Operating Node Applications - StrongLoop Documentation

Running local applications

This articles below describe how to run applications *locally*, that is, on your desktop system, typically during development and testing. To run a remote application (one on another system such as a production host), you need to build and deploy it, either with Arc or `slc`. See Building and deploying and Setting up a production host for more information.

You can run applications locally with Arc or with `slc`:

- Running apps with Arc
- Running apps with slc

Running apps with Arc

These are the instructions to run an app locally with Arc. To run an app remotely, you must start Process Manager on the remote system, then build and deploy the app in Arc; see Building and deploying for details.

NOTE: Because Arc app controller uses StrongLoop Process Manager, you cannot use it to run apps on Windows systems.

Follow the instructions in Using Arc to run Arc and log in.

Use Arc *App Controller* to run an application.

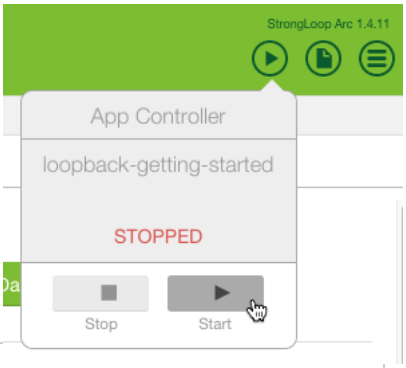
The application must have one of the following in the root directory (where you're running Arc):

- Main application script file named `server.js`, `app.js`, or `index.js`.
- A `package.json` file with a `main` property specifying the main application script file.

You can always run applications you create with the LoopBack Application generator, `slc loopback`.

To run an application:

1. Click  at the upper right of the view to display App Controller, shown at right.
2. Click the **Start** button to start the application.



When the application has initialized and is ready to accept requests, it will say RUNNING along with a link to the application root, by default http://localhost:3000/ for the local application.

Click **Stop** to stop a running application.

Typically, you'll see feedback in the console where you started Arc, for example, for a standard scaffolded LoopBack application:

```
Browse your REST API at http://localhost:3000/explorer
Web server listening at: http://localhost:3000/
```

To stop the application, click the **Stop** button.

After making changes to the application, click **Restart** button to restart the application so you can see the changes.

Running apps with slc

Run a Node application (including a LoopBack application) under control of StrongLoop PM to:

- View CPU profiles and heap snapshots to optimize performance and diagnose memory leaks.
- Keep processes and clusters alive forever.
- View performance metrics.
- Run the app as a cluster of Node processes.

For more information, see Using Process Manager.

The `slc start` command does not run on Windows systems. However, you can run StrongLoop PM on a Windows system and deploy to it. See Building and deploying for more information.

To run an app locally under control of StrongLoop Process Manager:

```
$ cd <app-root-dir>
$ slc start
```

Where `<app-root-dir>` is the application's root directory.

This starts a local instance of StrongLoop Process Manager (StrongLoop PM) and runs the specified application under its control. If PM is unable to start the application, it will periodically try to start the app until the PM is shut down.

StrongLoop PM will display some suggested commands, followed by information from the log file, for example:

```
...
--- tail of /Users/rand/.strong-pm/start.log ---
slc start(32276): StrongLoop PM v5.0.49 (API v6.1.0) on port `8701`
slc start(32276): Base folder `/Users/rand/.strong-pm`
slc start(32276): Applications on port `3000 + service ID`
Run request for commit "default-app/local-directory" on current (none)
Start Runner: commit default-app/local-directory
2015-08-14T17:07:23.361Z pid:32289 worker:0 INFO strong-agent v1.6.54 profiling app
'default-app' pid '32289'
2015-08-14T17:07:23.368Z pid:32289 worker:0 INFO strong-agent[32289] started profiling
agent
2015-08-14T17:07:23.369Z pid:32289 worker:0 INFO supervisor starting (pid 32289)
2015-08-14T17:07:23.374Z pid:32289 worker:0 INFO strong-agent strong-agent using
strong-cluster-control v2.1.2
2015-08-14T17:07:23.377Z pid:32289 worker:0 INFO supervisor reporting metrics to
`internal:`
2015-08-14T17:07:23.388Z pid:32289 worker:0 INFO supervisor size set to 1
2015-08-14T17:07:23.500Z pid:32289 worker:0 INFO supervisor started worker 1 (pid
32290)
2015-08-14T17:07:23.501Z pid:32289 worker:0 INFO supervisor resized to 1
2015-08-14T17:07:23.828Z pid:32290 worker:1 INFO strong-agent v1.6.54 profiling app
'default-app' pid '32290'
2015-08-14T17:07:23.832Z pid:32290 worker:1 INFO strong-agent[32290] started profiling
agent
```

You can also run an application from another directory. For example, if your application is in the `myApp` directory under the current working directory:

```
$ slc start myapp
```

To run an application remotely, use Process Manager; see Operating Node applications for more information.

To view the status of the application, use the `slc ctl` command, which by default displays the status of the locally-running StrongLoop PM:

```
$ slc ctl
Service ID: 1
Service Name: myapp
Environment variables:
  No environment variables defined
Instances:
    Version  Agent version  Cluster size
     4.1.1       1.5.1            4
Processes:
        ID      PID   WID  Listening Ports  Tracking objects?  CPU profiling?
    1.1.48554  48554   0
    1.1.48557  48557   1     0.0.0.0:3001
    1.1.48563  48563   2     0.0.0.0:3001
    1.1.48565  48565   3     0.0.0.0:3001
    1.1.48566  48566   4     0.0.0.0:3001
```

Where `myapp` is your application's service name (by default, the `name` property in `package.json`).

To see log output (including error messages and a stack trace), use this command:

```
$ slc ctl log-dump myapp
```

If PM cannot successfully run the application, you can make changes to the application code, and PM will automatically try to run it: you don't have to use `slc start` again.

Stop the application with:

```
$ slc ctl stop myapp
```

Debugging applications

See also:

- slc debug command reference
- How To Videos
- Debugging clustered apps with Node Inspector
- Node Inspector README
- Node Inspector wiki on GitHub

Overview

Use Node Inspector (invoked with the `slc debug` command) to debug Node applications. Node Inspector enables you to:

- Set breakpoints (and specify trigger conditions) in code, disable/enable all breakpoints.
- Step over, step in, step out, and resume code execution.
- Inspect scopes, variables, and object properties.
- Display the value of an expression in source code by hovering your mouse over it.
- Edit variables and object properties.
- Break on exceptions.

Prerequisite

Make sure you've installed StrongLoop, which installs the `slc` command-line tool.

Node Inspector currently works only in the Google Chrome and Opera browsers. If you are using a different browser, reopen the Node

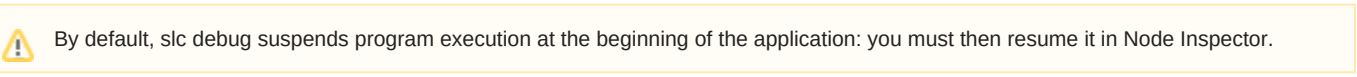
Inspector page in one of those browsers.

Starting the debugger

Run your application in debug mode like this:

```
$ slc debug
```

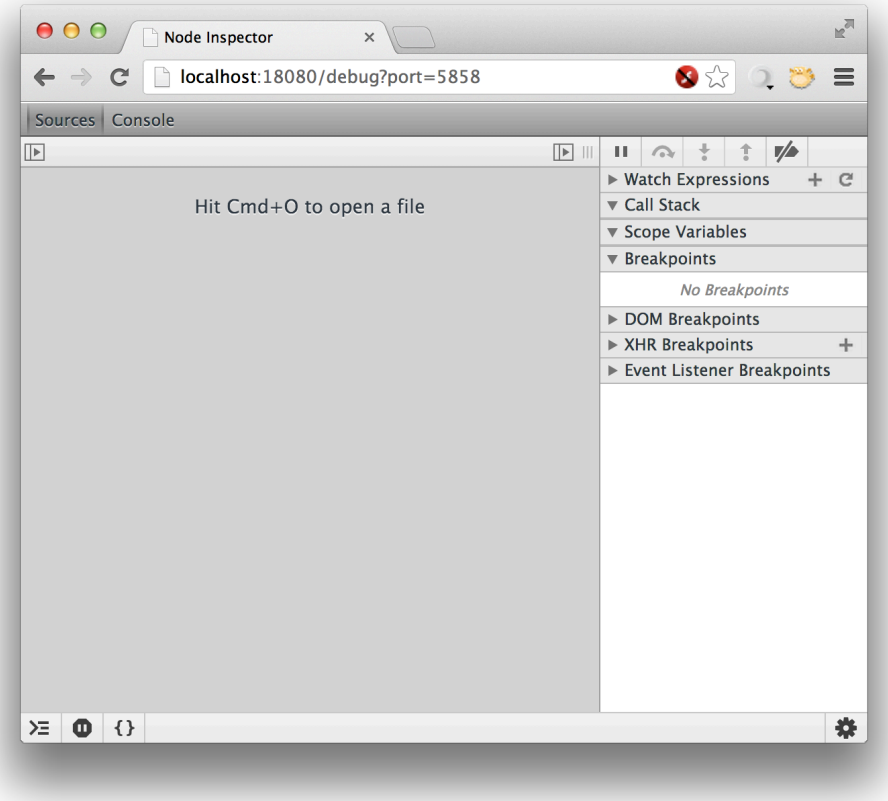
This command starts Node Inspector and opens the interface at http://localhost:8080/debug?port=5858 in your default browser.



By default, slc debug suspends program execution at the beginning of the application: you must then resume it in Node Inspector.

The `slc debug` command provides a number of options. See slc debug for more information. For additional configuration information, see the Node Inspector README.

If you don't want to run your application with the `slc debug` command, then you must use the `node --debug-brk` command to halt application execution on the first line. This enables you to set breakpoints, step through the code, and debug the application.



Debugging a running application

To enable debugging on an application that is already running:

1. Get the PID of the node process, for example:

```
$ pgrep -l node
2345 node your/node/server.js
```

2. Send the application the USR1 signal

```
$ kill -s USR1 2345
```

⚠ Windows does not support UNIX signals, so you must use a different technique. See the Node Inspector README for details.

Using Node Inspector

Overview

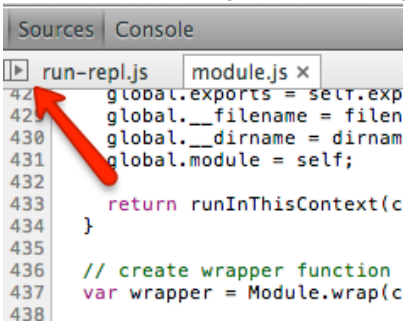
Node Inspector is based on Chrome Developer Tools and thus debugging a Node application with it is similar to debugging client JavaScript in Chrome. See the Chrome Developer Tools Guide for a walkthrough of other debugger features.

Node Inspector loads and parses files during a debug session and automatically adds them to the GUI. This is useful for `--debug-brk` and stepping through `require()` calls.

