Reference

Frequently asked questions

General questions

StrongLoop supports the following operating systems:

- RHEL/CentOS 6.3 (RPM)
- Debian/Ubuntu 12.10 (DEB)
- Mac OS X Mountain Lion 10.8 (PKG)
- Microsoft Windows 8, 2008 (MSI).
  NOTE: Node does not support using Cygwin. Instead use Windows Command Prompt for command-line tools.

What version of Node do you support?

We support most recent versions of Node, but for best results use the latest stable version of Node, as stated on http://nodejs.org/.

What is slc?

`slc` is a command line tool for building and managing applications. Use it to:

- Rapidly create LoopBack applications and REST APIs, using `slc loopback`. See Creating an application.
- Build and deploy any Node application and manage with StrongLoop Process Manager.
- Run an application in place under control of StrongLoop PM, using `slc start`. See slc start for more information.
- Update applications with zero downtime.

For a complete reference, see Command-line reference.

How do I install the latest version of StrongLoop?

In brief, just use:

```
$ npm install -g strongloop
```

For more information, see Installing StrongLoop.

See Updating to the latest version to update to the latest version.

LoopBack

See LoopBack FAQ for answers to additional detailed LoopBack questions.

Is LoopBack free? How much does it cost?

There are free and paid versions of LoopBack. See http://strongloop.com/node-js/subscription-plans/ for more information.

LoopBack uses a dual license model: you may use it under the terms of the open source MIT license, or under the commercial StrongLoop License. See the license file for the full text of both licenses.

Is there a developer forum or mailing list?

Yes! The LoopBack Google Group is a place for devlopers to ask questions and discuss LoopBack and how they are using it. Check it out!

There is also a LoopBack Gitter channel for realtime discussions with fellow LoopBack developers.

StrongLoop also publishes a blog with topics relevant to LoopBack; see Blog posts for a list of the latest posts.

What client SDKs does LoopBack have?

LoopBack has three client SDKs for accessing the REST API services generated by the LoopBack framework:

- iOS SDK (Objective C) for iPhone and iPad apps. See iOS SDK for more information.
- Android SDK (Java) for Android apps. See Android SDK for more information.
- AngularJS (JavaScript) for HTML5 front-ends. See AngularJS JavaScript SDK for more information.

Which data connectors does LoopBack have?

LoopBack provides numerous connectors to access enterprise and other backend data systems.

Database connectors:

- Memory connector
- MongoDB connector
- MySQL connector
- Oracle connector
- PostgreSQL connector
- Redis connector
- SQL Server connector

Other connectors:

- ATG connector
- Email connector
- Push connector
- Remote connector
- REST connector
- SOAP connector
- Storage connector

Additionally, there are community connectors created by developers in the LoopBack open source community.

Why do curl requests to my LoopBack app fail?

If the URL loads fine in a browser, but when you make a `curl` request to your app you get the error:

```
curl: (7) Failed to connect to localhost port 3000: Connection refused
```

The cause is likely to be because of incompatible IP versions between your app and `curl`.

On Mac OS 10.10 (Yosemite), `curl` uses IP v6 by default.

LoopBack, by default uses IP v4, and `curl` might be using IP v6. If you see IP v6 entries in your hosts file (::1 localhost, fe80::1%lo0 localhost), it is likely that `curl` is making requests using IP v6. To make request using IP v4, specify the `--ipv4` option in your curl request as shown below.

```
$ curl http://localhost:3000 --ipv4
```

You can make your LoopBack app use IP v6 by specifying an IP v6 address as shown below:

```
app.start = function() {
  // start the web server
  return app.listen(3000, '::1',function() {
    app.emit('started');
    console.log('Web server listening at: %s', app.get('url'));
  });
};
```

Application monitoring

What endpoints can you monitor with StrongLoop?

The endpoints you can monitor include:

- HTTP requests – URLs and parameters
- Memory object caching – Memcached
- Web-services – SOAP, REST, HTTP
- Relational databases - MySQL, Oracle
- Including code to connect to MongoDB backends
- NoSQL data stores like MongoDB

Is it possible to get a heap dump when I notice an issue?

You can get detailed heap snapshots with StrongLoops tools. See Taking heap snapshots for details.

What is CPU profiling?

StrongLoop CPU profiling collects the CPU footprint of the application, modules, functions and lines of code. You can start and stop CPU profiling with Arc or `slc`. Once the profile is collected, you can view the following analytics:

- Time distribution of code calls in perspective of CPU footprint over the collection period
- Call chains with insight into multiple child elements
- Ability to hide or explore call chain child elements taking less than 5% time of parent or with single child tiers.

For more information, see Profiling.

Other questions

What is a "private registry?"

Node applications use `npm` to automate the process of installing, upgrading, configuring, and removing Node packages. It automatically handles dependencies, so when you install a package, it will automatically install other packages that it requires. The standard public registry is npmjs.org, which maintains a database of node packages (over 47,000 and growing).

A private registry is a “non-public” source and configuration management system for node projects that enable an enterprise to use only the modules required by their project while being able to share the common registry across multiple teams within their organization. These registries allow node projects to whitelist and manage proliferation of module functionalities, versions, and source.

Once you have a private registry set up, StrongLoop Controller enables you to easily switch between it and the public registry, and to promote packages from private to public registry. For more information, see Using multiple package registries .

Glossary

ACL

Access control list. Used to restrict users' and applications' access to data in models.

adapters

Adapters provide the transport-specific mechanisms to make remote objects (and collections thereof) available over their transport.

cluster

A set of identical Node worker processes all receiving requests on the same port. See also: worker.

connector

See LoopBack Connector.

CRUD

Create, read, update and delete (CRUD), the four basic functions of persistent storage typically provided by LoopBack data sources.

data source

A factory for model classes. A data source connects with specific database or other backend system using a connector. All model classes within single data source shares same connector type and one database connection. But it's possible to use more than one data source to connect to different databases.

enterprise connector

Module the connects to backend data source such as Oracle, MySQL, or MongoDB.

hook

TBD. See Strong Remoting.

LDL

LoopBack Definition Language, a simple way to define LoopBack data models in JavaScript or JSON. LDL compiles a model definition into a JavaScript constructor.

libuv

Multi-platform support library with focus on asynchronous I/O, primarily developed for use by Node.js.

LoopBack

Mobile backend framework that can run in the cloud or on your own servers. Built on StrongLoop curated modules and open-source Node.js modules.

LoopBack connector

A LoopBack connector provides access to a backend system such as a database, REST API, or other service. Application code does not use connectors directly. Rather, you use the LoopBack Datasource Juggler to create a dataSource to interact with the connector.

LoopBack DataSource Juggler

An object-relational mapping that provides a common set of interfaces for interacting with databases, REST APIs, and other data sources.

MBaaS

Mobile backend as a service. See Backend as a service on Wikipedia.

model

A LoopBack model consists of: application data, validation rules, data access capabilities, and business logic. A model provides a remote API that clients use to display user interface elements and to interact with backend systems.   Every LoopBack application by default has some built-in models: user, application, email, and several models for access control.

Node.js

Software platform used to build scalable network applications. Node.js uses JavaScript as its scripting language, and achieves high throughput via non-blocking I/O and a single-threaded event loop. See Node.js on Wikipedia. Usage note: Initially, "Node.js," thereafter "Node."

npm

Node package manager, the command-line tool for installing applications and managing dependencies using the npm registry.

On-premises

Located in the customer's own datacenter. Note ending "s."

open-source, open source

When used as an adjective, hyphenate; for example: "This is open-source software." See Open-source software on Wikipedia. Note: Although it is common not to hyphenate this term, we are using the standard English rules for hyphenating a compound adjective.

push notification

A short text message or other notification sent from a server application to client mobile apps. See Push notifications.

remote objects

JavaScript objects exported by a StrongLoop app over the network in the same way you export functions from a module. You can invoke methods on remote objects locally in JavaScript

runtime

At the time of application execution, in contrast to when the application is compiled or deployed. Do not hyphenate.

SDK

Software development kit. In the context of LoopBack, a platform-specific library for creating client applications. LoopBack provides client SDKs for iOS, Android, and web (AngularJS) clients.

slc

StrongLoop Controller, the command-line tool for Node development and operations. It provides command line access to all StrongLoop facilities. For more information, see Operating Node applications.

streams

See Strong Remoting

Worker

A node child process.

Command-line reference

Use the `slc` command-line utility to create, manage, and work with LoopBack applications and other Node applications.

Command syntax

```
slc [command] [sub-command] [options] [args]
```

⚠ Some commands and sub-commands have *required* "options," as noted in the command reference for the specific command. Such cases are not accurately covered by the above syntax.

Syntax conventions

The documentation uses the following conventions for command-line syntax. In general:

- An item in *`monospace italics`* represents a variable for which you can substitute a string or number, as specified for the specific command.
- An item enclosed in square brackets `[ .. ]` represents an optional item.

Then:

`command` indicates one of the `slc` commands (see table below).

`[`*`sub-command`*`]` indicates a sub-command, for commands that take a sub-command, for example `runctl` or `ctl`.

`[`*`options`*`]` indicates one or more command options, possibly with corresponding values, as described for the particular command. Options, as their name implies, are typically optional. However, some commands or sub-commands have *required* "options," as noted in the command reference for the specific command.

Syntax for options is:

`-x` ***`value`*****:** Single-letter option; separate value (if provided) with a space.
`--extra=value` or **`--extra value`:** Full spelled-out option; separate value (if provided) with a space or an equals sign (=).

Where value is either:

- **`<`*`foo`*`>`**, a value that you must provide, either a string or number, as described for the particular command.
- **`[`*`foo`*`]`**, a value that you may optionally provide. It is not required.

`[`*`args`*`]` is an optional argument.

Most `slc` commands have the two standard options below. Most commands also have additional options.

STANDARD OPTIONS

`-h, --help`
Display help information.

`-v, --version`
Display version number.

Commands

The following table summarizes `slc` commands. Each command has specific options and arguments, and some provide sub-commands, as described in each command reference article.

Command	Description
slc arc	Start StrongLoop Arc
build	Build a Node application package, preparing it to be deployed.
debug	Debug module with Node Inspector.
deploy	Deploy a Node application to a process manager.
env	Display information about the Node runtime environment.
loopback	Scaffold a LoopBack application. See Command-line reference (slc loopback) for details.
pm	Manage deployed Node applications with StrongLoop Process Manager.
ctl	Control applications using StrongLoop Process Manager.

pm-install	Install StrongLoop Process Manager as an OS service.
registry	Switch between different npm registries.
start	Run an application locally, under control of StrongLoop Process Manager.

Getting the latest version of slc

See Updating to the latest version for instructions.

slc arc

Run StrongLoop Arc, by default opening it in a web browser window.

SYNOPSIS

```
slc arc [options]
```

OPTIONS

`--cli`
Start the backend only; do not open the browser.

`--licenses`
Start Arc and display the Licenses page. See Managing your licenses for more information.

STANDARD OPTIONS

`-h`, `--help`
Display help information.

`-v`, `--version`
Display version number.

StrongLoop Arc will use a different port number each time you run it. To run it on a specific port, use the PORT environment variable, for example:

```
$ PORT=4000 slc arc
```

slc build

Build a Node application package, preparing it for production. See Installing dependencies for more information on using this command.

See also: Building applications with slc.

