Inspector page in Chrome.

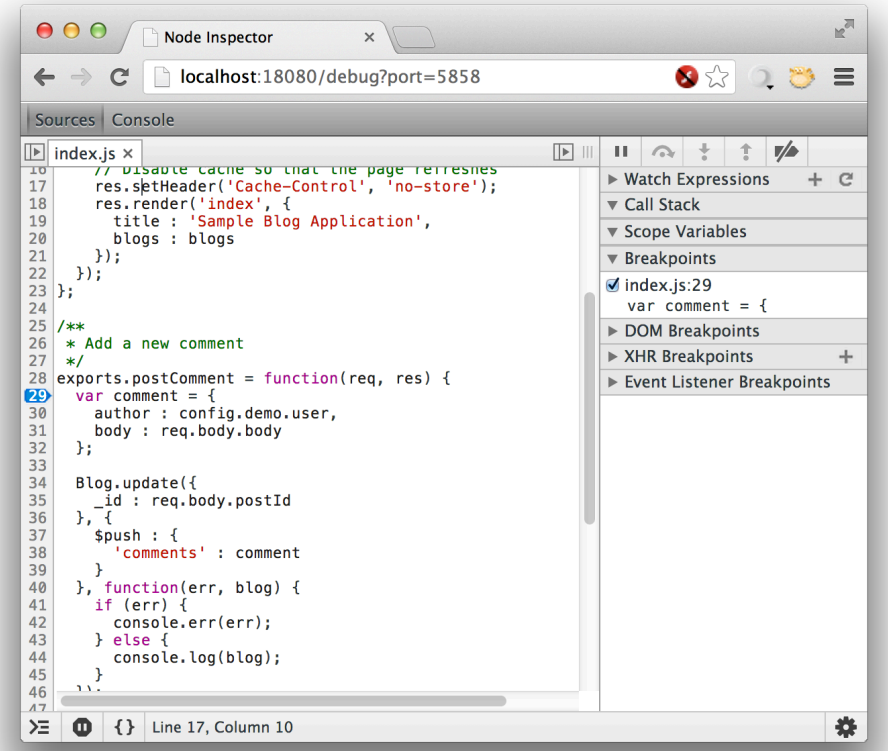


Working with Node Inspector

Set a breakpoint to inspect what's going on inside the blog application as followd:

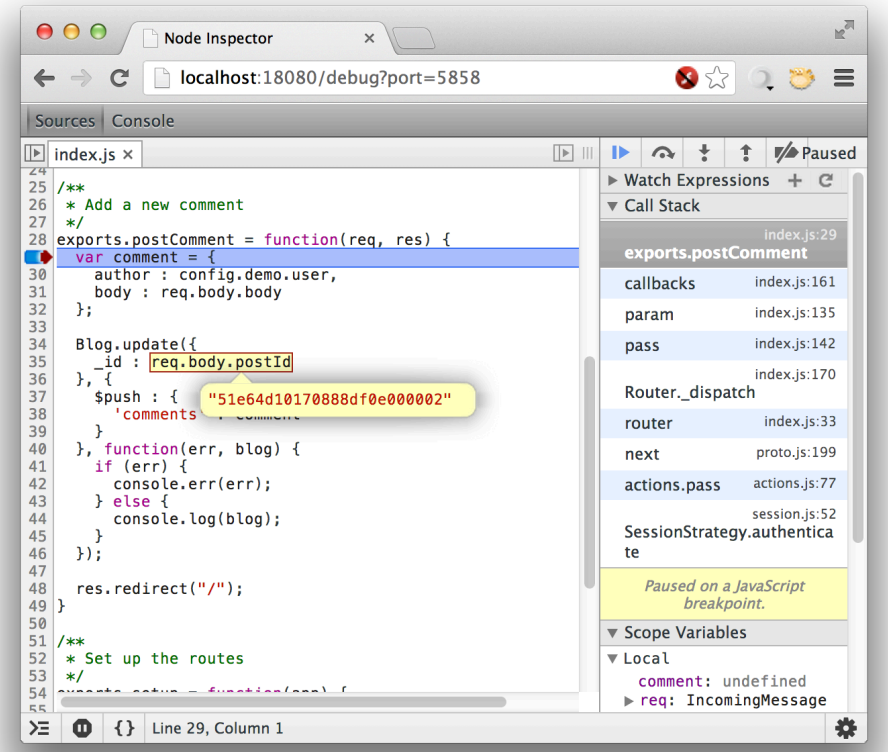
1. Click on the "Show navigator" icon in the upper-left corner to see a tree-list of all blog source files.
2. Expand "routes" folder and double -click on "index.js".
3. Click on line number 29 to set a breakpoint at the beginning of `postComment` function.