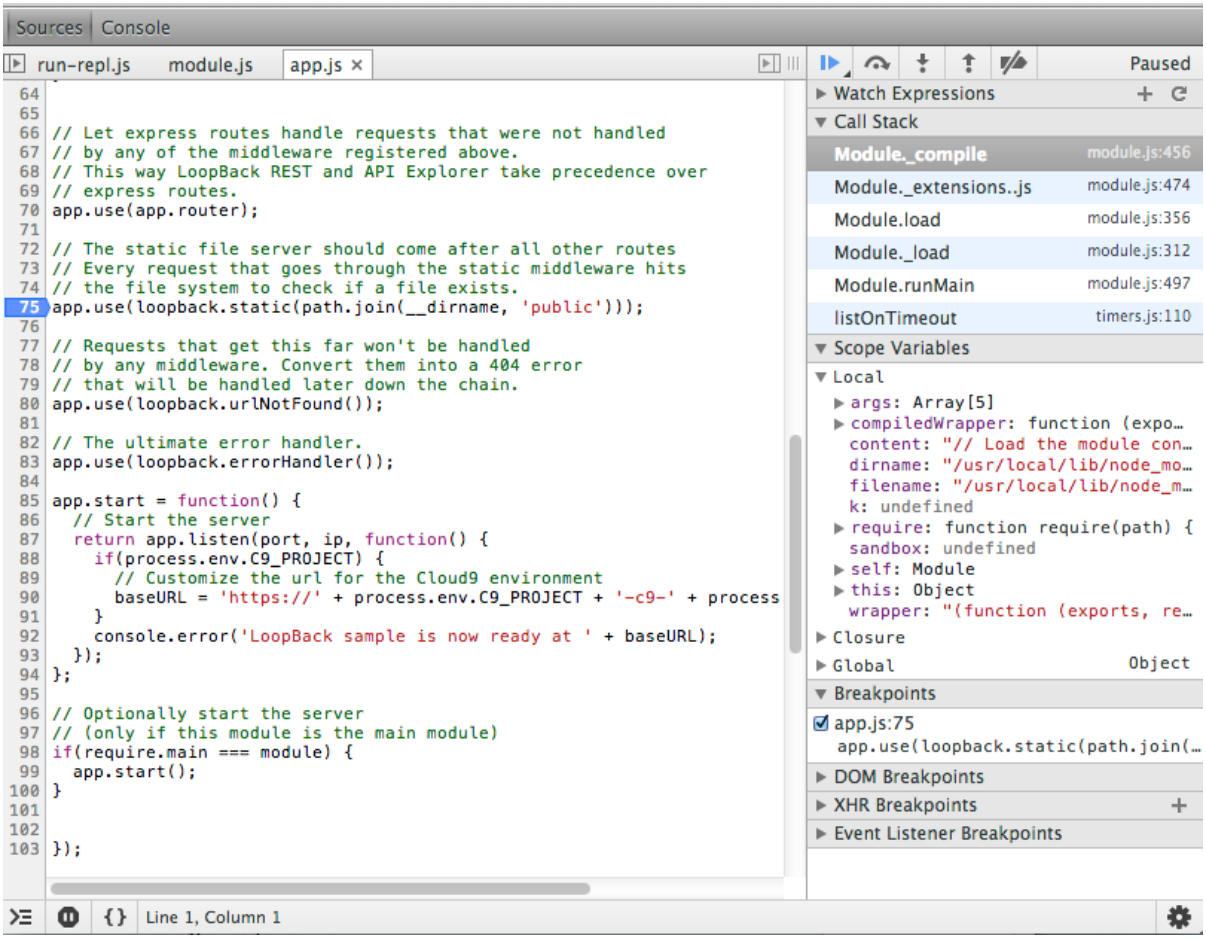
Setting a breakpoint

To set a breakpoint in application code:

1. Click on the "Show navigator" icon in the upper-left corner of the Node Inspector window to see a tree-list of all source files:



2. Find the desired JavaScript file and double-click to open it.
3. Click on the desired line number to set a breakpoint there.
   Node Inspector highlights the line number in blue.



Press F10 to step through Javascript statements. Press F8 to resume script execution.

Breakpoints and uncaught exceptions

Node Inspector restores breakpoints after you restart an application and remembers your breakpoints in the browser's local storage (HTML5). When you restart the debugger process, or start debugging the same application after several days, Node Inspector restores the breakpoints.

You can also break on uncaught exceptions. Node Inspector also provides integration with domains: exceptions handled by domain's error handler are still considered uncaught. This feature requires Node.js v0.11.3 or greater.

Setting breakpoints in files not yet loaded

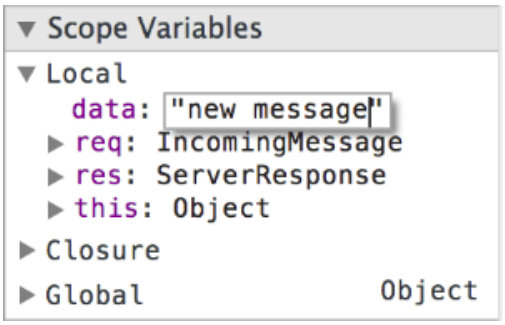
Node Inspector enables you to set breakpoints in files that are not yet loaded into the V8 runtime.

To do this, for example for Mocha unit tests:

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

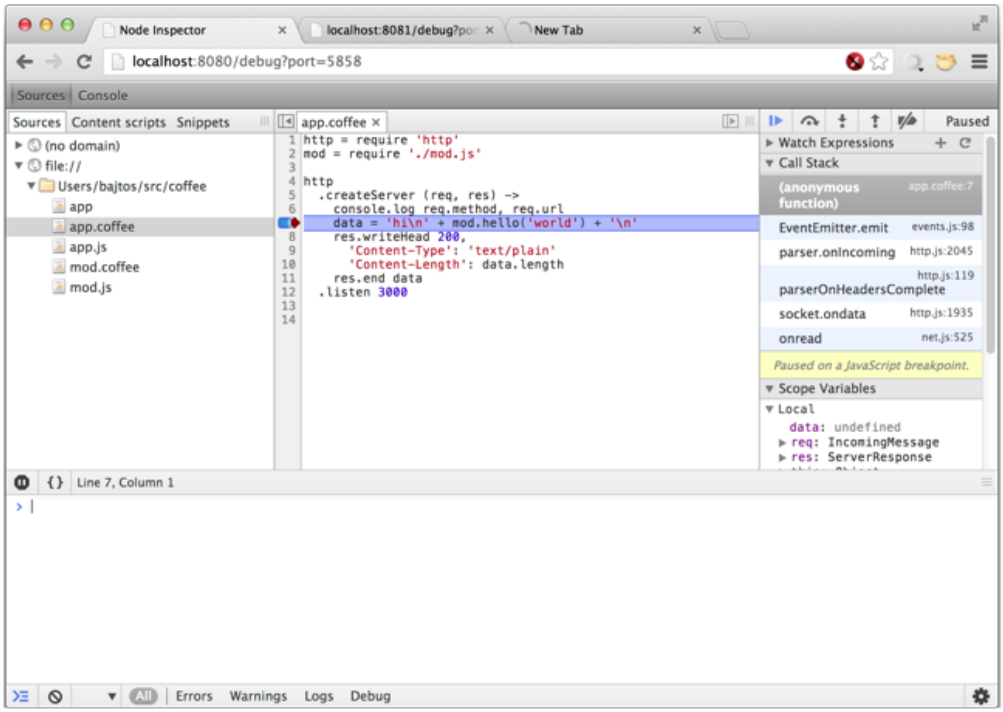
Display and edit variables

Hover a variable to display and edit its value, as illustrated below:



Support for source maps

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



Re-execute functions with "Restart Frame"

Right-click on a call frame (stack frame) in the right sidebar and select the ”Restart frame” command to re-execute the current function from the beginning.

Keyboard shortcuts

Keyboard shortcuts

Command	Mac	Windows / Linux
Next Panel	]	Ctrl-]
Previous Panel	[	Ctrl-[
Toggle Console	Esc	Esc
Focus Search Box	F	Ctrl-F

Find Next	G	Ctrl-G
Find Previous	G	Ctrl-Shift-G

Console keyboard shortcuts

Command	Mac	Windows / Linux
Next Suggestion	Tab	Tab
Previous Suggestion	Tab	Shift-Tab
Accept Suggestion	Right	Right
Previous Command / Line	Up	Up
Next Command / Line	Down	Down
Previous Command	P	
Next Command	N	
Clear History	K or L	Ctrl-L
Execute	Return	Enter

Debugger keyboard shortcuts

Command	Mac	Windows / Linux
Select Next Call Frame	.	Ctrl-.
Select Previous Call Frame	,	Ctrl-,
Continue	F8 or /	F8 or Ctrl-/
Step Over	F10 or ’	F10 or Ctrl-’
Step Into	F11 or ;	F11 or Ctrl-;
Step Out	F11 or ;	Shift-F11 or Ctrl-Shift-;
Edit Breakpoint	Click	Ctrl-Click

Profiling

Profiling means generating CPU profiles and heap memory snapshots for an application. These can help detect memory leaks and other run-time issues.

StrongLoop tools enable you to profile both local and remote Node applications. You can:

- Profile local or remote applications with Arc.
- Profile local applications with slc.

Using the StrongLoop profiler demo app

The StrongLoop profiler demo app is a simple LoopBack application that showcases CPU and heap profiling.

The demo app includes a load generator script that makes two POST requests:

- `POST /documents` - creates multiple instances of the document object and burns CPU. Use the CPU profiler to see the CPU usage.
- `POST /documents/content` - increases the Document count. Use the heap profiler to see the increased object count.

Install demo app

Install the profiler demo app from GitHub:

```
$ git clone https://github.com/strongloop/profiler-demo-app.git
$ cd profiler-demo-app
$ npm install
```

Perform monitoring

Run the app:

```
$ slc start
```

Then display the application status to ensure that the application is running:

```
$ slc ctl
```

You should see something like this:

```
Service ID: 1
Service Name: profiling-app
Environment variables:
  No environment variables defined
Instances:
    Version  Agent version  Cluster size
     4.1.1       1.5.1            4
Processes:
        ID      PID   WID  Listening Ports  Tracking objects?  CPU profiling?
    1.1.63264  63264   0
    1.1.63267  63267   1     0.0.0.0:3001
    1.1.63268  63268   2     0.0.0.0:3001
    1.1.63269  63269   3     0.0.0.0:3001
    1.1.63270  63270   4     0.0.0.0:3001
```

Now, run the load generator script:

```
$ node loadtest.js
```

You can collect CPU and heap profiling data using the `slc ctl` commands and Google Chrome tools.

For example, to profile CPU consumption for worker process number one (1):

```
$ slc ctl cpu-start 1
Profiler started, use cpu-stop to get profile
```

Then, after a few seconds, stop CPU profiling:

```
$ slc ctl cpu-stop 1
CPU profile written to `node.1.cpuprofile`, load into Chrome Dev Tools
```

To profile heap memory consumption:

```
$ slc ctl heap-snapshot 1
slc ctl heap-snapshot 1
Heap snapshot written to `node.1.heapsnapshot`, load into Chrome Dev Tools
```

See slc ctl for more details.

Profiling with Arc

StrongLoop Arc Profiler enables you to gather and view CPU profiling and heap memory snapshot data for Node applications running on remote servers as well as on the local system.

Prerequisites

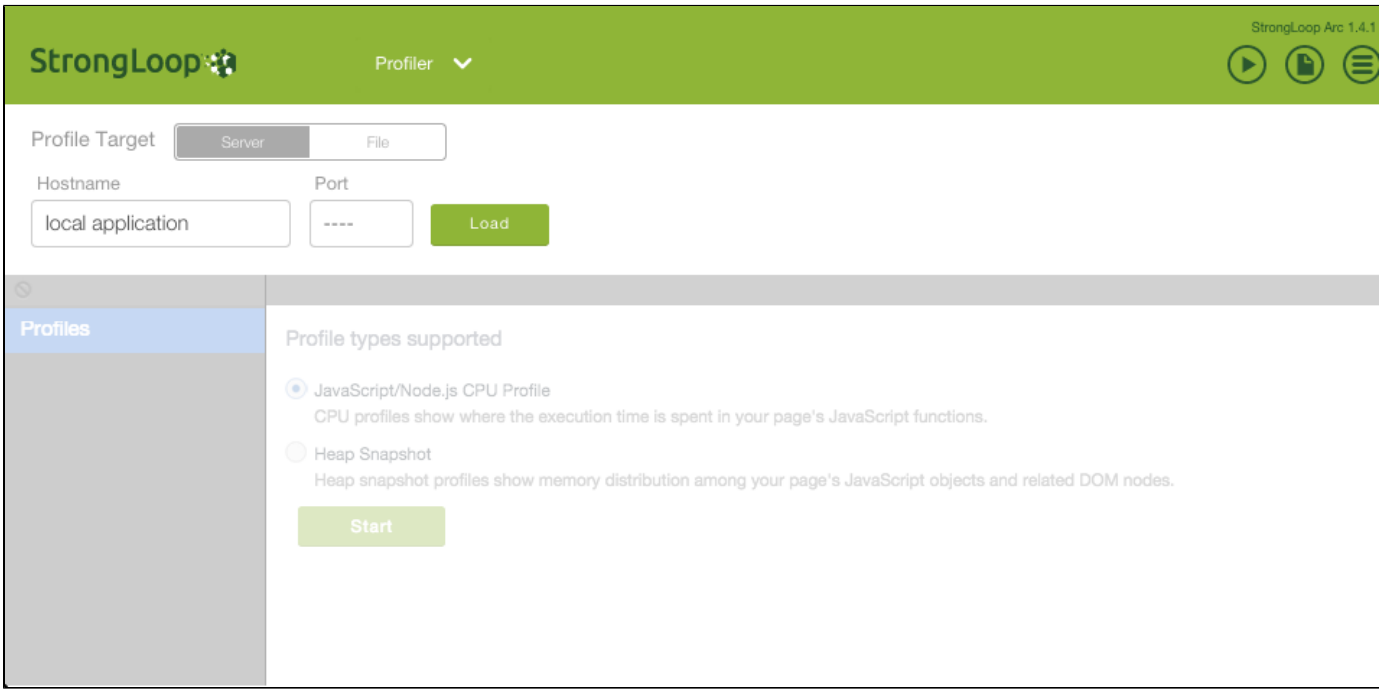
To use StrongLoop Profiler, you must:

- Have C++ compiler tools on the system running the application you wish to profile. See Installing compiler tools for details.
- Use Node version 0.10.*x* or later.