SYNOPSIS

```
slc build [options]
```

With no options, the default depends on whether the current app is in a Git repository (if there is a `.git` sub-directory in the current app root directory):

- If in a Git repository, the default option is `--git`, which installs and commits the build results to the "deploy" branch. If a "deploy" branch does not exist, the tool will create it.
- If not, the default options are `--npm` which will bundle, install, and pack the build results into a `<`*`package-name`*`>-<`*`version`*`>.tgz` file.

OPTIONS

`-b, --bundle`
Include dependencies in tar (.tgz) file.

`-c, --commit [--onto <branch> ]`
Commit build output to specified branch, and create the branch if necessary. Default branch is "deploy." This option requires that Git is installed.

`-i, --install [--scripts]`
Install dependencies, without scripts, by default.
Add the `--scripts` option to run scripts (to build add-ons).

`-g, –git`
Shortcut for `--install --commit`.

`-n, --npm`
Shortcut for `--install --bundle --pack`.

`-p, --pack`
Pack into a publishable archive. Output is a .tgz file in the format produced by `npm pack` and accepted by `npm install`.

STANDARD OPTIONS

`-h, --help`
Display help information.

`-v, --version`
Display version number.

Using Git

See Committing a build to Git for more information.

slc ctl

Control StrongLoop Process Manager. This command enables you to:

- Start, stop, and restart applications that are under management of StrongLoop PM.
- Start and stop application clusters; change number of workers in a cluster.
- Start and stop CPU profiling, take heap snapshots, and perform object tracking.
- Modify environment settings while an application is running.

See also:

- Using Process Manager
- Clustering
- Logging
- Profiling with slc
- Monitoring with slc

This command works only with applications running under control of StrongLoop Process Manager. For local use, that means you must run the application with `slc start`, not `node .` or `slc run` (that don't run applications in Process Manager).

SYNOPSIS

```
slc ctl [options] [sub-command]
```

OPTIONS

`-C, --control <ctl>`
Control endpoint for Process Manager. For a remote Process Manager, this must specify the URL on which the Process Manager is listening.

If Process Manager is using HTTP authentication then you must set valid credentials in the URL, in the form `http://username:password@example.com:7654`.

To tunnel over SSH using an HTTP URL, use the protocol `http+ssh`., for example `http+ssh://example.com:7654`.

- The SSH username defaults to your current user. Override the default with the SSH_USER environment variable.
- Authentication defaults to your current `ssh-agent`. Override the default with the SSH_KEY environment variable specifying the path of an existing private key to use.

Use the STRONGLOOP_PM environment variable to set a default value for the `--control` option; this eliminates the need to supply the option every time.

If you don't specify a channel with this option, the tool uses the following in this order of precedence:

1. STRONGLOOP_PM environment variable, that can specify a local domain path, or an HTTP URL. Use an HTTP URL to specify a remote Process Manager. Use `localhost` for a local Process Manager. The URL must specify at least the process manager's listen port, such as `http://example.com:7654` (default is 8701).
2. `./pmctl`: Process Manager running the current working directory, if any.
3. `~/.strong-pm/pmctl`: Process Manager running in the user's home directory.
4. `/var/lib/strong-pm/pmctl`: Process Manager installed by `slc pm-install`.
5. `http://localhost:8701`: Process Manager running on localhost.

STANDARD OPTIONS

`-h, --help`
Display help information.

`-v, --version`
Display version number.

SUB-COMMANDS

The default sub-command is `status`.

This command has three types of sub-commands:

- Global commands that apply to Process Manager itself.
- Commands that apply to a specific service.
- Commands that apply to a specific worker process.

When you deploy an application to Process Manager, you give the deployed application instance a name, referred to as the *service name* and indicated in command arguments as `<service>`. By default, it is the `name` property from the application's `package.json`.

Process Manager also automatically generates an integer ID for each application it's managing. Typically, the IDs start with one (1) and are incremented with each application deployed; however, the value of ID is not guaranteed. Always determine it with `slc ctl status` once you've deployed an app.

A service becomes available over the network at `http://hostname:port` where:

- `hostname` is the name of the host running Process Manager
- `port` is 3000 + service ID.

For example, if Process Manager is running on my.host.com, then service ID 1 is available at `http://my.host.com:3001`, service ID 2 at `http://my.host.com:3002`, and so on.

Command	Description	Arguments
Global sub-commands		
info	Display information about Process Manager.	
ls	List services under management.	
shutdown	Stop the process manager and all applications under management.	
Service sub-commands (apply to a specific service) The argument <*service*> is the name or ID of a service.		
create <*service*>	Create application instance *service*.	<*service*>, name or ID of the service to create.
cluster-restart <*service*>	Restart the current application cluster workers.	<*service*>, name of target service.
env[-get] <*service*> [*env*...]	List specified environment variables for <*service*>. If none are given, list all variables.	<*service*>, name or ID of target service. <*env*>, one or more environment variables.

env-set <*service*> <*env*>=<*val*>...	Set one or more environment variables for <*service*> and hard restart it with new environment.	<*service*>, name or ID of target service. One or more environment variables, <*env*>, and corresponding value <*val*>.
env-set <*service*> PORT=<*n*>	Run service on the specified port instead of the automatically-generated port. Normally, PM sets the port to a value guaranteed to be different for each app: Use this sub-command to override this behavior. **Do not specify a port already in use**. Doing so will cause the app to crash.	<*service*>, name or ID of target service. <n>, integer port number to use.
env-unset <*service*> <*env*>...	Unset one or more environment variables for <*service*> and hard restart it with the new environment.	<*service*>, name or ID of target service. <*env*>, one or more environment variables.
log-dump <*service*> [--follow]	Empty the log buffer, dumping the contents to stdout. Use `--follow` to continuously dump the log buffer to stdout.	<*service*>, name or ID of target service.
npmls <*service*> [*depth*]	List dependencies of service with id <*service*>	<*service*>, name or ID of target service. *depth*, an integer limit of levels for which to list dependencies; default is no limit.
remove <*service*>	Remove <*service*>.	<*service*>, name or ID of target service.
restart <*service*>	Hard stop the current application: kill the supervisor and its workers with SIGTERM; then restart the current application with new configuration.	<*service*>, name or ID of target service.
set-size <*service*> <*n*>	Set cluster size for <*service*> to <*n*> workers.	<*service*>, name or ID of target service. <*n*>, positive integer.
start <*service*>	Start <*service*>.	<*service*>, name or ID of target service.
status [*service*]	Report status. This is the default command.	*service*, optional name of target service. Default is to show status for all services.
stop <*service*>	Hard stop <*service*>: Kill the supervisor and its workers with SIGTERM.	<*service*>, name or ID of target service.
soft-stop <*service*>	Notify workers they are being disconnected, give them a grace period to close existing connections, then stop the current application.	<*service*>, name or ID of target service.
soft-restart <*service*>	Notify workers they are being disconnected, give them a grace period to close existing connections, then restart the current application with new configuration.	<*service*>, name or ID of target service.
tracing-start <*service*>	Restart all workers with tracing on.	<*service*>, name or ID of target service.
tracing-stop <*service*>	Restart all workers with tracing off.	<*service*>, name or ID of target service.

Worker process sub-commands (apply to a specific worker process)

The argument <*worker*> is a worker specification; either:

- <*service_id*>.1.<*worker_id*>, where <*service_id*> is the service ID and <*worker_id*> is the worker ID.
- <*service_id*>.1.<*process_id*>, where <*service_id*> is the service ID and <*process_id*> is the worker process ID (PID).

cpu-start <*worker*> [*timeout* [*stalls*]]	Start CPU profiling on worker or process ID <*id*>. Use cpu-stop to save the profile data. NOTE: Requires Node version 0.11 or higher. Saves profiling data to a file you can view with Chrome Dev Tools. See CPU profiling for more information.	<*worker*>, a worker specification (see above). **Linux only**: [*timeout*], timeout period (ms) for Smart profiling. Start CPU profiling when the specified process's Node event loop stalls for more than the specified timeout period. [*stalls*], number of event loop stalls after which the profiler will be stopped automatically (default is 0, never auto-stop). For more information, see Smart profiling with slc.
cpu-stop <*worker*> [*filename*]	Stop CPU profiling on worker process <*id*> and save results in file `name.cpuprofile`.	<*worker*>, a worker specification (see above). *filename*, optional base file name; default is `node.<id>.cpuprofile`.
heap-snapshot <*worker*> [*filename*]	Save heap snapshot data for worker process <*id*> and save results in file `name.heapsnapshot`. Saves profiling data to a file you can view with Chrome Dev Tools. See Heap memory profiling for more information.	<*worker*>, a worker specification (see above). *filename*, optional base file name; default is `node.<id>.heapsnapshot`.
objects-start <*worker*>	Start tracking objects on worker process <*id*>.	<*worker*>, a worker specification (see above).
objects-stop <*worker*>	Stop tracking objects on worker process <*id*>.	<*worker*>, a worker specification (see above).
patch <*worker*> <*file*>	Apply patch <*file*> to <*id*> to get custom metrics.	<*worker*>, a worker specification (see above). <*file*>, file name.

slc debug

Run the Node Inspector debugger.

See also: Debugging applications

SYNOPSIS

```
slc debug [node-inspector-options] [options] script [script-arguments]
```

The *script* argument is resolved relative to the current working directory. If no such file exists, then the command searches env.PATH.

By default, the command loads the module in the current working directory in the REPL session as `m`. This enables you to call and debug arbitrary functions exported by the current module.

The default mode is to break on the first line of the script. To run immediately on start, use `--no-debug-brk` or click the **Resume** button.

OPTIONS

--cli, -c
CLI mode, do not open browser.

`--debug-brk, -b`
Break on the first line (as with `node --debug-brk`). Default is true.

`--debug-port, -d`
Node/V8 debugger port. Default is 5858.

--web-port, -p, --port
Node Inspector port. Default is 8080.

NODE INSPECTOR OPTIONS

The `slc debug` command passes the following options to the `node-inspector` command.

`--hidden`
Array of files to hide from the UI (breakpoints in these files will be ignored).   Default is `[]`.

`--no-preload`
Do not preload JavaScript files. Use this option for faster startup.

`--save-live-edit`
Save live edit changes to disk (update the edited files). Default is false.

`--stack-trace-limit <`*`n`*`>`
Show <*n*> stack frames on a breakpoint. Default is 50.

`--web-host <`*`port`*`>`

HTTP host on which to listen. Default is 127.0.0.1.

STANDARD OPTIONS

`-h`, `--help`
Display help information.

`-v`, `--version`
Display version number.

slc deploy

Deploy a Node application to StrongLoop Process Manager

See also: Deploying applications with slc

SYNOPSIS

slc deploy [*options*] [*URL* [*package* | *branch*]]

OPTIONS

`-s, --service` `[`*`service`*`]`
Deploy to service <*service*>, where <*service*> is the service name or ID.   Defaults to the name of the application specified in the `package.json` file. If Process Manager does not have a service with that name, it creates one and deploys the application to it.

`-z, --size` <*n*>
Set size of cluster to <*n*> worker processes before deployment, where <*n*> is a positive integer or the string "cpus," meaning to run one worker per CPU.

When you deploy an application to Process Manager, you give the deployed application instance a name, referred to as the *service name* and indicated in command arguments as <*`service`*>. By default, it is the `name` property from the application's `package.json`.