You should see something like this:



Open the blog application in a new tab and submit a comment. You can see that the page is waiting for the server to respond. This is because the server process is paused on our breakpoint.

Switch to Node Inspector's tab in the browser to inspect the fields of the incomming http request. Leave your mouse over a variable or a property to see it's value.



You can step through javascript statements by pressing F10. Use F8 to resume script execution after you are done.

Check out the Chrome Developer Tools Guide for a walkthrough of other debugger features. (Remember Node-Inspector is based on Chrome Developer Tools and most features work exactly the same.)

Set breakpoints in files not yet loaded

You can set breakpoints on files that have yet to be loaded. Here's how:

- Run mocha unit-test in any project with –debug-brk option.
- Launch node-inspector.
- Look at source files of your unit-tests and set breakpoints inside them.
- Resume execution when you have all breakpoints set up
- Wait for debugger to hit your first breakpoint.

Breakpoints and uncaught exceptions

Node-Inspector now restores breakpoints after restarting an application. iAdditionally, Node-inspector remembers your breakpoints in the browser's local storage (HTML5). When you restart the debugger process, or start debugging the same application after several days – your breakpoints are restored!

You can now break on uncaught exceptions. Anoter feature is integration with domains – exceptions handled by domain's error handler are still considered as uncaught. Please note this requires Node.js v0.11.3 or greater.

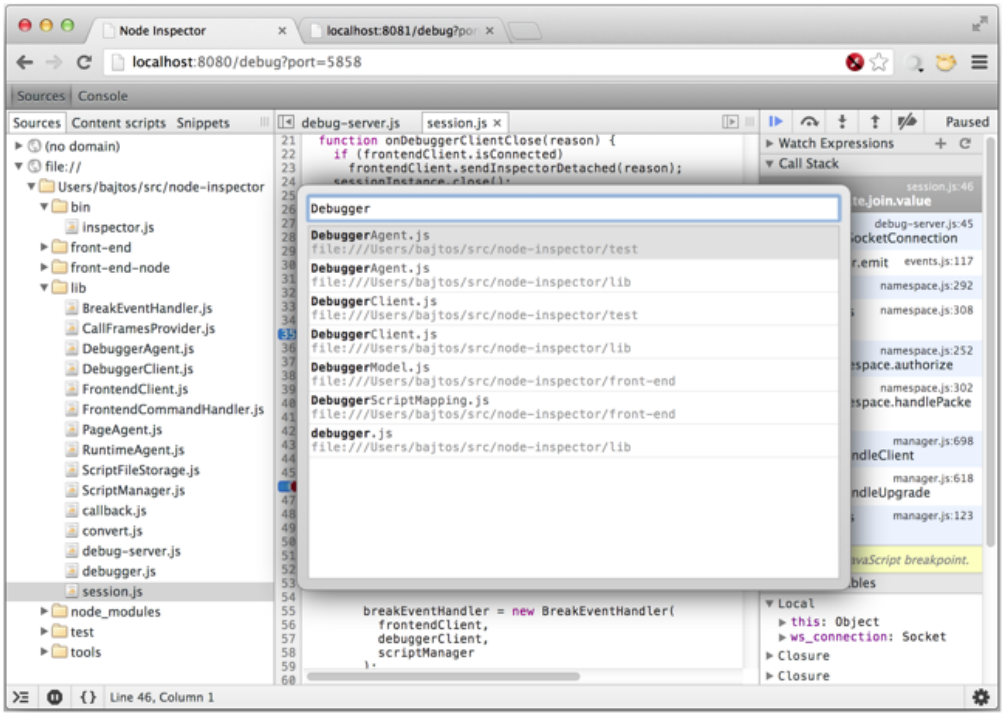
Automatically load and parse files in a debug session

Node-Inspector now loads and parses files during a debug session and automatically adds them to the GUI. This is useful for –debug-brk and stepping through require() calls. The previous version of node-inspector was not updating the list of files in the browser. This meant you had to reload the node-inspector page each time to see an updated list.