You cannot generate profiles for applications running on Windows 8 systems, due to a known issue in Node.

Loading the Profiler module

Click **Profiler** in module picker to display Arc Profiler:



Enter the following:

- **Hostname**: Host where StrongLoop PM is running.
- **Port**: Port number on which StrongLoop PM is listening.

Click **Load** to display all the processes for the application managed by that PM.

Now, you can create and view CPU profiles and heap snapshots; For more information, see:

- CPU profiling with Arc.
- Taking heap snapshots with Arc.

Viewing local profile files

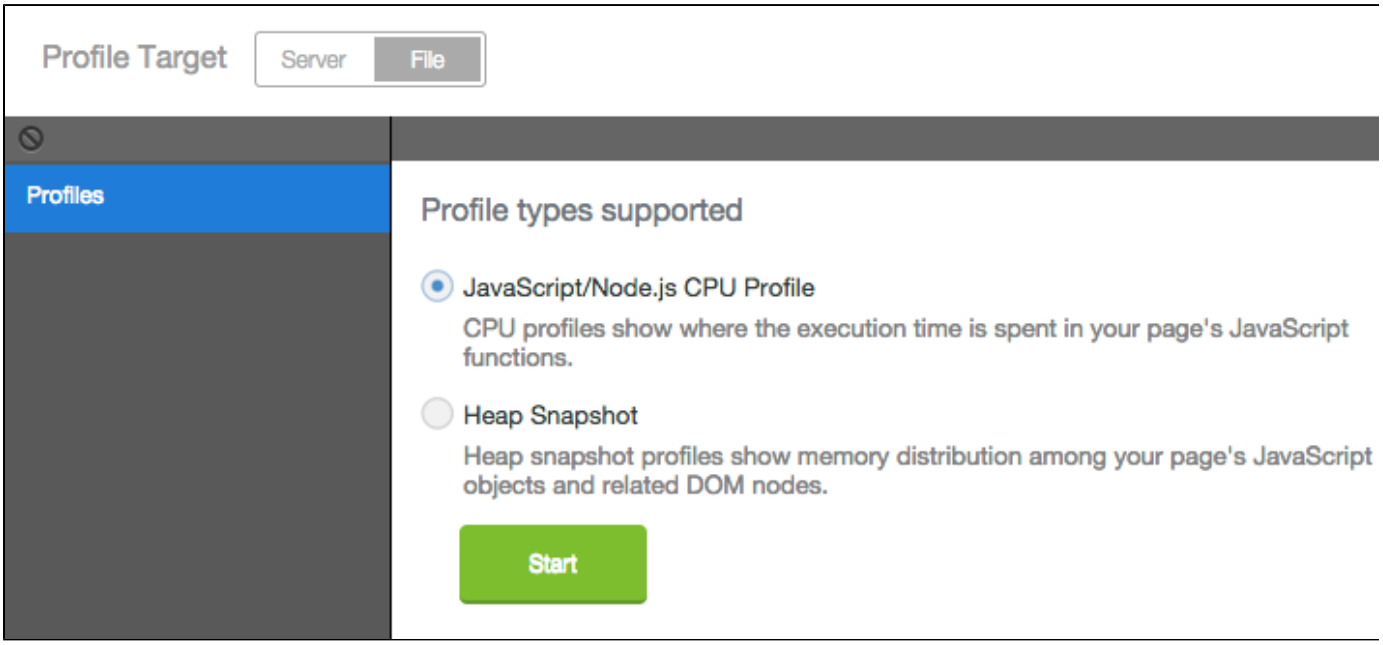
You can also generate CPU profiles and heap snapshots with StrongLoop Controller, the `slc` command-line tool. Then you can view the results in Arc.

For more information, see

- CPU profiling.
- Taking heap snapshots.

> Arc lists generated profiles on the left of the **Profiler** module. Click on a profile to view it.
>
> The profile files are saved on the system where PM is running in the `.strong-pm` subdirectory in the home directory of the user running PM (`~/.strong-pm`). The file names are:
>
> - CPU profiles: `profile.1.cpuprofile`, `profile.2.cpuprofile`, `profile.3.cpuprofile`, and so on.
> - Heap snapshots: `profile.1.heapsnapshot`, `profile.2.heapsnapshot`, `profile.3.heapsnapshot`, and so on.

Choose **File** to load CPU profiles and heap snapshots from the local file system:



Choose either **JavaScript/Node.js CPU Profile** or **Heap Snapshot**.

Click **Load**.

Clearing all loaded profiles

To clear all profiles click  .

CPU profiling with Arc

StrongLoop Arc Profiler enables you to gather and view CPU profiling data for Node applications.

- Generating CPU profiles
  - Local application
  - Server application
- Viewing CPU profile information

Click **Profiler** in module picker to display Arc Profiler.

Generating CPU profiles

Local application

By default, when you run the application you're working with, Arc will start a local instance of StrongLoop Process Manager, and display "local application" in the Hostname field, as shown above. Click **Load** to display the process IDs (PIDs) of the local application processes; by default the first PID will be selected. Then, you can follow the procedures below to generate and view profiles for that PID; or select another PID.

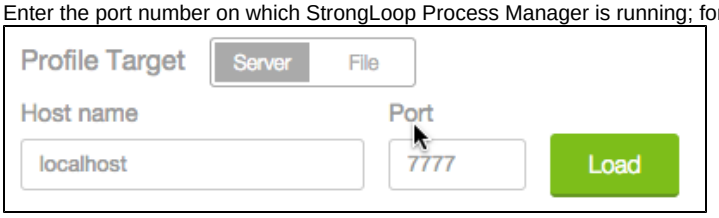
Server application

To profile an application running on a server (either a remote server, or localhost):

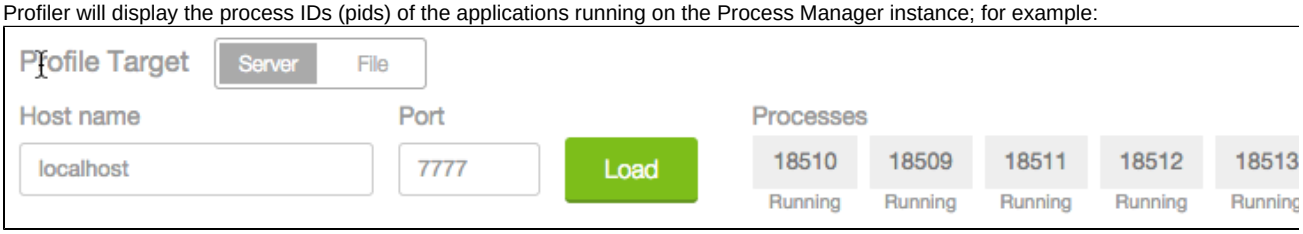
1. Start an instance of StrongLoop Process Manager on the system running the application; for example:

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

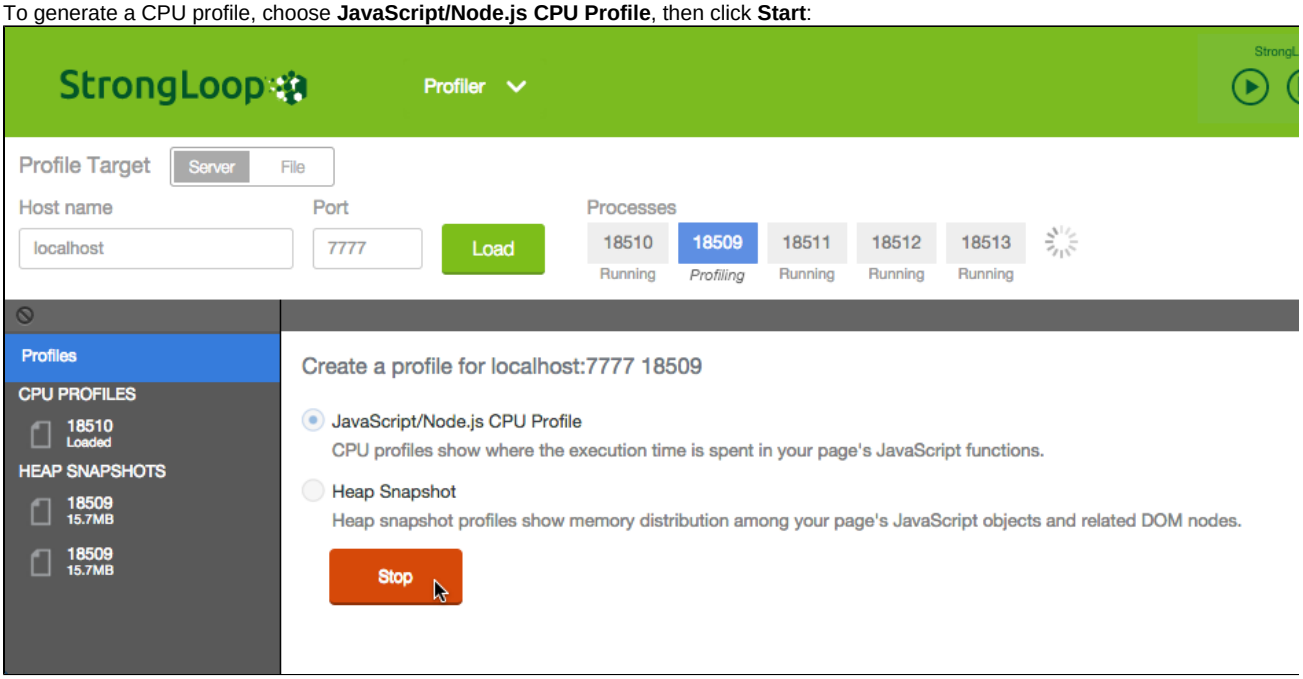
2. Deploy an application to the Process Manager.
3. In Arc Profiler, for **Profile Target** choose **Server**.
4. Enter the host name of the system where StrongLoop Process Manager is running. For example, if Process Manager is running on the same system as Arc, enter `localhost`.
5. Enter the port number on which StrongLoop Process Manager is running; for the above example, it would be 7777.



6. Click **Load**.
Profiler will display the process IDs (pids) of the applications running on the Process Manager instance; for example:



7. Click on the desired pid.
8. To generate a CPU profile, choose **JavaScript/Node.js CPU Profile**, then click **Start**:



Let the profiler run for the desired amount of time, typically 5 - 15 seconds.