Process Manager also automatically generates an integer ID for each application it's managing. Typically, the IDs start with one (1) and are incremented with each application deployed; however, the value of ID is not guaranteed. Always determine it with `slc ctl status` once you've deployed an app.

A service becomes available over the network at `http://hostname:port` where:

- `hostname` is the name of the host running Process Manager
- `port` is 3000 + service ID.

For example, if Process Manager is running on my.host.com, then service ID 1 is available at `http://my.host.com:3001`, service ID 2 at `http://my.host.com:3002`, and so on.

STANDARD OPTIONS

`-h`, `--help`
Display help information.

`-v`, `--version`
Display version number.

ARGUMENTS

URL
The URL of the StrongLoop process manager; for example, `http://prod.foo.com:7777/`. Defaults to `http://localhost:8701/`. Both host and port are optional. Host defaults to `localhost` and port defaults to `8701`. If the server requires authentication, the credentials must be part of the URL; see Securing Process Manager.

branch
Deploy the specified branch. The default is a branch named `deploy`.

package
Name of a tar (.tgz) file or npm package to deploy. The default is `../<package_name>-<package_version>.tgz`, where the package name and version come from the `package.json` in the current working directory. **NOTE**: A tar file (.tgz file) must be of the format created by `npm pack`, which is the format created by `slc build` and the Arc Build & Deploy module.

EXAMPLES

Deploy the default npm package or git branch to localhost:

```
$ slc deploy
```

Deploy the default npm package or git branch to a remote host:

```
$ slc deploy http://prod1.example.com
```

Deploy to a remote host, on a non-standard port, using authentication:

```
$ slc deploy http://user:pass@prod1.example.com:8765
```

Deploy "production" branch to localhost:

```
$ slc deploy http://localhost production
```

slc env

slc env – Print runtime environment

Prints information about node run-time from the process module (paths, platform, config, execPath, and features). The output can be limited by using one or more selectors, see examples.

SYNOPSIS

```
slc env [options] [selectors...]
```

OPTIONS

`-h`, `--help`:
print usage information

EXAMPLES

Print zlib version:

```
% slc env versions zlib
    { versions: { zlib: '1.2.3' } }
```

Print paths:

```
% slc env paths
    { paths:
       { link:
          { node: '/usr/local/bin/node',
            npm: '/usr/local/bin/npm',
            slc: '/usr/local/bin/slc' },
         exec:
          { node: '/usr/local/stow/installed-node/bin/node',
            npm: '/usr/local/lib/node_modules/slc/node_modules/npm/bin/npm-cli.js',
            slc: '/usr/local/lib/node_modules/slc/bin/slc',
            slcNpm: '/usr/local/lib/node_modules/slc/node_modules/.bin/npm',
            nodeNpm: '/usr/local/stow/installed-node/bin/npm' } } }
```

slc lb

This command is deprecated for LoopBack 2.0. Use slc loopback instead.

The `slc` command assumes that the application root directory is always the current directory, even if you change `appRootDir` in code.

Creates LoopBack examples, apps, and workspaces.

slc lb – Create LoopBack examples, apps, and workspaces

SYNOPSIS

```
slc lb <command> <arg> [flags]
```

COMMANDS

Supported `lb` commands are:

`workspace [name]`:
Initialize a workspace as a new empty directory with an optional name. The default name is “loopback-workspace”.

```
$ slc lb workspace my-loopback-workspace
```

`project <name> [--no-install]`:
Create a LoopBack application in a new directory within a workspace using the given name. The argument is required. With **`--no-install`**, will not install the npm dependencies.

```
$ cd my-loopback-workspace
$ slc lb project my-app
$ slc run my-app # to run the app
```

`model <name>`:
Create a model in an existing LoopBack application using the given name.
If you provide the **`-i`** or **`--interactive`** flags, you will be prompted

through a model configuration. The argument is required.

```
$ cd my-app
$ slc lb model product -i
```

`datasource <name>`:
Create a datasource in an existing LoopBack application using the given name.
You must supply a connector name using the **`--connector`** option.

```
$ cd my-app
$ slc lb datasource mongo --connector mongodb
```

`acl`:
Add a new permission to an existing model or to all models. Creates an ACL entry
Use options described below to configure the exact permission.

ACL Options

`--model <name>`:
Specify the model name to apply the new permissions against.
You must supply either the **`--model`** or **`--all-models`** argument.

`--all-models`:
Apply the permission to all models. You must supply either the **`--model`** or
`--all-models` argument.

ACL Access Types

`--all`:
Set the access type to the wildcard. This matches **`read`**, **`write`** and
`execute`.

`--read`:
Set the access type to **`READ`**.

`--write`:
Set the access type to **`WRITE`**.

`--execute`:
Set the access type to **`EXECUTE`**.

ACL Properties and Methods

`--property`:
Specify a specific property to apply the permission to. Defaults to all.

`--method`:
Specifiy a specific method to apply the permission to. Defaults to all.

ACL Role Identifiers

`--owner`:
Only apply this permission to users who own the specified model instance.

`--related`:
Any user with a relationship to the object

`--authenticated`:
Authenticated users

`--unauthenticated`:
Unauthenticated users

`--everyone`:
All users

ACL Permissions

`--alarm`:
Generate an alarm, in a system dependent way, the access

`--allow`:
Explicitly grants access to the model.

`--deny`:
Explicitly denies access to the model.

`--audit`:
Log, in a system dependent way, the access specified in the permissions component of the ACL entry.

ACL Notes

you may only supply a single access type

you may only supply a single role identifier

you may only supply a single permission

ACL Examples

```
# disable all access to all models
# note: other permissions will override this
$ slc lb acl --deny --all-models --everyone --all

# allow authors to edit posts
$ slc lb acl --model post --allow --owner --all

# only allow owners to access objects
$ slc acl --deny --all-models --everyone --all
$ slc acl --allow --all-models --owner --all
```

OPTIONS

`-h`, **`--help`**:
Display this help text.

`-i`, **`--interactive`**:
Run an interactive model creation. Only supported with the model command.

`--data-source <name>`:
Supply a custom data source when creating a model. Defaults to “db”.

`--private`:
Do not expose the model’s API remotely.

`-c`, **`--connector`**:
Specify the connector name when creating a datasource (required).

slc loopback

Create and scaffold a new LoopBack application or add to an existing application.

See also: Using LoopBack tools

SYNOPSIS

slc loopback[:*generator*] [*options*]

With no *generator*, prompts for:

- Directory in which to create the application. Press **Enter** to create the application in the current directory.
- Name of the application, defaults to the directory name previously entered.

The tool then creates the standard LoopBack application structure (see Project layout reference for details).

OPTIONS

`--skip-install`
Do not install npm dependencies. Default is false.

`--generators`
List Loopback generators available.

STANDARD OPTIONS

`-h, --help`
Display help information.

`-v, --version`
Display version number.

GENERATORS

```
slc loopback:generator
```

For an existing application, use in the application root directory and provide *generator*, a LoopBack generator:

- ACL generator
- Application generator
- Boot script generator
- Data source generator
- Example generator
- Middleware generator
- Model generator
- Property generator
- Relation generator
- Swagger generator

slc pm

Prerequisite: This command requires Git version 1.8 or higher.

Run StrongLoop Process Manager (PM) to manage your Node applications.

See also: Using Process Manager

After packaging an application with `slc build`, use `slc deploy` to deploy your application to the Process Manager. See Building and deploying and Using Process Manager for more information.

To control Process Manager via HTTP, start it with the `--control` option and specify the desired port. Then you can deploy apps to the PM using `slc deploy` , specifying the host name and port; and you can control PM using slc ctl. To secure access to PM, enable HTTP basic authentication; see Securing Process Manager for more information.

SYNOPSIS

```
slc pm [options]
```

OPTIONS

`-b, --base <basedir>`
Save applications to *`basedir`*, including both Git repositories and npm packages, as well as the working directories, and any other files required. Default is `.strong-pm` in the current user's home directory (`~/.strong-pm`).

`-C, --control <ctl>`
Listen for control messages on `<ctl>.` Default is ctl.

`-d, --driver <driver>`
Specify application execution driver, either `direct` (the default) or `docker`.

`-l, --listen <port>`
Listen on <*port*> for application deployment and control commands.

`--no-control`
Do not listen for control messages.

`-P, --base-port <port>`
Run applications on port number <*port*> + service ID. Default base port is 3000.

STANDARD OPTIONS

`-h, --help`
Display help information.

`-v, --version`
Display version number.

EXAMPLE

Run StrongLoop Process Manager, listening on port 777:

```
$ slc pm -l 7777
```

`Clone and push an application for a non-production deploy:`

```
$ git clone git@github.com:strongloop/loopback-example-app.git
$ cd loopback-example-app
$ slc deploy http://localhost:7777/repo
```

This brief example installed all the dependencies on the server. In practice, build in app dependencies. Do not install them dynamically at run-time; for example:

```
$ slc build --install --commit
$ slc deploy http://localhost:7777/repo
```

slc pmctl

This command has been renamed to slc ctl. Please update your installation. See Updating to the latest version.

slc pm-install

Install StrongLoop Process Manager as an operating system service.

See also: Setting up a production host

On a system where you have installed StrongLoop with `npm install -g strongloop`, use the `slc pm-install` command. One a production host where you've installed StrongLoop Process Manager with `npm install g strong-pm`, use `sl-pm-install`. The two commands are equivalent.

Use the the `slc pm-install` or `sl-pm-install` command to install StrongLoop Process Manager as a service on Linux distributions that support Upstart or systemd.

By default, the command uses Upstart 1.4 or newer. The default will work, for example on Ubuntu 12.04 or newer. Otherwise, on systems:

- With Upstart 0.6 - 1.3.x (for example Ubuntu 10.04, CentOS, or AWS Linux), use the `--upstart 0.6` option.
- That support Systemd instead (for example Fedora or OpenSUSE), use the `--systemd` option.

SYNOPSIS

```
slc pm-install [options]
```

Install StrongLoop Process Manager as a service using the specified options. By default, the command installs the service as an Upstart job using a template for Upstart 1.4 or higher. Use the `--systemd` option to install as a `systemd` service instead. Use the `--upstart <version>` option to specify a different version of Upstart.

See Installing Process Manager as a service for more information.

OPTIONS

`-b, --base <base>`
Base directory in which to work. Default is home directory of user running Process Manager; see `--user` option.

`-d, --driver <driver>`
Specify application execution driver, either `direct` (the default) or `docker`.

`-e, --set-env env=val...`
Initial application environment variables, where *env* is name of the enviroment variable, `val` is value. To set multiple variables, enclose them all in quotes as a single argument, and separate each pair with a space, for example "K1=V1 K2=V2 K3=V3".

`-f, --force`
Overwrite existing job file if present.

`--http-auth <creds>`
Enable HTTP basic authentication, requiring the specified credentials for every request sent to the REST API.
<*creds*> has the form <*username*>:<*password*>.

`-j, --job-file <path>`
Path of Upstart job to create (default `/etc/init/strong-pm.conf`)

`-m, --metrics <stats>`
Specify `--metrics` option for supervisor running deployed applications. See Integrating with third-party consoles for more information.

`-n, --dry-run`
Don't write any files.

`-p, --port <port>`
Listen on <*port*>. Default is 8701.

`-P, --base-port <port>`
Run applications on port number <port> + service ID. Default base port is 3000.