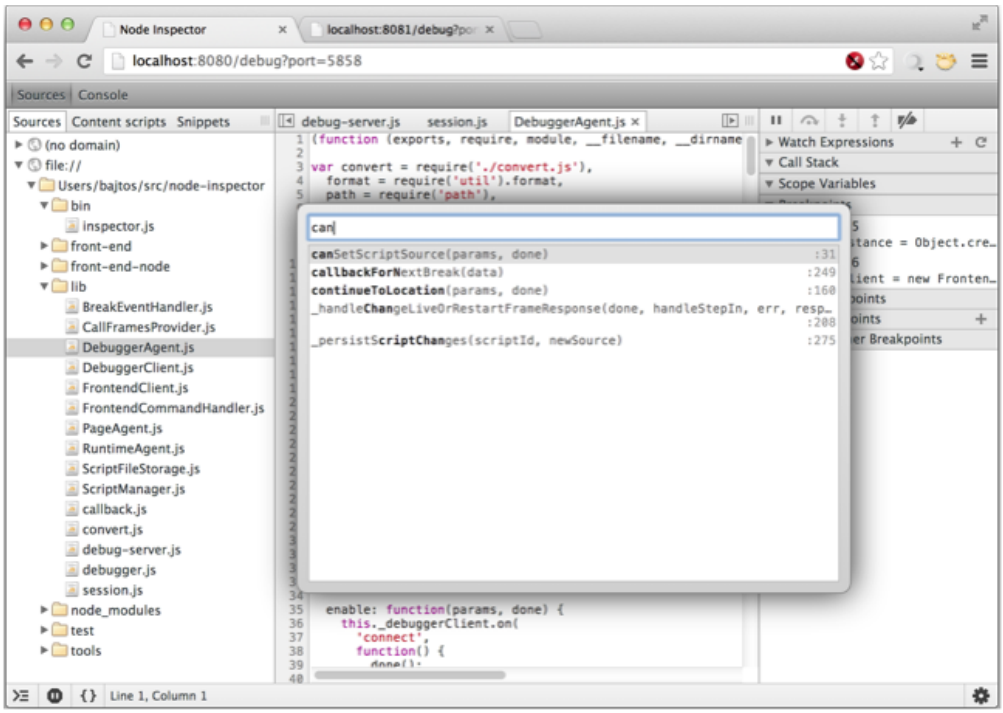
Quick navigation shortcuts

Quick navigation by jumping to files and functions easily with shortcuts. You can now:

Jump to file with search: Command+O

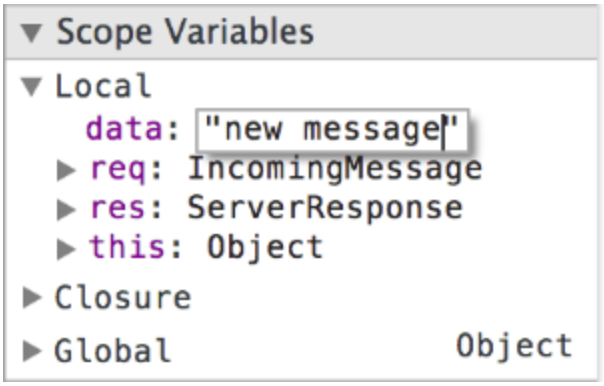


Jump to member function with search: Command+Shift+O



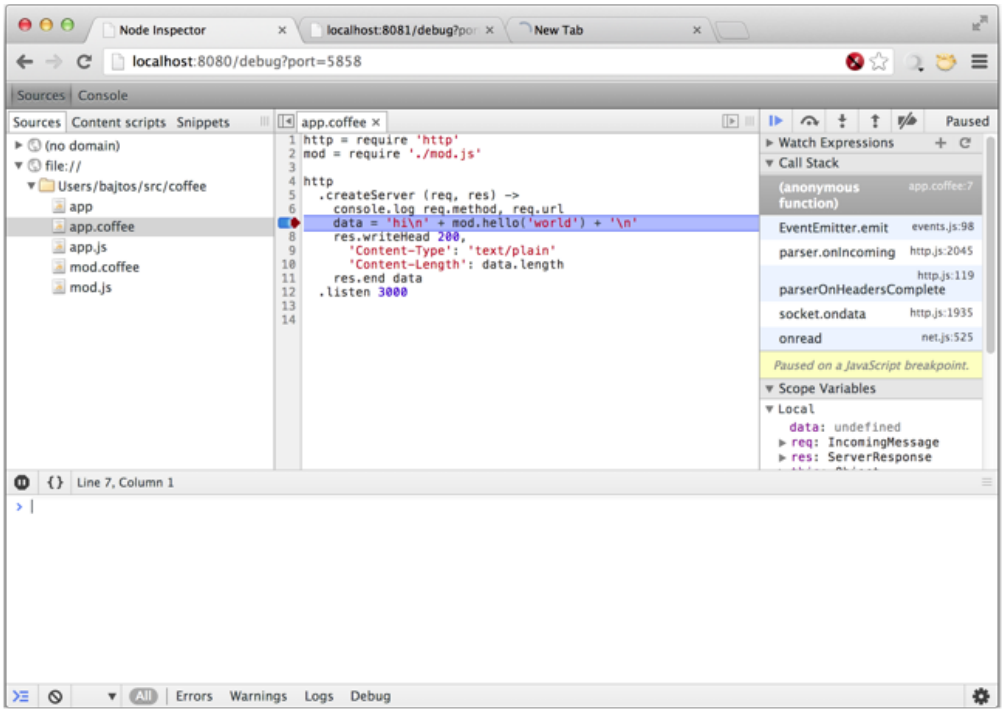
Display and edit variables

You can now hover a variable to display and edit its value.



Support for source maps

Node-Inspector now supports Source Maps, allowing you to keep client-side code readable and debuggable despite it being combined and minified, without impacting performance. For example, you can compile CoffeeScript, LiveScript or TypeScript files with source-map option turned on. Node-inspector will show you the CoffeScript, LiveScript or TypeScript source instead of trans-piled JavaScript. Also, you set breakpoints in these files.



Re-execute functions with "Restart Frame"

You can now right-click on a call frame (stack frame) in the right sidebar and select the ”Restart frame” command. This re-executes the current function from the beginning.

Sample application: blog system

- Install and run MongoDB
- Create and run sample blog application
- Walk through the application flow
- Build the application
    - Create a skeleton web application
    - Define data models using Mongoose schema
    - Configure and connect to MongoDB
    - Expose data services as REST APIs
    - Enable authentication
        - Define a passport strategy
        - Define login, logout, and account information pages
        - Glue all the pieces together

- Add test cases
- Add client-side artifacts
- Run the demo

Install and run MongoDB

The sample blog application uses MongoDB to store blog entries and users. Before you run it, you'll have to install or configure MongoDB. Follow the instructions to install MongoDB version 2.2.3 or later.

After installing MongoDB, run it by going to the directory where you installed it and entering:

```
$ mkdir sample-blog-db
$ bin/mongod --dbpath sample-blog-db
```

This creates a data directory called `sample-blog-db` and starts MongoDB using that directory for data storage. If you already have a MongoDB installation, edit the connection variables in `sample-blog/config/config.js`.

Create and run sample blog application

Use the slc command-line tool to create the sample application. Enter this command:

```
$ slc example blog
```

Now start the application as follows:

```
$ cd sn-example-blog
$ npm install
$ node app
```

You will see messages on the console such as this:

```
Sample blog server listening on port 3000
MongoDB connection opened
```

Now try the following links inside your web browser:

- View all posts -- http://localhost:3000
- View all posts as JSON -- http://localhost:3000/rest/blogs
- View all users as JSON -- http://localhost:3000/rest/users
- Create a new post (username: strongloop, pw: password) -- http://localhost:3000/post

Now that you know the application works as intended, look at the blog's structure to get an idea of how these endpoints are activated.

Walk through the application flow

The blog uses a model-view-controller design.

Here's what happens when you the hithttp://localhost:3000 URL.

1. The browser sends an HTTP GET / request to the StrongLoop Suite server.
2. The Express route for / kicks in, and it invokes the 'index' function as the controller.
3. The controller calls Mongoose blog model to retrieve all blog entries from MongoDB.
4. The EJS template for home page is rendered with the data from step 3.
5. The HTML response is sent back to the browser.