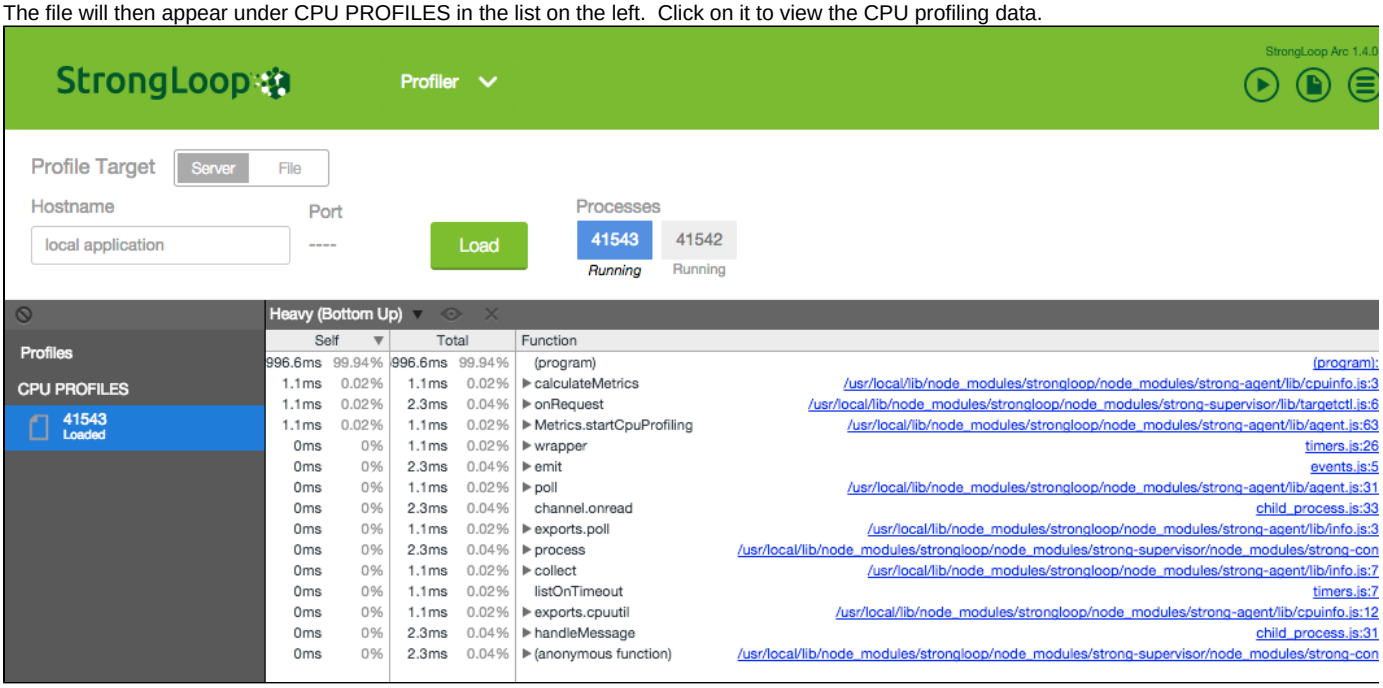
9. Click **Stop**.
   Arc will display the CPU profile in the **Profiles** list to the left. Click on it to view the CPU profile.

Viewing CPU profile information

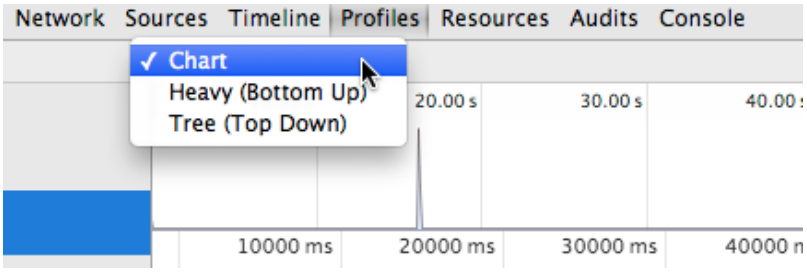
To view CPU profile information in Arc Profiler:

1. Click **Load**.
2. Choose the desired `.cpuprofile` file; for example, `node.8197.cpuprofile`.

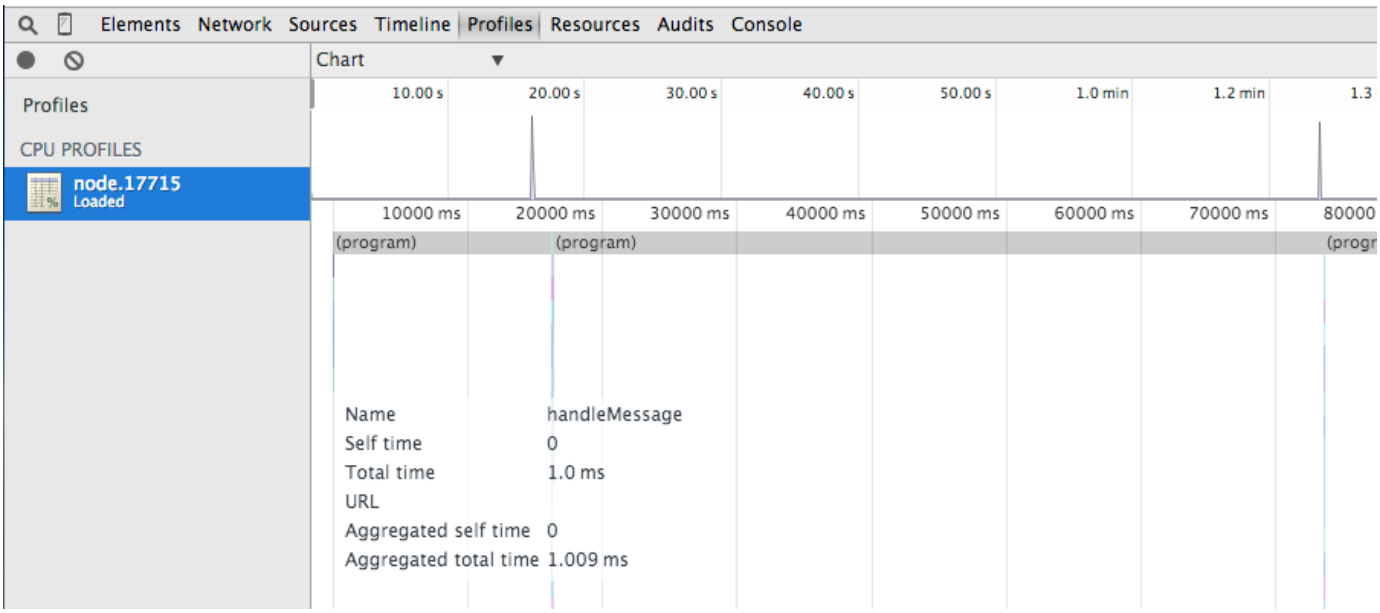
The file will then appear under CPU PROFILES in the list on the left. Click on it to view the CPU profiling data.



Click on the inverted triangle to choose the type of display:



Choose **Chart** to see a graph of CPU consumption:



Click on **Heavy (Bottom Up)** or **Tree (Top Down)** to display tabular data.

Click **Profiles** in the sidebar to display the list of profiles again.

Taking heap snapshots with Arc

StrongLoop Arc Profiler enables you to gather and view CPU profiling and heap memory snapshot data for Node applications.

Click **Profiler** in module picker to display Arc Profiler.

Generating heap snapshots

Local application

By default, when you run the application you're working with, Arc will start a local instance of StrongLoop Process Manager, and display "local application" in the Hostname field, as shown above. Click **Load** to display the process IDs (PIDs) of the local application processes; by default the first PID will be selected. Then, you can follow the procedures below to generate and view profiles for that PID; or select another PID.

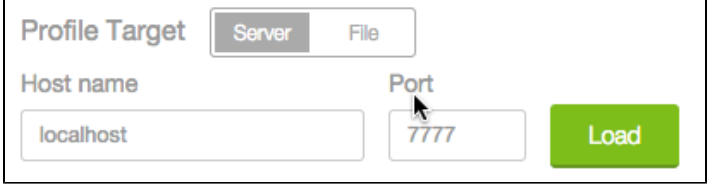
Server application

To profile an application running on a server (either a remote server, or localhost):

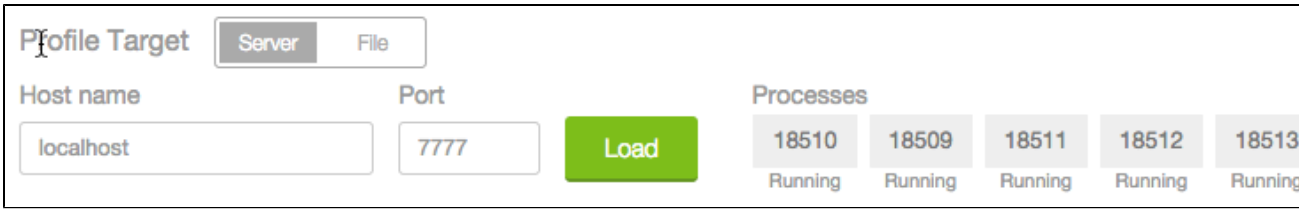
1. Start an instance of StrongLoop Process Manager on the system running the application; for example:

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

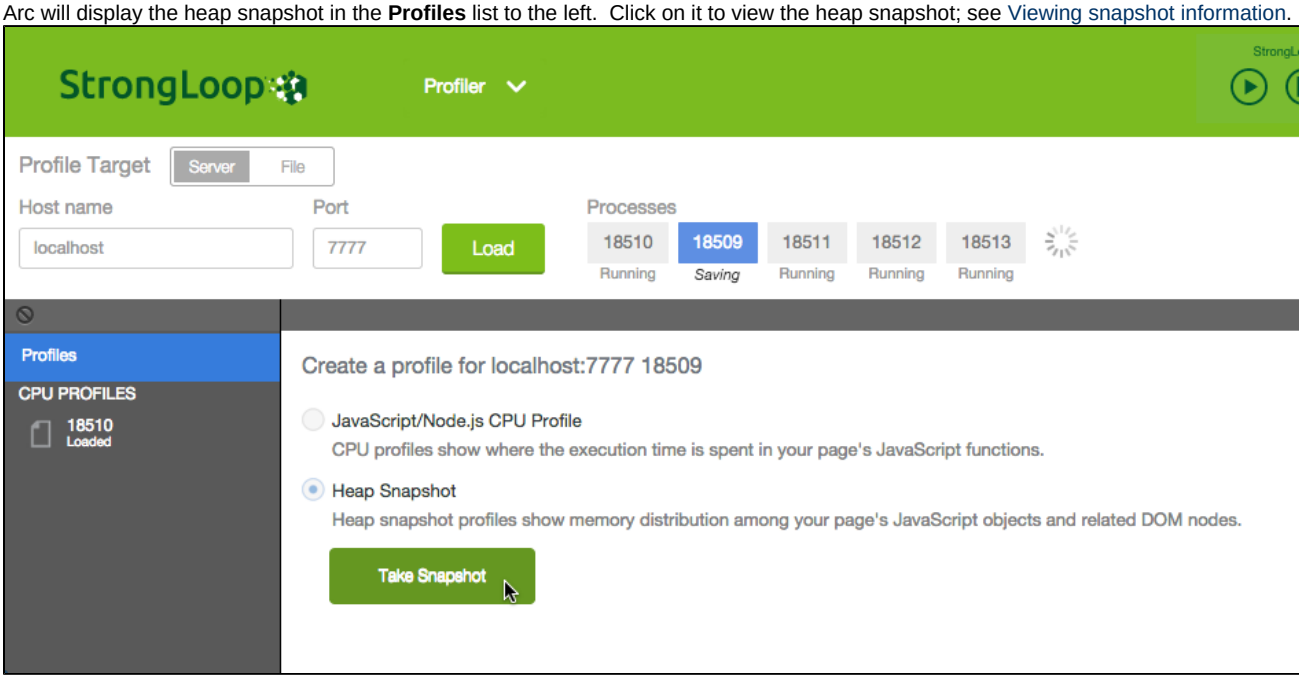
2. Deploy an application to the Process Manager.
3. In Arc Profiler, for **Profile Target** choose **Server**.
4. Enter the host name of the system where StrongLoop Process Manager is running. For example, if Process Manager is running on the same system as Arc, enter `localhost`.
5. Enter the port number on which StrongLoop Process Manager is running; for the above example, it would be 7777.