`[–systemd | --upstart <version> ]`
Install Process Manager as either:

- A systemd service.
- An Upstart job (Upstart 1.4 is the default). Use this option to specify Upstart 0.6.

`-u, --user`
Run StrongLoop Process Manager as this user. Default is strong-pm.

STANDARD OPTIONS

`-h, --help`
Display help information.

`-v, --version`
Display version number.

slc registry

Switch between different npm registries.

See also: Using multiple package registries

SYNOPSIS

```
slc registry [options] [sub-command] [arguments]
```

When you run this command the first time, it creates a directory in your HOME directory to keep all data files and adds a "default" entry configured to use your current registry as set in `~/.npmrc`.

STANDARD OPTIONS

`-h`, `--help`
Display help information.

`-v`, `--version`
Display version number.

SUB-COMMANDS

The `slc registry` command supports the following sub-commands:

- add
- list
- remove
- use
- promote

`add`

```
slc registry add name [url]
```

Create a new registry configuration with the given name. This command runs an interactive wizard that prompts for settings like username and email.

Arguments:

- *name* - Name of the registry to add. Required.
- *url* - URL of the registry server. Optional; if you don't provide as an argument you can specify in response to interactive prompts.

`list`

```
slc registry list
```

List configured registries. This command has no arguments.

```
slc registry remove name
```

Remove the specified registry configuration and related files (for example, cache files).

Argument:

- *name* - Name of the registry to remove. Required. You must have previously added the registry with the `add` command.

`use`

```
slc registry use name
```

Use one of the configured registries previously added with the `add` command.

1. If the registry URL configured in `~/.npmrc` matches one of the registries managed by this tool, the configuration of the matching registry is updated using the values in `~/.npmrc`. This ensures that internal npm settings like authentication tokens are preserved across registry switches.
2. `~/.npmrc` is modified using the selected registry configuration to let npm use the selected registry.

Arguments:

- *name* - Name of the registry to use. Required.

`promote`

```
slc registry promote --from registry1 --to registry2 package@version
```

Promote a given package version to another registry.

Options:

- `--from` *`registry1`* - Name of the registry from which to download the package. You must supply a value for this option if you do not supply a value for the `--to` option. Default is currently active registry.
- `--to` *`registry2`* - Name of the registry to which to publish the package. You must supply a value for this option if you do not supply a value for the `--from` option.   Default is currently active registry.

You can omit one of these options; then the command will use the currently active registry.

Argument:

- *package@version* - The name and the version of the package to promote. Example: loopback@1.8.0

The `--version` option displays the version of the registry tool, strong-registry, NOT the version of slc.

ADVANCED USE

The interactive wizard provided by the "add" command covers only a subset of npm options. Change other options using the `npm config` command.

1. Switch to the registry you want to configure.

   ```
   $ slc registry use my-registry
   ```

2. Update the configuration as needed; for example, set "local-address" option.

   ```
   $ npm config set local-address 127.0.0.1
   ```

3. Next time you switch to a different registry, the changes will be applied to the stored configuration.

   ```
   $ slc registry use default
   ```

The registry configuration tracks only the npm options that are related to the registry being used. Other options like "browser" or "git" remain always the same, no matter which registry is used.

List of tracked options:

- registry
- username
- email
- proxy
- https-proxy
- local-address
- strict-ssl
- always-auth
- _auth
- _token

For the full list of npm options and their descriptions, see npm-config docs.

slc run

Run an application with StrongLoop profiling and metrics, and optionally with clustering. This command runs an application just as the `node` command does.

This command is deprecated. Instead, to run an application:

- During development, use the `node` command, for example, `node .` to run the application in the current directory.
- Locally under control of StrongLoop Process Manager, use the `slc start` command instead.

SYNOPSIS

```
slc run [options] [app [app-options...]]
```

The optional argument *app* is the name of a JavaScript file to run or a directory path. If you provide a directory path, the command will run the first of the following files found in that directory (by default the current directory):

- `server.js`
- `app.js`
- File specified by the `main` property in `package.json`
- `index.js`

If you don't provide the *app* argument, you must run the command from the main directory of a Node application.

OPTIONS

`--cluster <n>`
Set the cluster size. Default is off unless NODE_ENV is "production", in which case it defaults to CPUs.
Argument *n* is one of:

- A integer number of workers to run.
- A string containing "cpu" to run one worker per CPU in the current system.
- The string "off" to run unclustered. Then you will not be able to control clustering, but you can still monitor it.

`-C --control <ctl>`
When running a cluster, listen for local control messages on *ctl*. Without this option, the application listens on the default, `ctl`.

`-d, --detach`
Detach master from terminal to run as a daemon (default is to not detach). By default, log files are saved to `supervisor.log`. `To` eliminate all log files, use `--log=/dev/null`.

`-l, --log <file>`
When clustered, write supervisor and worker terminal output to *file*. If *file* does not have absolute path, then it is relative to the app's working directory. To create a log file for each process, use `%p` for process ID and `%w` for worker ID. See Logging for details.

NOTE: To use the `--log` option, you *must* also use the `--cluster` option. If you don't want to run the application in a cluster, then use `--cluster 1`.

`--metrics statsd:[//host[:port]][/scope]`
Send metrics to the monitoring console specified by the URL:

- `host` is StatsD server host name. Default is `localhost`.
- `port` is StatsD server port Default port is 8125.
- `scope` is a string to define substituions for metric strings. Default is "%a.%h.%w".
  The scope supports the following substitutions prepended to metrics names:

  %p for process ID
  %w for worker ID
  %a for app name
  %h for hostname

See Integrating with third-party consoles for more information.

`--no-channel`
Do not listen for run-time control messages (default is to listen on "runctl" when clustered).

`--no-control`
Do not listen for local control messages.

`--no-profile`
Do not profile with StrongOps (default is to profile if registration data is found).

`--no-timestamp-workers`
Disable timestamping of worker log lines by supervisor.

`--no-timestamp-supervisor`
Disable timestamping of supervisor log messages.

`-p, --pid <file>`
Write supervisor's process ID to *file.* Fails if *file* already has a valid process ID in it.

`--syslog`
Send supervisor and collected worker logs to syslog(3).

STANDARD OPTIONS

`-h, --help`
Display help information.

`-v, --version`
Display version number.

slc runctl

⚠ This command is deprecated in favor of slc ctl; please use that command instead.

This command was previously named `clusterctl`.

slc start

Start application in the background under the Process Manager.

Process Manager will listen for control messages on the default port, 8701.

See also: Running local apps with slc

SYNOPSIS

```
slc start [app]
```

If not specified, the application in the current working directory is started.

STANDARD OPTIONS

`-h, --help`
Display help information.

`-v, --version`
Display version number.

slc strongops

This command is deprecated. Please update your installation with `npm install -g strongloop`.

To get your license key, use StrongLoop Arc. See Managing your licenses for more information.

StrongOps hosted monitoring is no longer supported. Use Arc and on-premises monitoring instead; see:

- Monitoring app metrics
- Setting up monitoring

SYNOPSIS

```
slc strongops [options]
```

Prompts for the email address and password of your StrongOps account, and saves the credentials in the specified configuration file.

There are no interactive prompts for data specified on the command line.

OPTIONS

`–email`
Specify the email address, e.g.: `--email someone@strongloop.com`. The address found in your `~/.gitconfig` or `~/.npmrc` is

offered as the default.

`–password`
Specify your StrongOps password, e.g.: `--password 12345678`

`–nosave`
Prevent saving of StrongOps account credentials, this overrides any save option.

`–local`
Saves StrongOps account credentials in a `./strongloop.json` file. This is set by default, if no other save options exist.

`–package`
Saves StrongOps account credentials in `./package.json` file, if the file exists.

`–global`
Saves StrongOps account credentials in a `~/strongloop.json` file.

STANDARD OPTIONS

`-h, --help`
Display help information.

`-v, --version`
Display version number.

sl-central

Run Strongloop Central.

You can control StrongLoop Central via HTTP on the specified port. You can also deploy applications (using slc deploy) to Executors under management of StrongLoop Central.   Enable HTTP basic authentication by setting the STRONGLOOP_PM_HTTP_AUTH environment variable to <*username*>:<*password*> (for example, strong-central:super-secret).

You can also control StrongLoop Central using local domain sockets, which look like file paths. You can change or disable the listen path. Local domain sockets do not support HTTP authentication.

SYNOPSIS

```
sl-central [options]
```

OPTIONS

`-b, --base <base-dir>`
Base directory (default is `.strong-central`) where the StrongLoop Central saves deployed applications, creates working directories, and creates other files.

`-l, --listen <port>`
Port on which to listen for Git pushes (default is 8701).

STANDARD OPTIONS

`-h, --help`
Display help information.

`-v, --version`
Display version number.

sl-central-install

Install StrongLoop Central as a service.

SYNOPSIS

```
sl-central-install [options]
```

OPTIONS

The `--systemd` and `--upstart` options are mutually exclusive. If you don't specify either option, the service is installed as an Upstart job using a template that assumes Upstart 1.4 or higher.

`-b, --base <`*`base-dir`*`>`
Base directory in which to work. Default is $HOME of the user that central is run as, see `--user`.

`-f, --force`
Overwrite existing job file if present.

`--http-auth <`*`credentials`*`>`
Enable HTTP authentication using HTTP basic-auth, requiring the specified credentials for every request sent to the REST API, where <credentials> has the form of `<`*`username`*`>:<`*`password`*`>`.

`-j, --job-file <`*`file`*`>`
Path of Upstart job to create. Default is `/etc/init/strong-central.conf`.

`-n, --dry-run`
Don't write any files.

`-p, --port <`*`port`*`>`
Listen on <*port*>. Default is 8701.

`--systemd`
Install as a Systemd service, not an Upstart job.

`-u, --user <`*`user`*`>`
User account under which to run StrongLoop Central. Default is strong-central.

`--upstart <`*`version`*`>`
Specify Upstart version, 1.4 or 0.6. Default is 1.4.

STANDARD OPTIONS

`-h, --help`
Display help information.

`-v, --version`
Display version number.

sl-executor

The Strongloop executor.

SYNOPSIS

```
sl-executor [options]
```

OPTIONS

`-b, --base <`*`base-dir`*`>`
Base directory (default is `.strong-executor`) where the executor saves deployed applications creates working directories and for any other files it needs.

`-C, --control <url>`
Connect to central at this URL.

`-P, --base-port <`*`port`*`>`
Run applications on port number <*port*> + service ID. Default base port is 3000.

STANDARD OPTIONS

`-h, --help`
Display help information.

`-v, --version`
Display version number.

sl-executor-install

Install Strongloop Executor.

SYNOPSIS

```
sl-executor-install [options]
```

OPTIONS

`-b, --base <base-dir>`
Base directory (default is home directory of the user that executor is run as, see --user) where the executor saves deployed applications, creates working directories, and creates any other files it needs.

`-C, --control <url>`
Connect to central at this URL.

`-f, --force`
Overwrite existing job file if present.

`-j, --job-file <file>`
Path of Upstart job to create (default is `/etc/init/strong-executor.conf`).

`-n, --dry-run`
Don't write any files.

`-P, --base-port <port>`
Run applications on port number <*port*> + service ID. Default base port is 3000.

`--systemd`
Install as a Systemd service, not an Upstart job.