The second flow is more involved. When you hit the 'New Post' button, it will bring up the blog posting page so that you can create a new blog. Again, there are multiple steps involved within the application.

1. The browser sends an HTTP GET /post request to the StrongLoop Suite server.
2. The Express authentication handler backed by Passport intercepts the request, as the /post URL is protected.
3. Since the user hasn't logged in yet, a redirect to the Login page is sent back to the browser.

4. The browser sends an HTTP GET /login request to StrongLoop Suite server.
5. The Express route for /login kicks in, and it invokes the 'loginForm' function as the controller.
6. The controller renders the login form from its EJS template. The HTML response is sent back to the browser.
7. The user types in the user name/password and click on the 'Login'.
8. The Express route for POST /login calls the passport module which in turn invokes the Mongoose User model to make sure it has a record matching the user name/password. If yes, it redirects the browser back to the blog post page.
9. Now the user fills in the title/content and click on the 'Save' button.
10. The browser sends an HTTP POST /post request to StrongLoop Suite server.
11. The Express authentication handler intercepts the request again and it finds out the user is authenticated. It let the request continue to flow to the Express route for POST /post.
12. The route calls Mongoose blog model to save the newly created blog into MongoDB and send a redirect to / back to the browser.
13. Now the browser gets the http://localhost:3000 page as we described in the first flow. Your new post will show up at the top of the page.

Build the application

This section will go step by step to illustrate how to build the sample-blog application using StrongLoop Suite.

Create a skeleton web application

```
$ slc create web sample-blog -mr

  create : sample-blog/app.js
  create : sample-blog/package.json
  create : sample-blog/public/stylesheets/style.css
  create : sample-blog/routes/index.js
  create : sample-blog/views/index.ejs
  create : sample-blog/db/config.js
  create : sample-blog/db/mongo-store.js
  create : sample-blog/models/user.js
  create : sample-blog/sample-blog
  create : sample-blog/routes/resource.js

$ cd sample-blog
$ slc install
```

This will create a simple Express-based web application, including these files:

- `package.json` - is the node.js package descriptor. It defines the name, version and dependencies of the application.
- `app.js` - is a JavaScript file serving as the main program for sample blog application. It creates a web server and registers Express routes and views.

Define data models using Mongoose schema

For the blog application, you need a persistent store to keep user data and blog entries. The application will use MongoDB and the Mongoose No de module as the persistence layer.

To start, you need to define the **blog** and **user** schemas using Mongoose so you can create, retrieve, update, and delete the corresponding entities easily. For example, **user** will have properties such as username, password, first name, last name, and email. **Blog** will have properties such as author, title, content, and comments. See http://mongoosejs.com/docs/guide.html for more information.

There are two models needed for the blog application:

- models/blog.js
- models/user.js

```
var mongoose = require('mongoose');
var Schema = mongoose.Schema;
var CommentSchema = new Schema({
    author:{type:String, index: true},
    body:{type:String},
    date:{type:Date, default: new Date()}
});
var BlogSchema = new Schema({
    title:{type:String, index: true},
    body:{type:String},
    author:{type:String, index: true},
    date:{type:Date, default: new Date()},
    tags:{type:[String], index: true},
    comments: [CommentSchema]
});
```

Configure and connect to MongoDB

How to install a new MongoDB database is described elsewhere. To make it easy to make changes, we abstract the MongoDB related configuration into an object inside `config/config.js`. The relevant snippet is:

```
exports.creds = {
  mongo: {
    'hostname': 'localhost',
    'port': 27017,
    'username': '',
    'password': '',
    'db': 'sample-blog_development'
  }
}
```

There is also code that reads the configuration and create connection to the given MongoDB database. The file is db/mongo-store.js.

Expose data services as REST APIs

Now you have the blog model defined and you can access the blog entries remotely using REST APIs. With Mongoose and Express, it's actually pretty straightforward.