6. Click **Load**.
   Profiler will display the process IDs (pids) of the applications running on the Process Manager instance; for example:



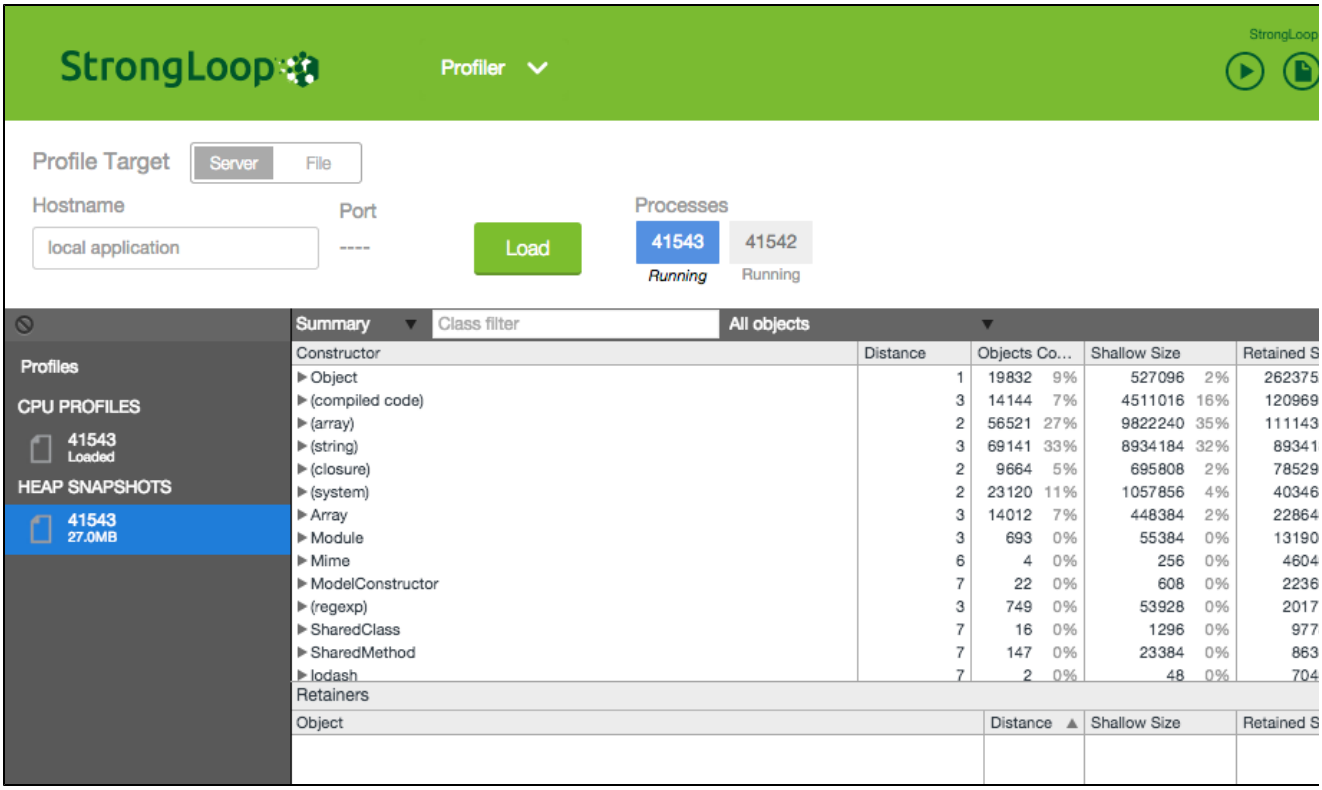
7. Click on the desired pid.
8. To generate a heap snapshot, choose **Heap Snapshot**, then click **Take Snapshot**.
   Profiler will display *Saving* beneath the selected pid. When it's done, it will display *Done*.
   Arc will display the heap snapshot in the **Profiles** list to the left. Click on it to view the heap snapshot; see Viewing snapshot information.



Viewing snapshot information

To view heap snapshot information in Arc Profiler:

1. Click **Load**.
2. Choose the desired heap snapshot from the list of profiles. Then, you will see the heap snapshot data:



Click on the inverted triangle to choose the type of display: **Summary**, **Comparison**, **Containment**, or **Statistics**.

For more information, see Taking heap snapshots and Profiling memory peformance (Chrome DevTools documentation).

Click **Profiles** in the sidebar to display the list of profiles again.

Smart profiling with Arc

Smart profiling is supported only for applications running on Linux x86_64 systems with:

- StrongLoop Arc version 1.6.6+
- StrongLoop Process Manager 5.0.1+

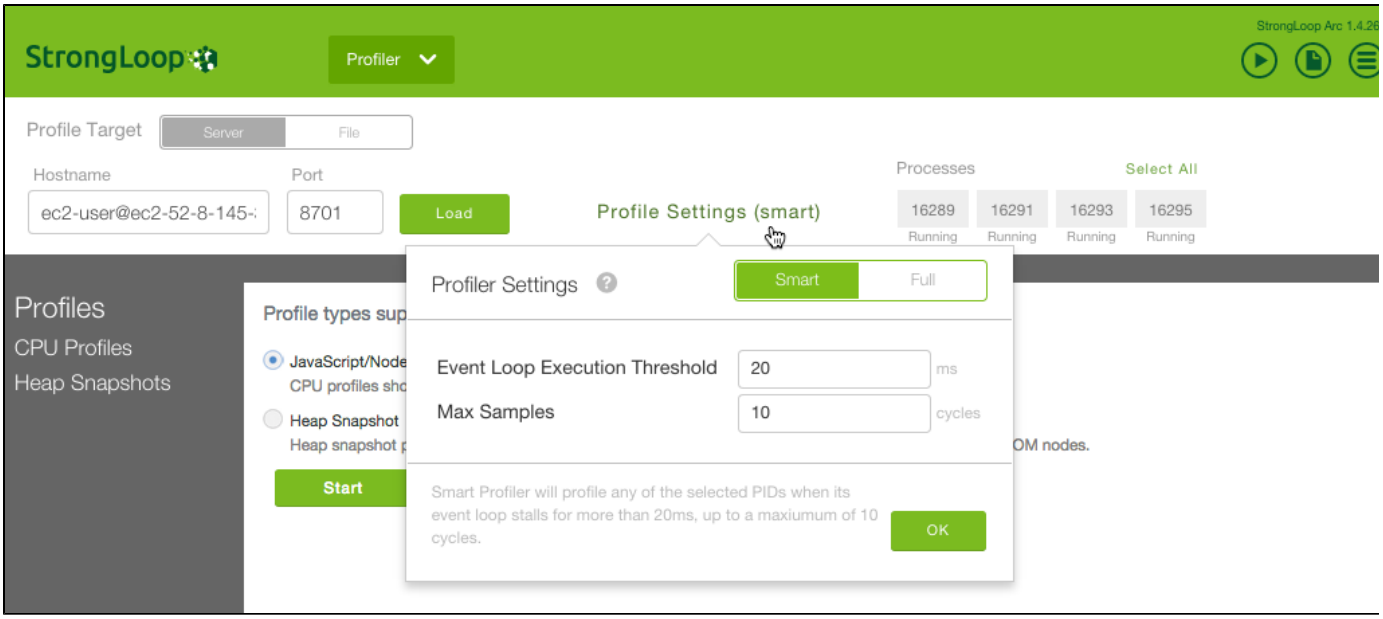
You don't get a free 30-day trial license for Smart Profililng, as you do for other features. To try it out, please email **slsales@us.ibm.com**.

Use Smart Profiling to start CPU profiling automatically when the Node event loop stalls.

For a sample application to demonstrate Smart Profiling in action, see https://github.com/strongloop/smartprofiling-example-app.

For a tutorial-style introduction to Smart Profiling, see Node.js Smart Profiling Using StrongLoop Arc.

Click **Profiler** in the Arc launchpad or module selector.



In Arc Profiler, follow these steps:

1. In the **Hostname** field, enter the name of a Linux host where StrongLoop Process Manager (PM) is running.
2. In the **Port** field, enter the port number where StrongLoop PM is listening.
3. Click **Load**.
   Arc will display the process IDs (PIDs) of the application running in that PM.
4. Select the PIDs for which you want to set smart profiling.
5. Click **Profile Settings (full)**. You'll see the **Profiler Settings** dialog:
6. Click **Smart** to enable smart profiling.
7. In the **Event Loop Execution Threshold** field, enter the number of milliseconds after which profiling will start. If the Node event loop stalls for longer than this, then smart profiling starts automatically.
8. In the **Max Samples** field, enter the maximum number of event loop cycles during which to profile.
9. Click **OK** to close the **Profiler Settings** dialog and save the settings.
10. Click the desired PID on which to do Smart Profiling; the PID box will turn blue.
11. Click **Start** to begin Smart Profiling.
    To end Smart Profiling, click **Stop**.

NOTE: StrongLoop Profiler will save CPU profiles after you set the Smart Profiling settings and click **Start**; and before you click **Stop**.

To disable smart profiling:

1. Click **Full**.
2. Click **OK**.

Arc lists generated profiles on the left of the **Profiler** module. Click on a profile to view it.

The profile files are saved on the system where PM is running in the `.strong-pm` subdirectory in the home directory of the user running PM (`~/.strong-pm`). The file names are:

- CPU profiles: `profile.1.cpuprofile`, `profile.2.cpuprofile`, `profile.3.cpuprofile`, and so on.
- Heap snapshots: `profile.1.heapsnapshot`, `profile.2.heapsnapshot`, `profile.3.heapsnapshot`, and so on.

Profiling with slc

You can generate CPU profiles and heap memory snapshots with `slc`.

It saves all profile files in the directory where you are running `slc`. The file names will be:

- `node.`*`id`*`.cpuprofile`, where *`id`* is either the worker ID (an integer such as 1 or 2) or the process ID, depending on which you use as the argument to the `slc ctl cpu-start` command.
- `node.`*`id`*`.heapsnapshot`, where *`id`* is the worker ID (an integer such as 1 or 2), depending on which you use as the argument to the `slc ctl heap-snapshot` command.

For details, see:

CPU profiling

To profile CPU consumption, you must:

- Use Node version 0.10.*x* or later.
- Install compiler tools on your system.

CPU profiling with slc

The CPU profiler is not designed to be kept running indefinitely; rather, turn it on for 5 to 30 seconds to get some quick insights into application performance.

To profile CPU consumption with slc, follow these steps:

1. Run your application:

```
$ cd <your-app>
$ slc start
```

   This will start a local instance of StrongLoop Process Manager and deploy the application to it.
2. Get process and worker IDs:

```
$ slc ctl
Service ID: 1
Service Name: myapp
Environment variables:
  No environment variables defined
Instances:
    Version  Agent version  Cluster size  Driver metadata
     4.3.1       1.6.3            4             N/A
Processes:
        ID      PID   WID  Listening Ports  Tracking objects?  CPU profiling?
Tracing?
    1.1.44417  44417   0
    1.1.44418  44418   1     0.0.0.0:3002
    1.1.44419  44419   2     0.0.0.0:3002
    1.1.44420  44420   3     0.0.0.0:3002
    1.1.44421  44421   4     0.0.0.0:3002
```

   This lists the process ID of StrongLoop Process Manager and the worker IDs and process IDs of all the worker processes. In the example above, there are four worker processes. Notice that the Process Manager itself is running in process 44417.
3. Start CPU profiling for a specific worker process; for example:

```
$ slc ctl cpu-start 1.1.44418
Profiler started, use cpu-stop to get profile
```

   Let CPU profiling run while your application is under typical or high load to get meaningful CPU profile data.