`-u, --user <user>`
User account under which to run StrongLoop Executor. Default is strong-executor.

`--upstart <version>`
Specify Upstart version, 1.4 or 0.6 (default is 1.4).

OS Service support

The --systemd and --upstart VERSION options are mutually exclusive. If neither is specified, the service is installed as an Upstart job using a template that assumes Upstart 1.4 or higher.

STANDARD OPTIONS

`-h, --help`
Display help information.

`-v, --version`
Display version number.

sl-meshadm

Administer Strongloop Mesh.

SYNOPSIS

```
sl-meshadm [options] [sub-command ...]
```

OPTIONS

`-C, --control <ctl>`
Control endpoint for StrongLoop Process Manager.

STANDARD OPTIONS

`-h, --help`
Display help information.

`-v, --version`
Display version number.

If you don't specify the StrongLoop Process Manager control endpoint with the option, the command uses the default endpoints in this order of precedence:

- STRONGLOOP_PM in environment: may be a local domain path, or an HTTP URL.
- `http://localhost:8701`: a process manager running on localhost

SUB-COMMANDS

Use the exec-list command to determine <*exec-id*>, the executor ID.

Sub-command	Description
exec-create	Create an executor ID.
exec-remove <*exec-id*>	Remove specified executor ID.
exec-list	List executors.
exec-shutdown <*exec-id*>	Shutdown specified executor.

sl-meshctl

Control StrongLoop Executor. This command enables you to:

- Start, stop, and restart applications that are under management of StrongLoop PM.
- Start and stop application clusters; change number of workers in a cluster.
- Start and stop CPU profiling, take heap snapshots, and perform object tracking.
- Modify environment settings while an application is running.

See also:

- Using Process Manager
- Clustering
- Logging
- Profiling with slc
- Monitoring with slc

This command works only with applications running under control of StrongLoop Process Manager. For local use, that means you must run the application with `slc start`, not `node .` or `slc run` (that don't run applications in Process Manager).

SYNOPSIS

```
slc meshctl [options] [sub-command]
```

OPTIONS

`-C, --control <ctl>`
Control endpoint for Process Manager. For a remote Process Manager, this must specify the URL on which the Process Manager is listening.

If Process Manager is using HTTP authentication then you must set valid credentials in the URL, in the form `http://`*`username`*`:`*`password`*`@example.com:7654`.

To tunnel over SSH using an HTTP URL, use the protocol `http+ssh`., for example `http+ssh://example.com:7654`.

- The SSH username defaults to your current user. Override the default with the SSH_USER environment variable.
- Authentication defaults to your current `ssh-agent`. Override the default with the SSH_KEY environment variable specifying the path of an existing private key to use.

Use the STRONGLOOP_PM environment variable to set a default value for the `--control` option; this eliminates the need to supply the option every time.

If you don't specify a channel with this option, the tool uses the following in this order of precedence:

1. STRONGLOOP_PM environment variable, that can specify a local domain path, or an HTTP URL. Use an HTTP URL to specify a

remote Process Manager. Use `localhost` for a local Process Manager. The URL must specify at least the process manager's listen port, such as `http://example.com:7654` (default is 8701).
2. `./pmctl`: Process Manager running the current working directory, if any.
3. `~/.strong-pm/pmctl`: Process Manager running in the user's home directory.
4. `/var/lib/strong-pm/pmctl`: Process Manager installed by `slc pm-install`.
5. `http://localhost:8701`: Process Manager running on localhost.

STANDARD OPTIONS

`-h, --help`
Display help information.

`-v, --version`
Display version number.

SUB-COMMANDS

The default sub-command is `status`.

This command has three types of sub-commands:

- Global commands that apply to Process Manager itself.
- Commands that apply to a specific service.
- Commands that apply to a specific worker process.

When you deploy an application to Process Manager, you give the deployed application instance a name, referred to as the *service name* and indicated in command arguments as *`<service>`*. By default, it is the `name` property from the application's `package.json`.

Process Manager also automatically generates an integer ID for each application it's managing. Typically, the IDs start with one (1) and are incremented with each application deployed; however, the value of ID is not guaranteed. Always determine it with `slc ctl status` once you've deployed an app.

A service becomes available over the network at `http://hostname:port` where:

- `hostname` is the name of the host running Process Manager
- `port` is 3000 + service ID.

For example, if Process Manager is running on my.host.com, then service ID 1 is available at `http://my.host.com:3001`, service ID 2 at `http://my.host.com:3002`, and so on.

Command	Description	Arguments
Global sub-commands		
info	Display information about Process Manager.	
ls	List services under management.	
shutdown	Stop the process manager and all applications under management.	
Service sub-commands (apply to a specific service) The argument <*service*> is the name or ID of a service.		
create <*service*>	Create application instance *service*.	<*service*>, name or ID of the service to create.
cluster-restart <*service*>	Restart the current application cluster workers.	<*service*>, name of target service.
env[-get] <*service*> [*env*...]	List specified environment variables for <*service*>. If none are given, list all variables.	<*service*>, name or ID of target service. <*env*>, one or more environment variables.
env-set <*service*> <*env*>=<*val*>...	Set one or more environment variables for <*service*> and hard restart it with new environment.	<*service*>, name or ID of target service. One or more environment variables, <*env*>, and corresponding value <*val*>.

env-unset <*service*> <*env*>...	Unset one or more environment variables for <*service*> and hard restart it with the new environment.	<*service*>, name or ID of target service. <*env*>, one or more environment variables.
log-dump <*service*> [--follow]	Empty the log buffer, dumping the contents to stdout. Use `--follow` to continuously dump the log buffer to stdout.	<*service*>, name or ID of target service.
npmls <*service*> [*depth*]	List dependencies of service with id <*service*>	<*service*>, name or ID of target service. *depth*, an integer limit of levels for which to list dependencies; default is no limit.
remove <*service*>	Remove <*service*>.	<*service*>, name or ID of target service.
restart <*service*>	Hard stop the current application: kill the supervisor and its workers with SIGTERM; then restart the current application with new configuration.	<*service*>, name or ID of target service.
set-size <*service*> <*n*>	Set cluster size for <*service*> to <*n*> workers.	<*service*>, name or ID of target service. <*n*>, positive integer.
start <*service*>	Start <*service*>.	<*service*>, name or ID of target service.
status [*service*]	Report status. This is the default command.	*service*, optional name of target service. Default is to show status for all services.
stop <*service*>	Hard stop <*service*>: Kill the supervisor and its workers with SIGTERM.	<*service*>, name or ID of target service.
soft-stop <*service*>	Notify workers they are being disconnected, give them a grace period to close existing connections, then stop the current application.	<*service*>, name or ID of target service.
soft-restart <*service*>	Notify workers they are being disconnected, give them a grace period to close existing connections, then restart the current application with new configuration.	<*service*>, name or ID of target service.
tracing-start <*service*>	Restart all workers with tracing on.	<*service*>, name or ID of target service.
tracing-stop <*service*>	Restart all workers with tracing off.	<*service*>, name or ID of target service.
Worker process sub-commands (apply to a specific worker process) The argument <*worker*> is a worker specification; either: • <*service_id*>.1.<*worker_id*>, where <*service_id*> is the service ID and <*worker_id*> is the worker ID. • <*service_id*>.1.<*process_id*>, where <*service_id*> is the service ID and <*process_id*> is the worker process ID (PID).		
cpu-start <*worker*> [*timeout* [*stalls*]]	Start CPU profiling on worker or process ID <*id*>. Use cpu-stop to save the profile data. NOTE: Requires Node version 0.11 or higher. Saves profiling data to a file you can view with Chrome Dev Tools. See CPU profiling for more information.	<*worker*>, a worker specification (see above). **Linux only**: [*timeout*], timeout period (ms) for Smart profiling. Start CPU profiling when the specified process's Node event loop stalls for more than the specified timeout period. [*stalls*], number of event loop stalls after which the profiler will be stopped automatically (default is 0, never auto-stop). For more information, see Smart profiling with slc.

cpu-stop <*worker*> [*filename*]	Stop CPU profiling on worker process <*id*> and save results in file `name.cpuprofile`.	<*worker*>, a worker specification (see above). *filename*, optional base file name; default is `node.<id>.cpuprofile`.
heap-snapshot <*worker*> [*filename*]	Save heap snapshot data for worker process <*id*> and save results in file `name.heapsnapshot`. Saves profiling data to a file you can view with Chrome Dev Tools. See Heap memory profiling for more information.	<*worker*>, a worker specification (see above). *filename*, optional base file name; default is `node.<id>.heapsnapshot`.
objects-start <*worker*>	Start tracking objects on worker process <*id*>.	<*worker*>, a worker specification (see above).
objects-stop <*worker*>	Stop tracking objects on worker process <*id*>.	<*worker*>, a worker specification (see above).
patch <*worker*> <*file*>	Apply patch <*file*> to <*id*> to get cust om metrics.	<*worker*>, a worker specification (see above). <*file*>, file name.

sl-nginx-ctl

Prequisite
You must first install the StrongLoop Nginx Controller. See Configuring Nginx load balancer.

Run StrongLoop NGINX controller.

See also sl-nginx-ctl-install to install the controller as a service.

SYNOPSIS

```
sl-nginx-ctl [options]
```

OPTIONS

`-b, --base <base>`
Base directory in which to work. Default without this option is `.strong-nginx-controller`.

`-c, --control <control>`
Control API endpoint URL. Default without this option is `http://0.0.0.0:0`.

`-l, --listen <endpoint>`
Listen endpoint URL for incoming HTTP traffic. Default without this option is `http://0.0.0.0:8080`.

`-x, --nginx <nginx-path>`
Path to Nginx binary. Default without this option is `/usr/sbin/nginx`.

STANDARD OPTIONS

`-h, --help`
Display help information.

`-v, --version`
Display version number.

sl-nginx-ctl-install

Prequisites
You must first install the StrongLoop Nginx Controller. See Configuring Nginx load balancer.

This command is supported only on Linux distributions that support Upstart or systemd, including RedHat-based distributions (CentOS, RHEL, Fedora, and so on), Debian-based distributions (Ubuntu, Mint, and so on).

Install the StrongLoop Nginx Controller as an operating system service.

See also sl-nginx-ctl to run the controller as a transient process.

SYNOPSIS

```
sl-nginx-ctl-install [options]
```

OPTIONS

`-b, --base <base>`
Base directory in which to work. Default without this option is `.strong-nginx-controller`.

`-c, --control <control>`
Control API endpoint. Default without this option is `http://0.0.0.0:0/`.

`-u, --user <user>`
User account under which to run manager. Default without this option is `strong-nginx-controller`.

`-l, --listen <endpoint>`
Listen endpoint for incoming HTTP traffic. Default without this option is: `http://0.0.0.0:8080`.

`-n, --dry-run`
Don't write any files.

`-j, --job-file <file>`
Path of Upstart job to create. Default without this option is `/etc/init/strong-nginx-controller.conf`.

`-f, --force`
Overwrite existing job file if present.

`-x, --nginx` ***<nginx-path>***
Path to Nginx binary. Default without this option is `/usr/sbin/nginx`.

`--upstart <version>`
Specify the version of Upstart, either 1.4 or 0.6. Default without this option is 1.4.

`--systemd`
Install as a systemd service, not an Upstart job.