Use HTTP POST to the `/rest/blogs` URI to create a new blog entry. First, add a new function `create()` as follows:

```
/**
 * Create a new entity */
exports.create = function(mongoose) {
  var mongo = mongoose;
  return function(req, res, next) {
      var mongoModel = mongo.model(req.params.resource);
      if(!mongoModel) {
          next();
          return;
      }
      mongoModel.create(req.body, function (err, obj) {
          if (err) {
              console.log(err);
              res.send(500, err);
          }
          else {
              res.send(200, obj);
          }
      });
  };
}
```

Interestingly, the create function can be used to create instances against any defined Mongoose models as it uses the resource name to look up the schema. The body of the HTTP request is the JSON representation of the entity.

Now you need to tell Express that the create function will be used to handle HTTP POST to /rest/:resource URLs. The registration is just one line of code:

```
app.post('/rest/:resource', exports.create(mongoose));
```

The corresponding file for our sample application is /routes/resource.js. You can find all CRUD operations are supported:

HTTP Verb	URL Pattern	MongoDB Operation
POST	/rest/:resource	Create a new document
GET	/rest/:resource?skip=&limit=	List all documents for the given collection. Optionally, skip and limit parameters can be provides from the query string to support pagination
GET	/rest/:resource/:id	Retrieve a document by ID
PUT	/rest/:resource/:id	Update a document by ID
DELETE	/rest/:resource/:id	Delete a document by ID

Enable authentication

On the internet, certain web pages or APIs need to be protected so that only authorized users can access them. For example, you would generally want to secure the REST API as well as the blog post page so that only logged-in users can create blog posts.

Luckily, with Express and Passport modules, the job is not difficult. There are a few simple steps to enable authentication.

Define a passport strategy

The strategy determines the authentication mechanism to use. Here, you'll use the user collection from the MongoDB defined in `routes/auth.js`.

```
passport.use(new LocalStrategy(function(username, password, done) {
  User.findByUsernamePassword(username, password, function(err, user) {
      if (err) {
          return done(err);
      }
      if (!user) {
          return done(null, false);
      }
      return done(null, user);
  });
}));
```

Define login, logout, and account information pages

These pages are modeled as Express views and routes too. See `views/account.ejs` and `views/login.ejs`.

Glue all the pieces together

The code below sets up the login handler:

```
exports.setup = function(app) {
  app.use(passport.initialize());
  app.use(passport.session());
  app.all('/post', function(req, res, next) {
      console.log(req.path);
      if (req.path === "/login") {
          next();
      } else {
          ensure.ensureLoggedIn('/login')(req, res, next);
      }
  });
  app.get('/login', exports.loginForm);
  app.post('/login', exports.login);
  app.get('/logout', exports.logout);
  app.get('/account', exports.account);
};
```

Add test cases

Developing an application cannot go well without test cases. There are various test frameworks for Node.js. The sample application uses Mocha. Typically, you add test cases to `/test` folder and name them as `*-mocha.js`.

In `package.json`, create a script so that you can use "npm test" to run the test cases.

```
"scripts": {
  "start": "node app",
  "test": "./node_modules/mocha/bin/mocha --timeout 30000 --reporter spec
test/*-mocha.js --noAuth"
}
```

Add client-side artifacts

At this point, you pretty much have all the backend code ready for the blog application. If you are comfortable with REST APIs, you can definitely start to use 'curl' scripts to try out the blog functions. It would be nice to have simple UI to list blog entries, add comments, and create new entries.

Here are the artifacts to provide the UI:

- `views/index.ejs`: The EJS template for index page that lists all blog entries. It takes an array of blog entries.
- `views/post.ejs`: The form to post new blog entries.

There are peers of these two views in Express routes:

- `routes/index.js`: Define the functions to list blog entries and update them with new comments.
- `routes/post.js`: Define the functions to render post form and create new blog entries. The routes also register URLs, such as:

```
exports.setup = function(app) {
  app.get('/', exports.index);
  app.post('/postComment', exports.postComment);
};
exports.setup = function(app) {
  app.get('/post', exports.post);
  app.post('/post', exports.save);
};
```

The last piece of the puzzle is static assets, such as HTML files, images, or CSS sheets. We place them under /public and register a static handler with Express in app.js as follows:

```
app.use(express.static(path.join(__dirname, 'public')));
```

Run the demo

Finally, it's demo time! Enter these commands to run the demo:

```
$ cd sample-blog
$ mkdir sample-blog-db
$ mongod --dbpath=sample-blog-db
$ slc install
$ node app
```

Enjoy the blog application at: http://localhost:3000/