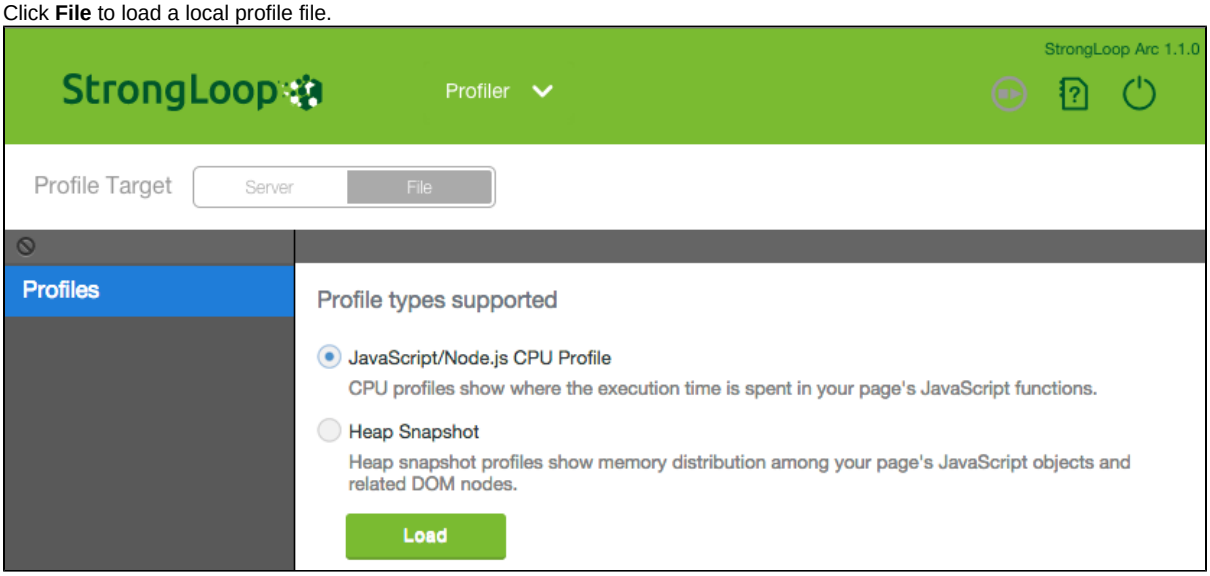
4. Stop CPU profiling with the `cpu-stop` command, for example:

```
$ slc ctl cpu-stop 1.1.44418
CPU profile written to `node.1.1.44418.cpuprofile`, load into Chrome Dev Tools
```

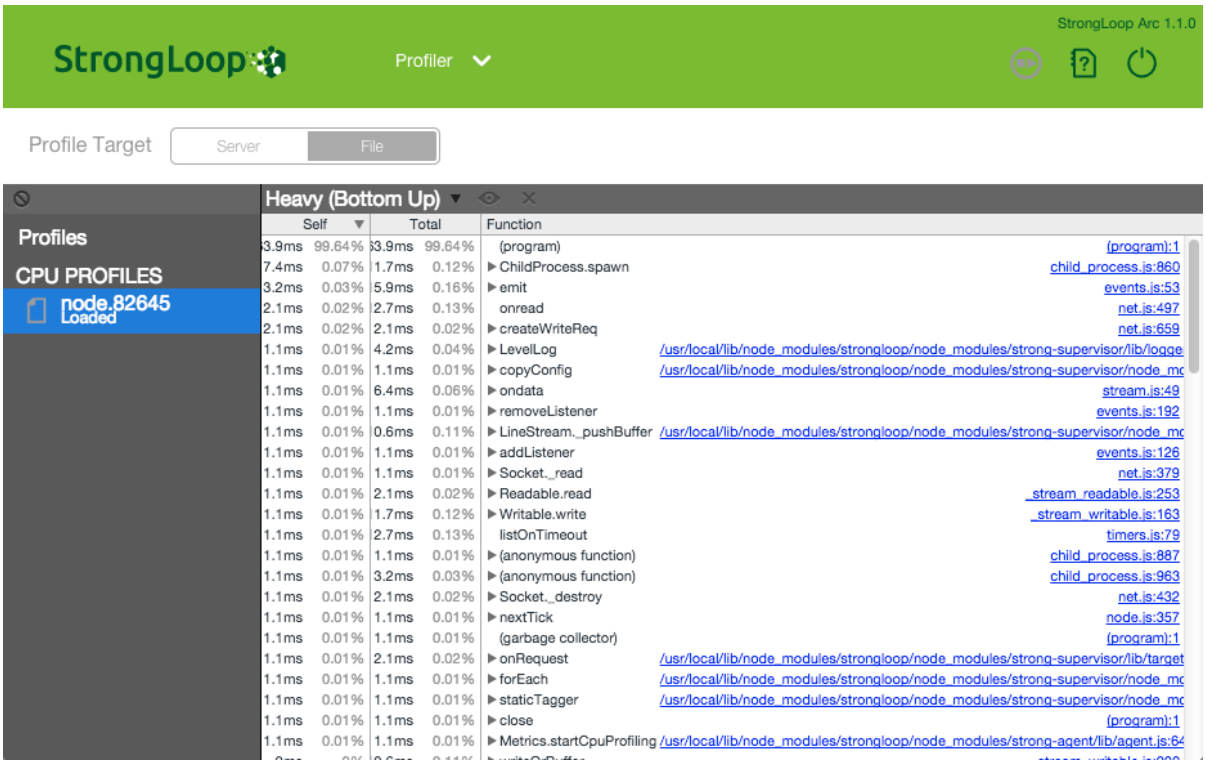
   You can optionally provide a filename after the process/worker ID. When you stop CPU profiling, `slc` saves the output to the current working directory to the file *`name`*`.cpuprofile`. If you don't provide the filename argument, by default it saves results to the file `node.<`*`workerID`*`>.cpuprofile`; in the above example, `node.1.1.44418.cpuprofile`.
5. Stop your application:

```
$ slc ctl stop 1
```

   See slc ctl for more information.
6. Start StrongLoop Arc to view the profile data.
7. In Arc, click **Profiler**.
8. Click **File** to load a local profile file.



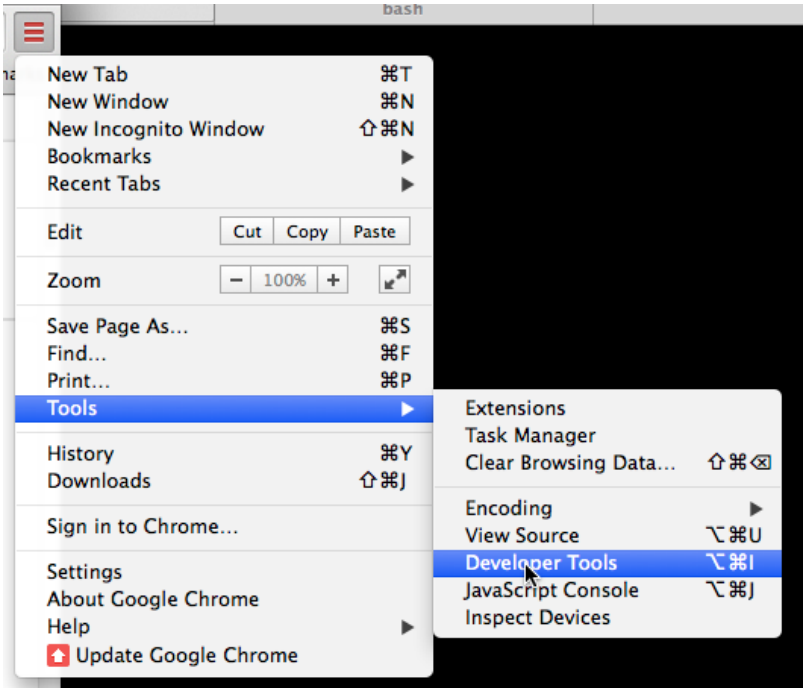
9. Click **Load**.
10. Choose the `.cpuprofile` file you generated.
11. The file will then be listed in in the Profiles list: click on it to view the CPU profile data, for example, as shown below.
    For information on viewing CPU profile data in Chrome DevTools, see Profiling JavaScript Performance.



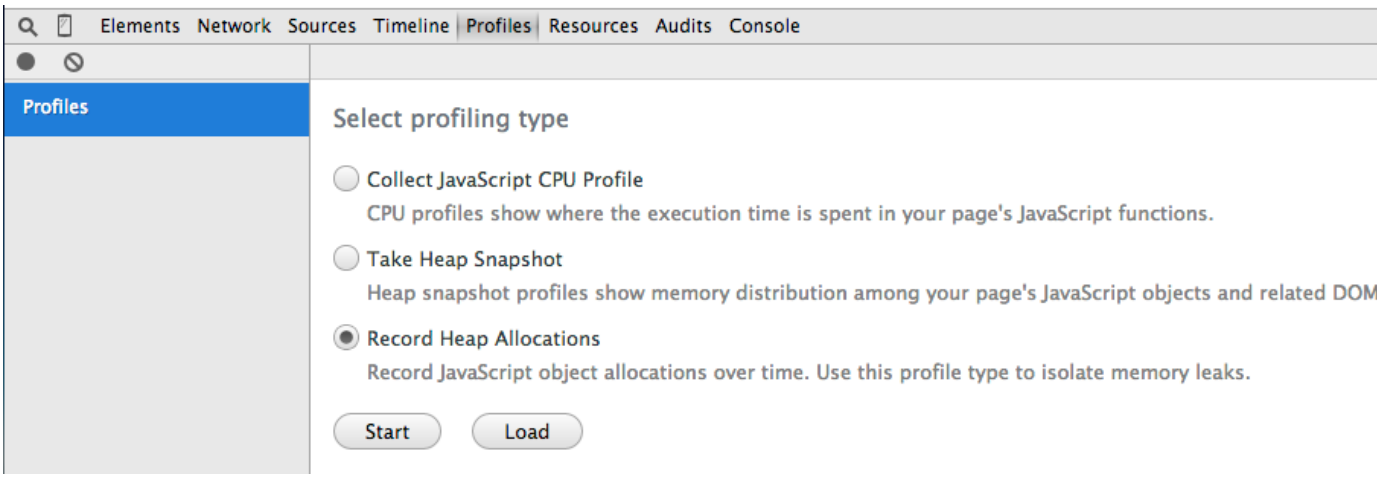
Viewing CPU profile data

> For more information on viewing CPU profiles in Chrome DevTools, see Profiling JavaScript Performance.

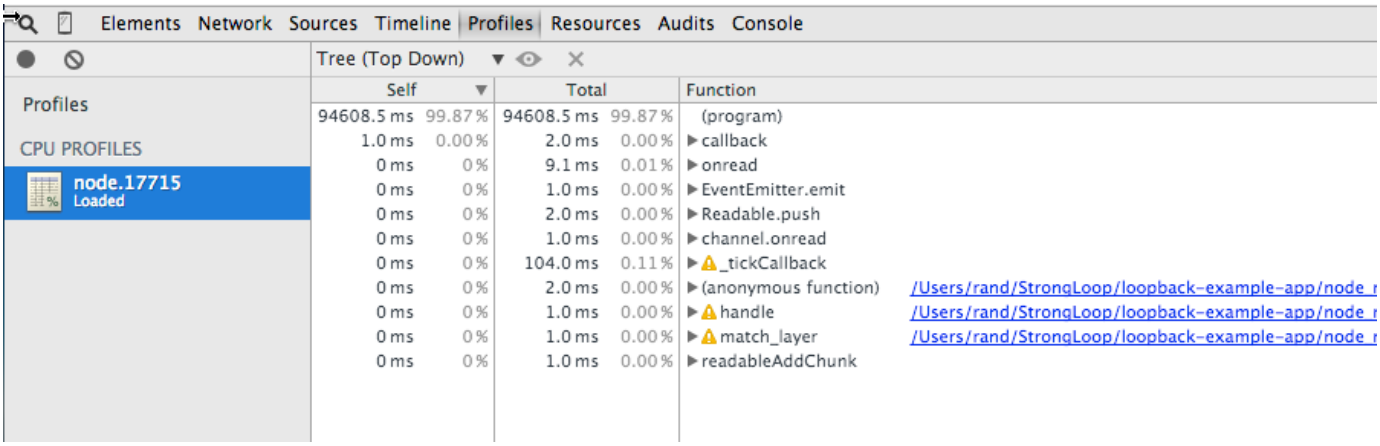
Now in your Chrome browser, click the icon in the upper right of the window and choose **Tools > Developer Tools** (on some systems, the menu item may be **More tools**):



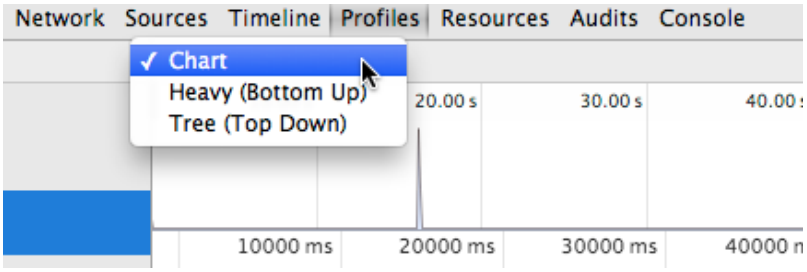
Chrome will display Developer Tools in the lower part of the window. Choose **Profiles** in the menu bar. You'll see the Profiles page:



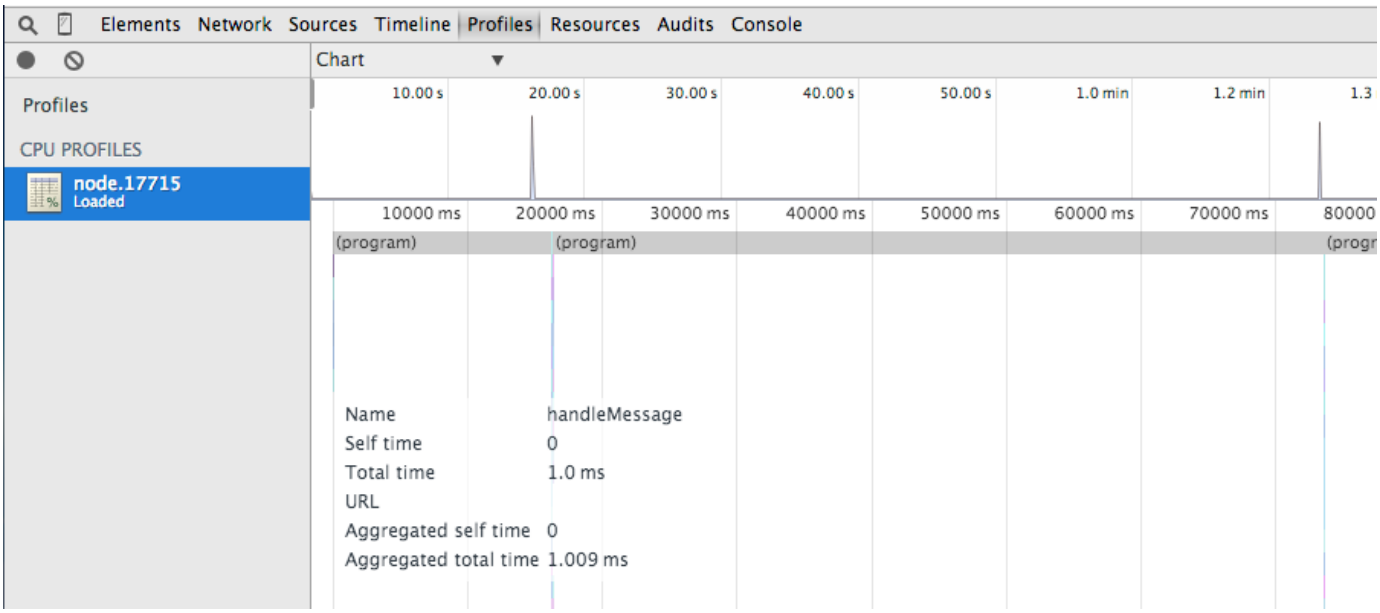
Click **Load** then choose the file you saved previously; in the example above, `node.8197.cpuprofile`. The file will then appear under **CPU PROFILES** in the list on the left. Click on it to view the CPU profiling data.



Click on the inverted triangle to choose the type of display:



Choose **Chart** to see a graph of CPU consumption:



A known issue in Node versions before 0.10.26 and Node 0.11.14 causes CPU consumption to be reported higher than it actually is.

Taking heap snapshots

To get heap snapshot information, you must install compiler tools on your system.

Overview

Take heap snapshots of your application to help track down memory leaks.

First run your application:

```
$ slc start
```

Then display the application logs to the console to ensure that the application is running:

```
$ slc ctl
```

You should see something like this:

```
Service ID: 1
Service Name: express-example-app
Environment variables:
  No environment variables defined
Instances:
    Version  Agent version  Cluster size
     4.1.0       1.5.1            4
Processes:
        ID      PID   WID  Listening Ports  Tracking objects?  CPU profiling?
    1.1.50320  50320   0
    1.1.50321  50321   1     0.0.0.0:3001
    1.1.50322  50322   2     0.0.0.0:3001
    1.1.50323  50323   3     0.0.0.0:3001
    1.1.50324  50324   4     0.0.0.0:3001
```

This command will display the PIDs and worker IDs for all worker processes.

Then, take a heap snapshot with the slc ctl command, providing as the argument the worker or process ID of interest; for example:

```
$ slc ctl heap-snapshot 1.1.1
```

In general, the argument 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).

The tool writes the heap snapshot data to a `.heapshapshot` file in the current working directory that you can open in Chrome Developer Tools. In this example, the heap snapshot data will be in a file named `node.1.1.1.heapsnapshot`.

For more information on `slc` options, see slc ctl.

Strategies for taking heap snapshots

Not all memory leaks are created equal. Some are slow leaks that can take hours or even days to build. Some happen on under-utilized code paths. Some are constant. So when should you take a snapshot? The answer: it depends.

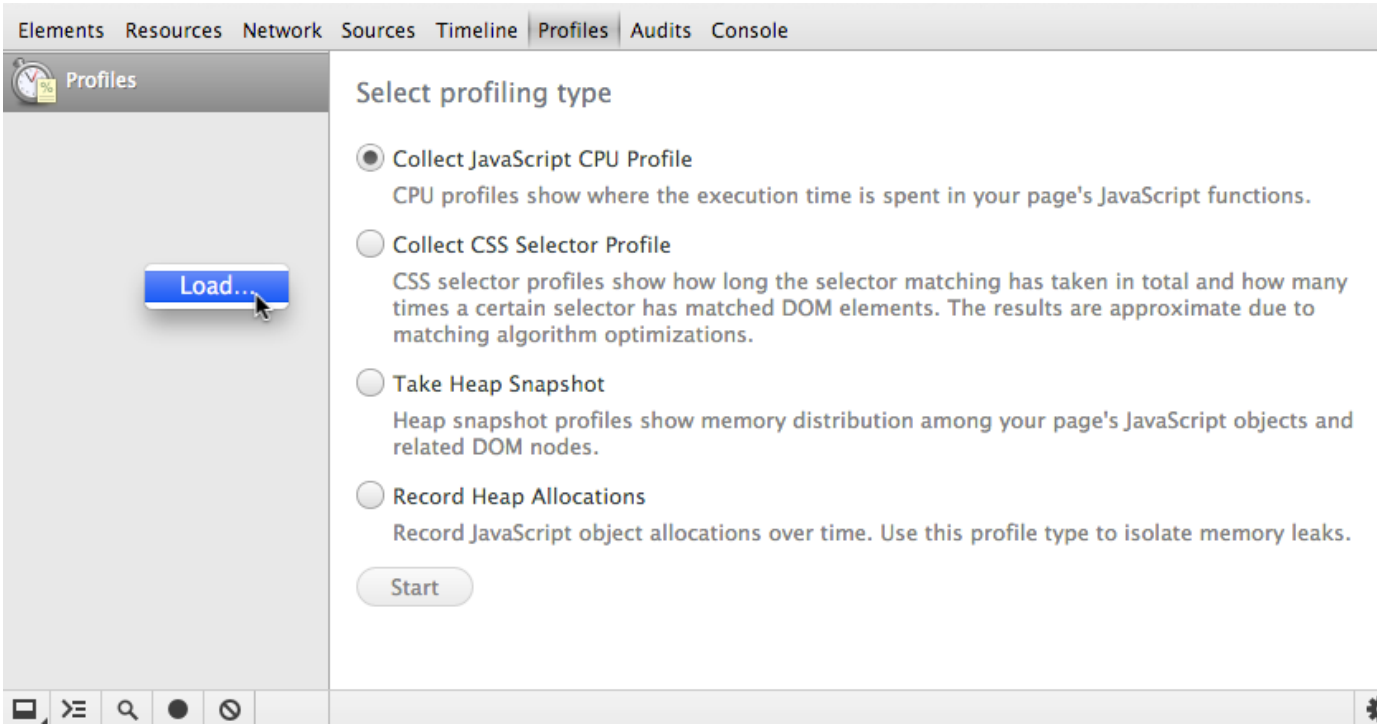
It's good to have a baseline snapshot to compare memory usage. You may want to generate a snapshot after your application is up and running for a few minutes. As you notice the memory use increase, repeat the process to gather more data for analysis later.

When you take a heap snapshot, V8 performs a GC prior to taking the snapshot to give you an accurate picture of what is sticking around in your process. So don't be surprised if you notice your memory usage drop after taking a snapshot.

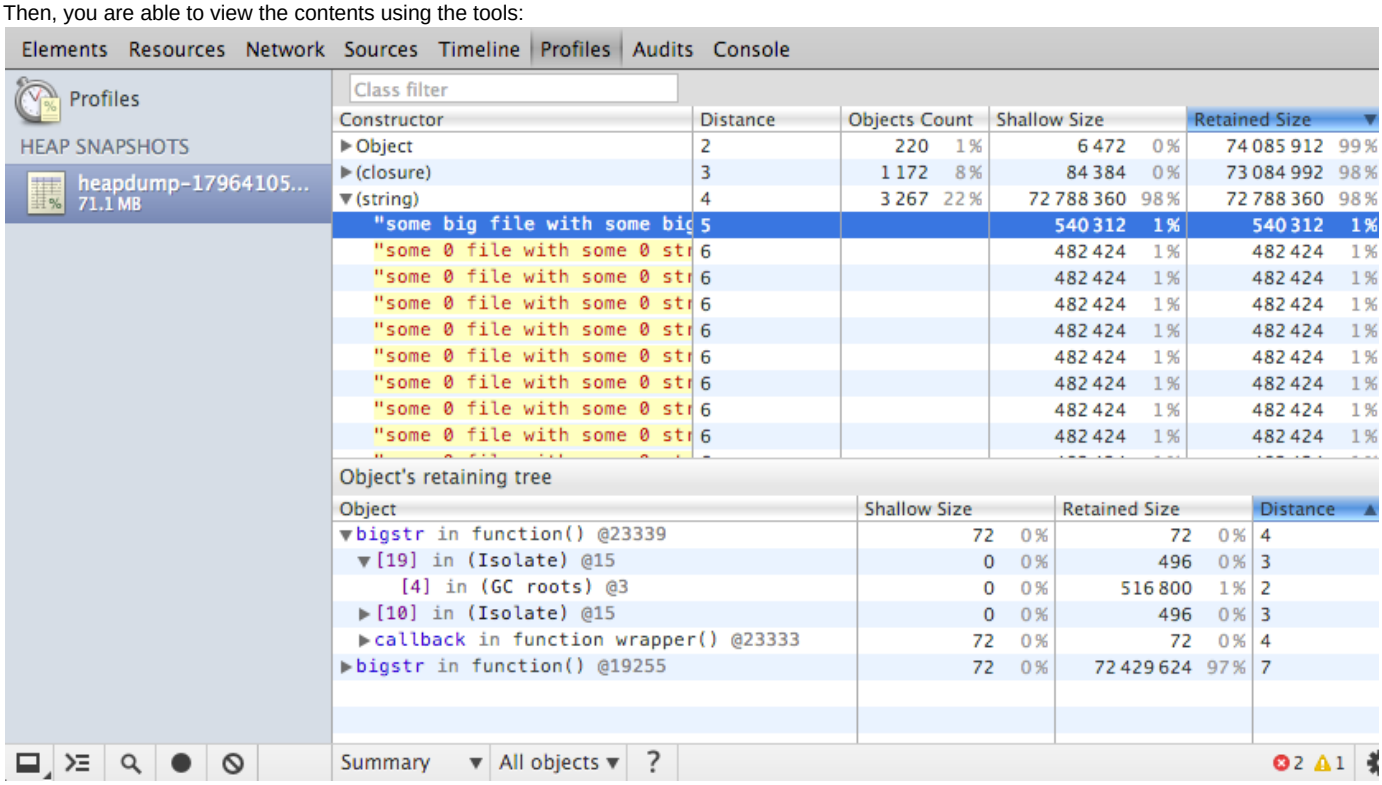
Loading heap snapshots into Chrome Developer Tools

After taking a heap snapshot, load the `.heapsnapshot` file into Chrome Developer Tools (CDT):

1. In the Chrome menu at the upper right of the window, choose **Tools > Developer Tools**.
2. In the developer tools frame, choose **Profiles** tab.
3. Right-click and choose **Load...** as shown below.
4. Choose the `.heapsnapsnot` file.



Then, you are able to view the contents using the tools:



For more information, see Profiling memory performance (Chrome DevTools documenation)

Analyzing snapshot data

If you haven't worked with heap snapshot analysis in CDT, see the following articles for an introduction:

- Memory Analysis 101
- Taming The Unicorn: Easing JavaScript Memory Profiling In Chrome DevTools

The CDT Comparison view is often helpful. For more detail on the purpose for the Comparison view and the other views, see Chrome Dev Toos heap profiling documentation.

Taking heap snapshots programmatically

Rather than using the slc commands, you can use the heapdump module directly and programmatically take snapshots in certain circumstances. You just have to modify your application code slightly.

For example, you might want to take a snapshot at an interval that makes sense for the memory growth observed. For example, this code writes a snapshot to the current working directory every half hour:

```
var heapdump = require('heapdump');
...
setInterval(function () {
  heapdump.writeSnapshot();
}, 6000 * 30);
```

Taking a snapshot is not free. It will consume all CPU resources until the snapshot is written. The larger the heap, the longer it takes. On UNIX, snapshots are written in a forked process asynchronously, but Windows will block.