OS Service support

The `--systemd` and `--upstart` options are mutually exclusive. If you specify neither, the command installs the service as an Upstart job using a template that assumes Upstart 1.4 or higher.

STANDARD OPTIONS

`-h, --help`
Display help information.

`-v, --version`
Display version number.

sl-pm

This command is available only when you have installed StrongLoop Process Manager with `npm install -g strong-pm` as described in Setting up a production host.

Run StrongLoop Process Manager, when installed as a standalone module (typically on a production host). The options and sub-commands are the same as `slc pm`.

See slc pm for more information.

sl-pmctl

This command is available only when you have installed StrongLoop Process Manager with `npm install -g strong-pm` as described in Setting up a production host.

Control StrongLoop Process Manager.   The options and sub-commands are the same as `slc` ctl. See slc ctl for more information.

sl-pm-install

This command is available only when you have installed StrongLoop Process Manager with `npm install -g strong-pm` as described in Setting up a production host.

Install StrongLoop Process Manager as an operating system service. The options and sub-commands are the same as `slc pm-install`.

See slc pm-install for more information.

Arc release notes

- Known issues
- How to report an issue
- Upcoming features

Known issues

This is a list of known issues (bugs). Please check this list before reporting a new issue. For the complete list of open issues, see https://github.com/strongloop/strong-arc/issues.

Related articles:

- General
- Composer
- Build & Deploy

General

- StrongLoop Arc does not run in Internet Explorer 8 or Internet Explorer 9. You must use IE 10+.

Composer

Creating models and properties:

- If you choose a base model that is not compatible with the data source, Arc should give error, but does not.
- Model property type field should be populated via loopback-workspace API.
- If you choose a base model that's a custom model, can't validate. See issue #34.

Data sources:

- Error messages shown are global, not specific to the data source you are currently editing. That is, you may see error an message stating that you need to install a connector, even when testing connection for another connector that *is* installed.
- Composer should show only data sources that are installed as part of the project.

Build & Deploy

- You cannot deploy applications to Windows systems.
- When using the Oracle connector on a remote host, you must first configure the environment variable (with the `slc ctl env-set` command) and make sure the dynamic libraries from Oracle Instant Client are available for the Node process.

To use API Explorer within Arc, if you created your project before Oct 9, 2014, you must first run `npm update` in your project.

How to report an issue

If you discover a bug or other issue:

Related articles:

1. Review issues that have already been reported:
   - Known issues.
   - GitHub issues at https://github.com/strongloop/strong-arc/issues.
2. If your issue has not been reported, go to GitHub Issues.
3. Click the big green **New** button to report a new issue: https://github.com/strongloop/strong-arc/issues/new.

   - Try to describe the issue in as much detail as possible. If a bug, provide steps to reproduce.
   - Apply the appropriate label: bug, enhancement, question, and so on. If unsure, leave this field empty.
   - Leave the **Assignee** and **Milestone** fields blank.

Thank you!

Upcoming features

Related articles:

The following Arc Composer features are planned for future releases:

- LoopBack API Explorer integrated.
- Scaffold a static web application based on your LoopBack models and data sources.
- Enter data for the models you have created through a form-based web application.
- Canvas view.

The following features will be added in subsequent releases. Specific timing and implementation have not yet been determined.

General

- ACL editor
- Relations editor
- Scopes editor
- Index editor
- Support for validation in LDL

Data sources

- Support for email data sources.
- Manage connector dependencies.

Latest documentation updates

LoopBack	StrongLoop Controller Strong agent	Other Node resources
component-config.json yesterday at 4:31 PM • updated by Rand McKinney • view change Using promises yesterday at 3:57 PM • updated by Rand McKinney • view change LoopBack components yesterday at 3:47 PM • updated by Rand McKinney • view change Remote hooks yesterday at 3:08 PM • updated by Rand McKinney • view change Operation hooks yesterday at 2:50 PM • updated by Rand McKinney • view change Define a remote hook yesterday at 1:59 PM • updated by Rand McKinney • view change	API Analytics Sep 08, 2015 • updated by Rand McKinney • view change Process Manager release notes Sep 04, 2015 • updated by Rand McKinney • view change Using Process Manager Aug 31, 2015 • updated by Rand McKinney • view change Monitoring with slc Aug 31, 2015 • updated by Rand McKinney • view change Setting up monitoring Aug 31, 2015 • updated by Rand McKinney • view change _Gateway icon Aug 27, 2015 • created by Rand McKinney	slc pm Sep 10, 2015 • updated by Rand McKinney • view change slc ctl Sep 10, 2015 • updated by Rand McKinney • view change Frequently asked questions Sep 09, 2015 • updated by Rand McKinney • view change slc build Aug 06, 2015 • updated by Rand McKinney • view change sl-meshadm Aug 05, 2015 • updated by Rand McKinney • view change sl-executor-install Aug 03, 2015 • updated by Rand McKinney • view change

- Storage component
  Sep 15, 2015 • updated by Rand McKinney • view change
- Controlling data access
  Sep 15, 2015 • updated by Shared community account • view change
- Defining middleware
  Sep 14, 2015 • updated by Rand McKinney • view change
- Environment-specific configuration
  Sep 14, 2015 • updated by Rand McKinney • view change
- SOAP connector
  Sep 14, 2015 • updated by Rand McKinney • view change
- SOAP connector
  Sep 11, 2015 • updated by Raymond Feng • view change
- LoopBack core concepts
  Sep 10, 2015 • updated by Rand McKinney • view change
- StrongLoop Labs
  Sep 10, 2015 • updated by Rand McKinney • view change
- Where filter
  Sep 09, 2015 • updated by Rand McKinney • view change
- Installing StrongLoop
  Sep 09, 2015 • updated by Rand McKinney • view change
- _Supported platforms
  Sep 09, 2015 • updated by Rand McKinney • view change
- PersistedModel REST API
  Sep 08, 2015 • updated by Rand McKinney • view change
- Exposing models over REST
  Sep 08, 2015 • updated by Rand McKinney • view change
- HasOne relations
  Sep 08, 2015 • updated by Raymond Feng • view change
- Tracing
  Aug 25, 2015 • updated by Rand McKinney • view change
- Metrics API
  Aug 20, 2015 • updated by Rand McKinney • view change
- Viewing metrics with Arc
  Aug 20, 2015 • updated by Rand McKinney • view change
- Available metrics
  Aug 20, 2015 • updated by Rand McKinney • view change
- Redis metrics.png
  Aug 20, 2015 • attached by Rand McKinney
- DB metrics.png
  Aug 20, 2015 • attached by Rand McKinney
- HTTP metrics.png
  Aug 20, 2015 • attached by Rand McKinney
- Smart profiling with Arc
  Aug 17, 2015 • updated by Rand McKinney • view change
- Smart profiling with slc
  Aug 17, 2015 • updated by Rand McKinney • view change
- Setting up a production host
  Aug 13, 2015 • updated by Rand McKinney • view change
- Operating Node applications
  Aug 12, 2015 • updated by Rand McKinney • view change
- Scaling with mesh
  Jul 23, 2015 • updated by Rand McKinney • view change
- Controlling Process Manager
  Jul 20, 2015 • updated by Rand McKinney • view change
- Command references
  Jul 17, 2015 • updated by Rand McKinney • view change
- sl-executor
  Aug 03, 2015 • updated by Rand McKinney • view change
- slc pm-install
  Aug 03, 2015 • updated by Rand McKinney • view change
- slc deploy
  Aug 03, 2015 • updated by Rand McKinney • view change
- slc strongops
  Aug 03, 2015 • updated by Rand McKinney • view change
- Command-line reference
  Aug 03, 2015 • updated by Rand McKinney • view change
- slc run
  Aug 03, 2015 • updated by Rand McKinney • view change
- Glossary
  Aug 03, 2015 • updated by Rand McKinney • view change
- Environment variables
  Aug 03, 2015 • updated by Rand McKinney • view change
- Setting up StrongOps monitoring
  Aug 03, 2015 • created by Rand McKinney
- Redirects
  Aug 03, 2015 • updated by Rand McKinney • view change
- StrongOps hosted monitoring
  Aug 03, 2015 • created by Rand McKinney
- sl-meshctl
  Jul 24, 2015 • updated by Rand McKinney • view change
- sl-central
  Jul 23, 2015 • updated by Rand McKinney • view change
- sl-central-install
  Jul 17, 2015 • updated by Rand McKinney • view change

Other StrongLoop modules

StrongLoop created and maintains many open-source modules. In addition to the LoopBack modules, StrongLoop curates several modules for clustering plus several general-purpose Node modules.

The StrongLoop modules depend on a number of important open-source community modules; see Supporting module reference for a list.

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.

Other modules

- Node-inspector - Debugging interface for Node.js.
- Strong-mq - An abstraction layer over common message distribution patterns.
- 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.

Strong Pubsub

Installation

```
$ npm install strong-pubsub
```

Use

NOTE: until version `1.0.0` the API may change!

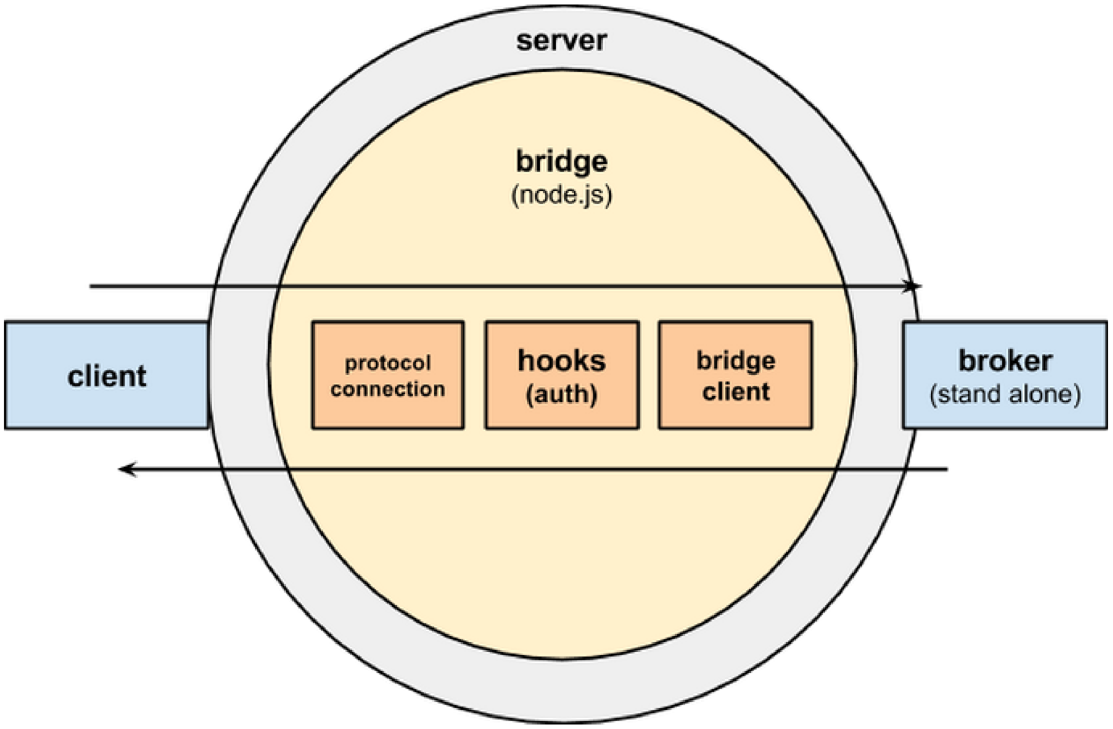
Client (strong-pubsub)

The `Client` class provides a unified pubsub client in Node.js and browsers. It supports subscribing to topics or topic patterns (topics and wildcards). Clients can connect to brokers or bridges that support the `client.adapter`'s protocol.

Bridge (strong-pubsub-bridge)

In some cases, clients should not connect directly to a message broker. The Bridge class allows you to create a bridge between a client connecting to your **node.js** server and a broker. It supports hooks for injecting logic before the bridge performs an action on behalf of the client. Hooks allow you to implement client authentication and client action (publish, subscribe) authorization using vanilla node.js (usually in place of broker specific access controls and authentication).

Bridges also allow clients to connect to brokers over a protocol that the broker may not support. For example, a client can connect to the bridge using one protocol (eg. MQTT) and the bridge will connect to the broker using another (eg. **Redis** or **STOMP**).



Note: It is not possible to guarantee all features when bridging connections of one protocol to a broker that speaks another protocol. For example MQTT quality of service (`options.qos`) will be not be guaranteed when a bridge is accepting MQTT protocol connections and bridging to a redis broker.

Creating a bridge

Here is an example setting up a bridge. This would bridge messages between MQTT clients and a RabbitMQ server.

This example shows how to connect a `Client` to a `Bridge` and then a `Bridge` to a broker (using another `Client`).

Message broker

To distribute a message published to a topic, a client connects to a message broker.
Client adapters allow pubsub clients to connect to various brokers. Clients can connect directly
to brokers or indirectly using a bridge.

Adapter

Client adapters implement the `Client` API in a broker protocol-specific way.

Specify the adapter specific options using the name as the key.

Protocols

Strong-pubsub supports these two protocols:

- MQTT
- STOMP

It is possible to extend **strong-pubsub** to support other protocols, but that is beyond the scope of this README.

Transports

Adapters / Clients require a tranpsort to create a connection and read / write data to / from.

A module (or object) that implements `net.createConnection()`.

- The standard TCP protocol: **`require('net')`**
- Transport Layer Security (TLS), a secure cryptographic protocol: **`require('tls')`**
- Primus (in development): **`require('strong-pubsub-transport-primus')`**

Transport swapping

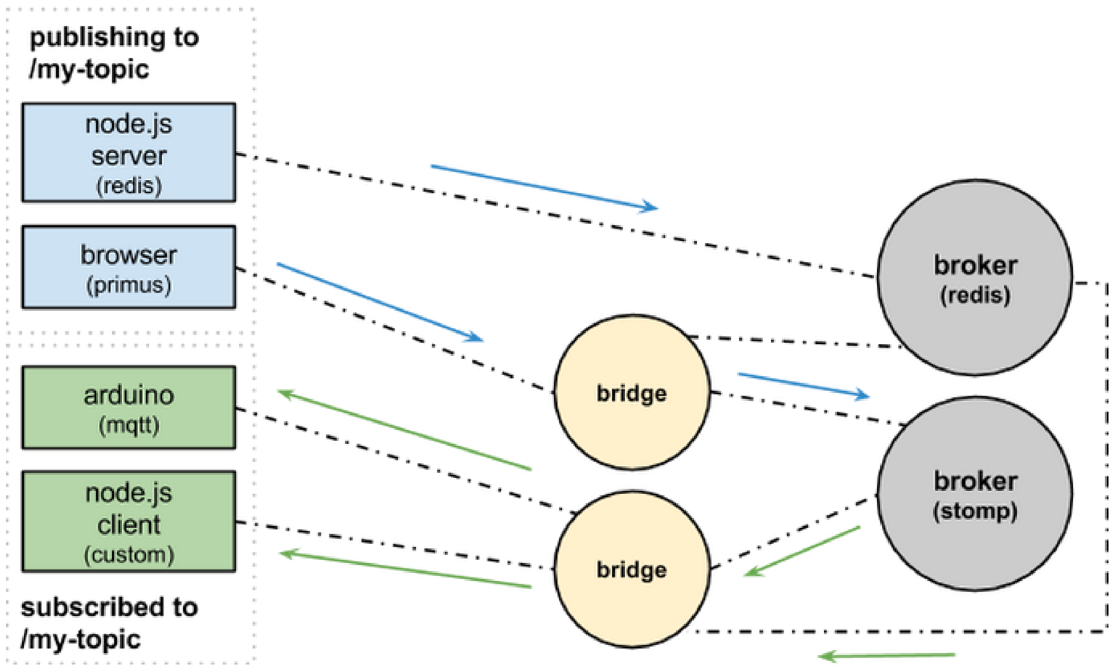
This example shows how to switch betweenn different transports on the client.

Connection

A Protocol connection implements a specific pubsub protocol in Node.js for use by strong-pubsub-bridge.

Architecture

This diagram illustrates how messages flow between clients, bridges, servers and brokers.
The blue arrows represent a message published to a topic. The green arrow represents the message being sent to a subscriber.



Modules / Plugins

- strong-pubsub-bridge
- strong-pubsub-mqtt
- strong-pubsub-redis
- strong-pubsub-connection-mqtt

Examples

See the strong-pubsub-example repo.

Zones

StrongLoop zone library

Overview

The Zone library provides a way to represent the dynamic extent of asynchronous calls in Node. Just like the scope of a function defines where it may be used, the extent of a call represents the lifetime that is it active.

A Zone also provides execution context that can persist across the lifecycle of one or more asynchronous calls. This is similar to the concept of thread-local data in Java.

Zones provide a way to group and track resources and errors across asynchronous operations. In addition zones:

- Enables more effective debugging by providing better stack traces for asynchronous functions
- Makes it easier to write and understand asynchronous functions for Node applications
- Makes it easier to handle errors raised asynchronously and avoid resulting resource leaks
- Enables you to associate user data with asynchronous control flow

Dart's async library and Brian Ford's zone.js library provide similar functionality.

Implementation status

- The zone library and documentation are still under development: there are bugs, missing features, and limited documentation.
- The zone library dynamically modifies Node's asynchronous APIs at runtime.
  As detailed below, some of the modules have not yet been completed, and thus you cannot use them with zones.

The following modules have not been zone enabled:

- cluster
- crypto: **`pbkdf2`**, **`randomBytes`**, **`pseudoRandomBytes`**
- fs: **`fs.watch`**, **`fs.watchFile`**, **`fs.FSWatcher`**
- process object: **`process.on('SIGHUP')`** and other signals.
- tls / https
- udp
- zlib

Using zones

To use zones, add the following as the very first line of your program:

The zone library exports a global variable, `zone`. The `zone` global variable always refers to the currently active zone. Some methods that can always be found on the 'zone' object are actually static methods of the `Zone` class, so they don't do anything with the currently active zone.

After loading the zone library the program has entered the 'root' zone.

Creating a zone

There are a few different ways to create a zone. The canonical way to create a one-off zone is:

The zone constructor function is called synchronously.

Defining zone functions

Under some circumstances it may be desirable to create a function that is always wrapped within a zone.
The obvious way to do this:

To make this a little less verbose there is the ‘zone.define()’ API.
With it you can wrap a function such that when it’s called a zone is created.
Example:

Now you can use this zone template as follows:

Obtaining the result of a zone

Zones are like asynchronous functions. From the outside perspective, they can return a single value or “throw” a single error. There are a couple of ways the outside zone may obtain the result of a zone. When a zone reports its outcome:

No more callbacks will run inside the zone.

All non-garbage-collectable resources have been cleaned up.

Zones also automatically exit when no explicit value is returned.

A way to obtain the outcome of a zone is:

You can also use the `then` and `catch` methods, as if it were a promise.
Note that unlike promises you can’t currently chain calls callbacks.

Sharing resources between zones

Within a zone you may use resources that are “owned” by ancestor zones. So this is okay:

However, using resources owned by child zones is not allowed:

NOTE: Currently zones don’t always enforce these rules, but you’re not supposed to do this. It would also be dumb, since the server will disappear when `SomeZone()` exits itself!

The rules of engagement

It is okay for a zone to temporarily enter an ancestor zone. It is not allowed to enter child zones, siblings, etc. The rationale behind this is that when a zone is alive its parent must also be alive. Other zones may exit unless they are aware that code will run inside them.

Exiting a zone

There are a few ways to explicitly exit a zone:

- **`zone.return(value)`** sets the return value of the zone and starts cleanup.

- **`zone.throw(error)`** sets the zone to failed state and starts cleanup. **`zone.throw`** itself does not throw, so statements after it will run.

- **`throw error`** uses normal exception handling. If the exception is not caught before it reaches the binding layer, the active zone is set to failed state and starts cleanup.

- **`zone.complete(err, value)`** is a zone-bound function that may be passed to subordinates to let them exit the zone.

A rather pointless example:

This is equivalent to:

To run code in a child zone, you can use `zone.bindCallback` and `zone.bindAsyncCallback` to create a callback object which can be invoked from a parent zone.

Sharing resources between zones

Within a zone you may use resources that are “owned” by ancestor zones. So this is okay:

However, using resources owned by child zones is not allowed:

NOTE: Currently zones don’t always enforce these rules, but you’re not supposed to do this.
It would also be dumb, since the server will disappear when `SomeZone()` exits itself!

Zone.data

zone.data is a magical property that that associates arbitrary data with a zone.
In a way you can think of it as the ‘scope’ of a zone. Properties that are not explicitly defined within the scope of a zone are inherited from the parent zone.

- In the root zone, **`zone.data`** equals the global object.

- In any other zone, **`zone.data`** starts off as an empty object with the parent zone’s **`data`** property as it’s prototype.

- In other words, **`zone.data.__proto__ === zone.parent.data`**.

Zones examples

Examples

Zones and express

This example show a very simple case where Zones can:

- Store user provided data with different lifetimes

- Capture unexpected Exceptions in asynchronous callbacks

- Provide a long stack-trace of exceptions

Example code:
```js
require(‘zone’).enable();
express = require(‘express’);
var Zone = zone.Zone;
```

~~~
var app = express();
var router = express.Router();
Zone.longStackSupport = true;

//Initialize the Request id in the root zone.
//This value will be available to all child zones.
zone.data.requestId = 0;

app.use(function(req, res, next) {
//Increment the request ID for every new request
++zone.data.requestId;

//Create a new Zone for this request
zone.create(
function RequestZone() {
//Store the request URL in the Request zone
//This value will be only to this zone and its children
zone.data.requestURL = req.url;
~~~

~~~
//Continue to run other express middleware within this child zone
         next();
       })
  .then(
      //The call was succesful
      function successCallback(err) {
        res.write('Transaction succesful\n');
        res.end();
      },

      //An error was thrown while processing this request
      function errorCallback(err) {
        res.write('Transaction failed\n');
        res.write('x' + err.zoneStack + '\n');
        res.end();
      });
~~~

~~~
});

router.get('/', function(req, res) {
if (Math.random() > 0.5) {
//Simulate some async I/O call that throws an exception
process.nextTick(function() { throw new Error("monkey wrench"); });
}

res.write('Running request #' + zone.data.requestId + ' within zone: ' +
zone.name + ' (URL:' + zone.data.requestURL + ')\n');
});

app.use('/', router);
app.listen(3001);
```
~~~

Output:
```sh

~~~
curl localhost:3001?q=ghi
Running request #3 within zone: RequestZone (URL:/?q=ghi)
Transaction failed
xError: monkey wrench
at Zone. (/scratch/server.js:44:41)
at Zone._apply (/scratch/node_modules/zone/lib/zone.js:588:15)
at Zone.apply (/scratch/node_modules/zone/lib/zone.js:611:23)
at processCallbacks (/scratch/node_modules/zone/lib/scheduler.js:47:10)
at processQueues (/scratch/node_modules/zone/lib/scheduler.js:67:5)
at process._tickCallback (node.js:343:11)
In zone: RequestZone
at Zone.create (/scratch/node_modules/zone/lib/zone.js:273:10)
at Layer.handle (/scratch/server.js:17:8)
at trim_prefix (/scratch/node_modules/express/lib/router/index.js:226:17)
at /scratch/node_modules/express/lib/router/index.js:198:9
at Function.proto.process_params (/scratch/node_modules/express/lib/router/index.js:251:12)
at next (/scratch/node_modules/express/lib/router/index.js:189:19)
~~~
```

> *at Layer.expressInit as handle*
> *at trim_prefix (/scratch/node_modules/express/lib/router/index.js:226:17)*
> *at /scratch/node_modules/express/lib/router/index.js:198:9*
> *at Function.proto.process_params (/scratch/node_modules/express/lib/router/index.js:251:12)*
>
> *curl localhost:3001?q=klm*
> *Running request #4 within zone: RequestZone (URL:/?q=klm)*
> *Transaction succesful*

```
### Inspecting a running application

This showcase demonstrates what the _inspect_ tool does.

__demo.js__:

The demo script spins up a TCP server and creates a bunch of clients that send data to the server on an
interval basis.
```

require('zone').enable(); // enable zones

var Zone = zone.Zone;
var net = require('net');

Zone.longStackSupport = true;

zone.create(function ServerZone() {
var server = net.createServer(function(conn) {
conn.resume();
});

server.listen(3001);
});

for (var i = 0; i < 10; i++) {
zone.create(function ConnectionZone() {
var conn = net.connect(3001, function() {
zone.create(function IntervalZone() {
setInterval(function() {
conn.write('hello');
}, 1);
});
});
});
}

console.log("Run the inspect tool to see what's going on in this process.");
```

inspect.js

The inspect tool outputs a snapshot of all the asynchronous action going on inside all node processes using zones. It also displays resources (like sockets and file descriptors) that by themselves do not represent future events, but are relevant to zones because these resources are automatically cleaned up when the zone returns or throws.

The output looks like a tree, because zones act as containers for asynchronous I/O and related resources.

Entries marked with a + are active sources of events that are running inside the zone. Entries not marked with a plus sign are passive resources that don't produce callbacks, but that are cleanup up when a zone returns or throws.

Output:

```
txt > node inspect.js (2194) node /Volumes/Aether/Users/kraman/Documents/strongloop/zone.kr/scratch.js
+[Zone ] #1 Root (64 tasks, 4 external callbacks) +[ZoneCallback] #14 OnRead +[ZoneCallback] #15
OnWriteComplete +[ZoneCallback] #16 OnRead +[ZoneCallback] #17 OnWriteComplete +[Zone ] #8 DebugServerZone
```
```

```
(7 tasks, 5 external callbacks) +[ZoneCallback] #10 OnRead +[ZoneCallback] #11 OnWriteComplete
+[ZoneCallback] #12 OnConnection +[ZoneCallback] #181 OnRead +[ZoneCallback] #182 OnWriteComplete [Stream ]
Pipe server [Stream ] Pipe handle [Stream ] TTY handle (fd: 1) [Stream ] TTY handle (fd: 2) +[Zone ] #18
ServerZone (23 tasks, 23 external callbacks) +[ZoneCallback] #20 OnRead +[ZoneCallback] #21 OnWriteComplete
+[ZoneCallback] #22 OnConnection ... [Stream ] TCP server (:::3001) [Stream ] TCP handle
(::ffff:127.0.0.1:3001 <=> ::ffff:127.0.0.1:53945) [Stream ] TCP handle (::ffff:127.0.0.1:3001 <=>
::ffff:127.0.0.1:53946) [Stream ] TCP handle (::ffff:127.0.0.1:3001 <=> ::ffff:127.0.0.1:53947) [Stream ]
TCP handle (::ffff:127.0.0.1:3001 <=> ::ffff:127.0.0.1:53948) ... +[Zone ] #23 ConnectionZone (3 tasks, 2
external callbacks) +[ZoneCallback] #27 OnRead +[ZoneCallback] #28 OnWriteComplete [Stream ] TCP handle
(127.0.0.1:53945 <=> 127.0.0.1:3001) +[Zone ] #153 IntervalZone (1 tasks, 1 external callbacks)
+[ZoneCallback] #154 setInterval +[Zone ] #31 ConnectionZone (3 tasks, 2 external callbacks)
+[ZoneCallback] #35 OnRead +[ZoneCallback] #36 OnWriteComplete [Stream ] TCP handle (127.0.0.1:53946 <=>
127.0.0.1:3001) +[Zone ] #155 IntervalZone (1 tasks, 1 external callbacks) +[ZoneCallback] #156 setInterval
...
```

Other examples

Other examples are available in the Zones github repository

Strong Task Emitter

- Overview
- Installation
- Examples
- Extending TaskEmitter

See also the Strong Task Emitter API referenc
e.

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('strong-task-emitter');
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('strong-task-emitter');
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('strong-task-emitter');
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 Task Emitter API

View docs for strong-task-emitter in GitHub

Strong Task Emitter API

taskEmitter.task()

Execute a task and emit an event when it is complete.

You must provide

> A **`scope`** (eg. the fs module) and a name of a function on the scope (eg. **`'readFile'`**).

Example

```
var fs = require('fs');
var te = new TaskEmitter();

te
  .task(fs, 'readFile')
  .on('readFile', ...);
```

or

> Your task name (eg. ‘my-task’) and a function that executes the task and takes a conventional node callback (**`fn(err, result)`**).

Example

```
var te = new TaskEmitter();

te
  .task('my-task', myTask)
  .on('my-task', ...);

function myTask(fn) {
  // an async task of some sort
  setTimeout(function() {
    // callback
    fn(null, 'my result');
  }, 1000);
}
```

Example

It is safe to execute tasks in the event listener of another task as long as it is in the same tick.

```
var te = new TaskEmitter();

te
  .task(fs, 'stat', '/path/to/file-1')
  .task(fs, 'stat', '/path/to/file-2')
  .task(fs, 'stat', '/path/to/file-3')
  .on('stat', function(err, path, stat) {
    if(stat.isDirectory()) {
      // must add tasks before
      // this function returns
      this.task(fs, 'readdir', path);
    }
  })
  .on('readdir', function(path, files) {
    console.log(files); // path contents
  })
  .on('done', function() {
    console.log('finished!');
  });
```

taskEmitter.remaining()

Determine how many tasks remain.

Example

```
var te = new TaskEmitter();

te
  .task(fs, 'stat', '/path/to/file-1')
  .task(fs, 'stat', '/path/to/file-2')
  .task(fs, 'stat', '/path/to/file-3')
  .on('stat', function(err, path, stat) {
    console.log('%s is a %s', stat.isDirectory() ? 'directory' : 'file');
  })
  .on('done', function() {
    console.log('finished!');
  });

var remaining = te.remaining();

console.log(remaining); // 3
```

taskEmitter.stop()

Stop all remaining tasks.

taskEmitter.reset()

Remove all tasks and listeners.

Events

<taskName>

Emitted when the `<taskName>` has completed.

Example:

```
var te = new TaskEmitter();

te
  .task('foo', function(arg1, arg2, fn) {
    var err, result = 'foo';

    fn(err, result);
  })
  .on('error', ...)
  .on('foo', function(arg1, arg2, result) {
    // ...
  });
```

Example using the `fs` module

```
var te = new TaskEmitter();

te
  .task(fs, 'stat', '/path/to/file-1')
  .task(fs, 'stat', '/path/to/file-2')
  .task(fs, 'stat', '/path/to/file-3')
  .on('stat', function(err, path, stat) {
    console.log('%s is a %s', stat.isDirectory() ? 'directory' : 'file');
  })
  .on('done', function() {
    console.log('finished!');
  });
```

done

Emitted when all tasks are complete.

error

Emitted when any error occurs during the running of a `task()`. If this event is not handled an error will be thrown.

progress

Emitted after a task has been completed.

Example:

```
te.on('progress', function(status) {
  console.log(status);
});
```

Output:

```
{
  remaining: 4,
  total: 8,
  task: 'foo'
}
```

Strong remoting

See also Strong remoting API

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

Quick start

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

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

```
$ curl http://localhost:3000/user/greet?str=hello
```

Result:

```
{
  "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).

See the before-after example for more info.

Streams

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

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

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

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

...

Supporting module reference

StrongLoop uses a set of community modules that provide crucial functionality, as described in the following table.

Module documentation	NPM Package	Description
Express	Express	Web application framework for building single-page, multi-page, and hybrid web applications.
Connect	Connect	Extensible HTTP server framework using "plugins" known as *middleware*.
Passport	Passport	Authentication middleware for Node.
Mongoose	Mongoose	MongoDB object modeling.
Async	Async	Functions for working with asynchronous JavaScript such as map, reduce, filter, each, and so on, plus common patterns for asynchronous control flow (parallel, series, waterfall).

• Q README • Wiki	Q	Library for promises. If a function cannot return a value or throw an exception without blocking, it can return a promise instead. A promise is an object that represents the return value or the thrown exception that the function may eventually provide. A promise can also be used as a proxy for a remote object to overcome latency.
Request	Request	Request is designed to be the simplest way possible to make http calls. It supports HTTPS and follows redirects by default.
Socket.IO	Socket.IO	Makes WebSockets and realtime apps possible in browsers and mobile devices. Enhances WebSockets by providing built-in multiplexing, horizontal scalability, automatic JSON encoding/decoding, and more.
Engine.IO	Engine.IO	Implementation of transport-based cross-browser/cross-device bi-directional communication layer for Sock et.IO.
Node Inspector	Node-inspector	Debugger interface for Node using the Blink Developer Tools (formerly WebKit Web Inspector).
Heap-dump	Heapdump	Provides V8 heap dumps to help track down memory leaks and memory consumption issues.

Environment variables

Variable	Description
STRONG_AGENT_LICENSE	Deprecated. Use STRONGLOOP_LICENSE instead.
STRONGLOOP_CLUSTER	The number of workers the supervisor process will create. Determines the default cluster size if you don't provide an argument to `slc ctl set-size`. See slc ctl set-size for more information.
STRONGLOOP_LICENSE	Your StrongLoop license key(s). Separate multiple license keys with a colon (`:`).
STRONGLOOP_METRICS	Metrics URL. For more information, see Monitoring with slc.
STRONGLOOP_PM	Control channel or Process Manager URL to use. For a remote Process Manager, this must specify the URL on which the Process Manager is listening. See the slc ctl -C option for more information.
STRONGLOOP_PM_HTTP_AUTH	Set to the user name and password for secure access with HTTP authentication with the format STRONGLOOP_PM_HTTP_AUTH=*username*:*password.* See Securing Process Manager for more information.