You could also watch the memory use growth programmatically and generate snapshots:

```
var heapdump = require('heapdump');
var nextMBThreshold = 0;
...
setInterval(function () {
  var memMB = process.memoryUsage().rss / 1048576;
  if (memMB > nextMBThreshold) {
    heapdump.writeSnapshot();
    nextMBThreshold += 100;
  }
}, 6000 * 2);
```

Explanation of the above code:

- Line 2: Next MB threshold to be breached
- Line 5: Calculate RSS size in MB
- Line 6: Whenever memory use exceeds the `nextMBThreshold`, write a snapshot and increment the threshold by 100 MB.
- Line 10: Run this check every couple minutes

Smart profiling with slc

> ⚠ **Smart profiling is supported only for applications running on Linux x86_64 systems** with StrongLoop Process Manager 5.0.1+.
>
> You don't get a free 30-day trial license for Smart Profililng, as you do for other features. To try it out, please email **sales@strongloop.com**.

Use Smart Profiling to start CPU profiling automatically when the Node event loop stalls.

Simply add a *timeout* argument to the slc ctl command to specify the threshold; if the event loop stalls for longer than that time, then CPU profiling starts. CPU profiling stops when the event loop resumes execution.

For example, for an application with service ID one (1) running locally:

```
$ slc ctl cpu-start 1.1.76901 12
```

This command will start CPU profiling for worker process ID 76901 if the event loop on that process stalls for more than 12ms (12 milliseconds). CPU profiling stops when then event loop resumes execution, that is, when it "becomes un-stuck".

Here's an example of smart CPU profiling for an app with service ID one (1) running on a remote host:

```
$ slc ctl -C http://my.remote.host cpu-start 1.1.76901 12
```

For a sample application to demonstrate Smart Profiling in action, see https://github.com/strongloop/smartprofiling-example-app.

Using Process Manager

StrongLoop Process Manager does not run on Windows systems. However, you can deploy an application from a Windows system to a Linux system.

See also:

- Setting up a production host
- Securing Process Manager
- Command references:
  slc pm
  slc pm-install

StrongLoop Process Manager (StrongLoop PM) is a production process manager for Node.js applications that enables you to:

- Build, package, and deploy your Node application to a local or remote system.
- View CPU profiles and heap snapshots to optimize performance and diagnose memory leaks.
- Keep processes and clusters alive forever.
- View performance metrics on your application.
- Easily scale to multiple hosts with an integrated Nginx load balancer.

Running StrongLoop PM

You can run StrongLoop PM:

- As a transient process during development and testing; see below.
- As an operating system service. For production use, always run it as a service to ensure it restarts if the system restarts or the process stops for some other reason.

Running StrongLoop PM as a transient process

Use the `slc pm` command to run Process Manager as a transient process, that is, a regular (non-service) application; for example:

```
$ slc pm
```

This starts StrongLoop Process Manager as a transient process, listening on the default port 8701. Functionally, it is the same as a service, *exce pt* the process will not automatically restart if the system restarts.

Running StrongLoop PM as a transient process is appropriate during initial development and testing.

For production use, install StrongLoop PM as a service. For more information, see Setting up a production host.

Connecting to Process Manager from Arc

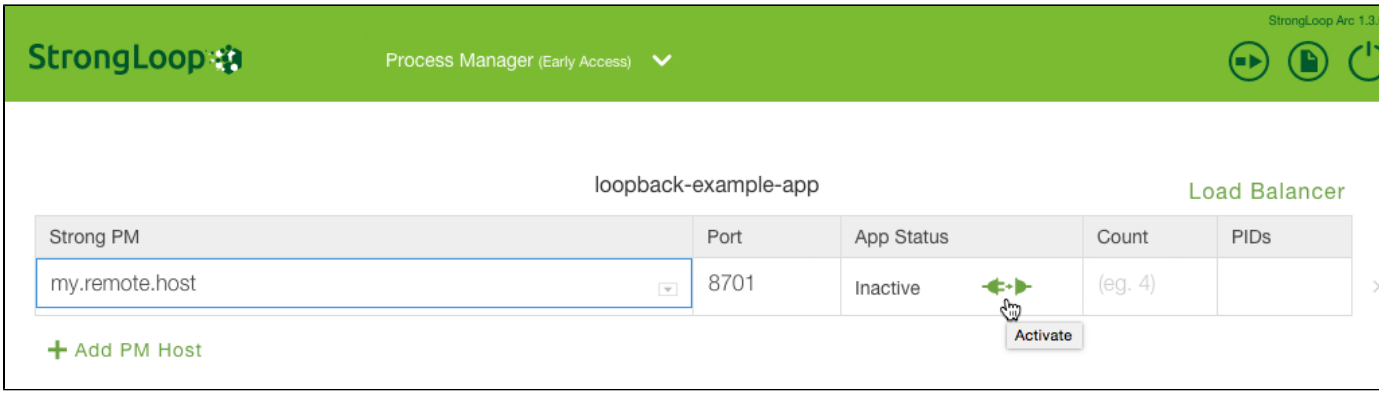
Currently, Arc can connect to only to a Process Manager hosting an application with service ID of one (1).

If you have multiple applications deployed to a PM, or to change the service ID of a deployed app, use the `slc ctl` command.

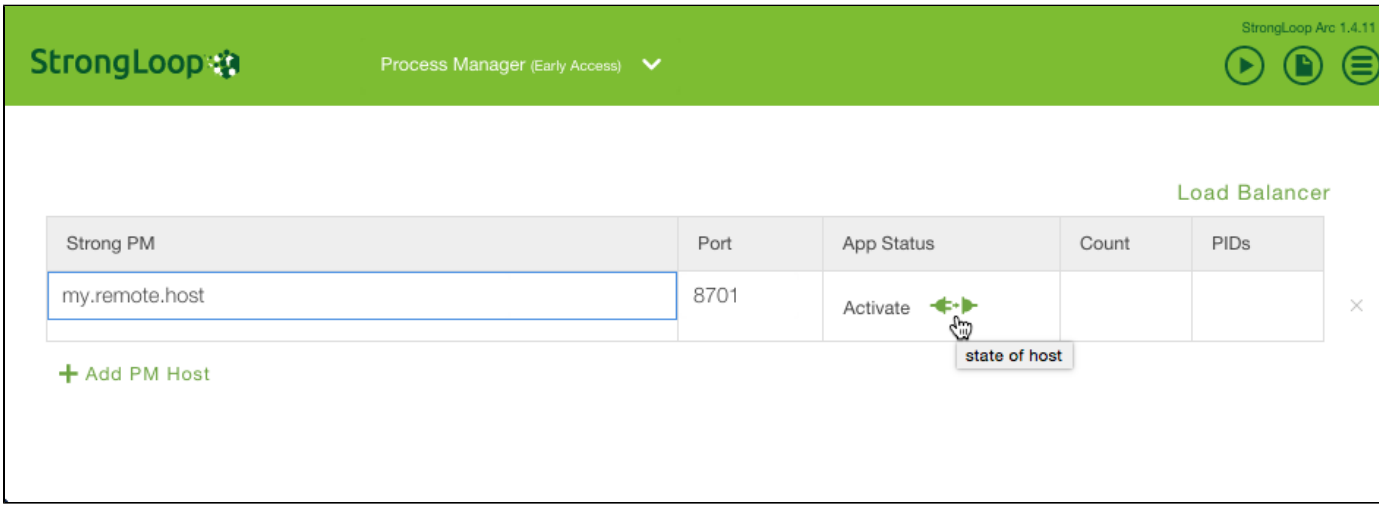
Once you've started StrongLoop Process Manager (StrongLoop PM) on your host system, you can deploy applications to it and control it from StrongLoop Arc. Then you can connect to the Process Manager from Arc:

1. Choose **Process Manager** in the Arc module selector.
2. Enter the host name where StrongLoop PM is running in the **Strong PM** field.

   Make sure you enter *only* the host name. Don't include "http://".
   For example, to connect to a Process Manager running on the local system, just enter "localhost", not "http://localhost".
3. Enter the port number where StrongLoop PM is listening in the **Port** field (by default 8701).
4. Click the Activate icon, as shown below.



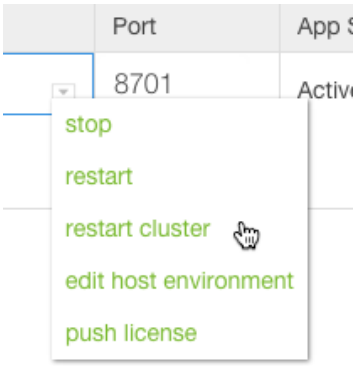
Under App Status, the icon will change from **Inactive** to **Active**.



Click **Add PM Host** to connect to additional Process Managers running on other hosts.

Controlling Process Manager from Arc

Once you're connected, you can control StrongLoop PM by clicking on the inverted triangle on the right side of the Strong PM field to display the menu of actions you can perform:



Click:

- **Stop** to stop the selected Process Manager or **restart** to restart it. Initially, you must start PM from the command-line, but once you connect to it from Arc, you can start and restart it via the menu.
- **Restart cluster** to restart the application clusters managed by that PM; see Controlling clusters with Arc for more information.
- **Edit host environment** to set or change environment variables for that PM; see Setting and viewing environment variablesfor more information.
- **Push license** to set licenses for that PM; see Managing your licenses for more information.

Environment variables

Several environment variables affect StrongLoop Process Manager:

- 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.
- STRONGLOOP_METRICS - set to a back-end URL (such as `statsd:`) for metrics collection. See Monitoring app metrics for more information.

Relevant environment variables on the client system:

- STRONGLOOP_PM - URL of Process Manager to which to connect.
- SSH_USER - username when connecting with `http+ssh`.
- SSH_KEY - path to the SSH private key to use when connecting with `http+ssh`.
- SSH_PORT - port to use with `http+ssh`.

For more information, see Setting and viewing environment variables.

Setting up a production host

See also:

- Using Process Manager
- Securing Process Manager
- Command references:
  - slc pm
  - slc pm-install

Prerequisites

The information here assumes that your production host is a Linux system.

Make sure you have installed Node on the production host. See https://nodejs.org/ for information on installing Node

To monitor performance metrics (CPU and heap consumption, event loop metrics, and so on), the system must have compiler tools, which are standard on most Linux systems. For more information, see Installing compiler tools.

To generate and view metrics, you must have:

- A standard set of C++ compiler tools on the system running the application. See Installing compiler tools for more information.
- A valid StrongLoop license key on the system running the application.
  You automatically get a free trial license valid for 30-days from when you first run Arc that applies to apps running locally within Arc. See Managing your licenses for more information.

Install using npm and run PM service

Installing StrongLoop PM as a service is not compatible with nvm.

Follow these steps on each production system:

1. Install StrongLoop PM with npm.
2. Install and run the StrongLoop PM service.

Installing just the `strong-pm` package rather than the full `strongloop` package is preferable on a production system, because it contains *only* the StrongLoop Process Manager and its runtime dependencies, not all the development-time packages, such as LoopBack, the StrongLoop development tools, `slc` and Arc, and so on, that you need on a development system.

When you install `strong-pm` package, you must use the `sl-pm-install` and `sl-pm` commands. Where you've installed the full `strongloop` package, you can also use `slc pm` and `slc pm-install`, which are equivalent.

Install npm package

On each production host, install the standalone StrongLoop Process Manager module (strong-pm), rather than the version packaged with the full StrongLoop installation:

```
$ npm install -g strong-pm
```

If you encounter issues such as file permission errors, consult Installation troubleshooting.

Install and run the service

On each host, install Process Manager as an operating system service, then run the service. The specific command depends on the operating system.

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, 12.10, 13.04, 13.10, 14.04, or 14.10. 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.
- On systems that support systemd instead (for example RHEL 7, Ubuntu 15.04 /15.10, Fedora, or OpenSUSE), use the `--systemd` option.

By default, the StrongLoop Process Manager service listens on port 8701. To make it listen on a different port, use the `--port` (`-p`) option. See slc pm-install for details.

When installing, you can use command options to:

- Set up HTTP authentication; see Set up HTTP authentication below.
- Enable monitoring of application metrics; see Enable monitoring below.

Ubuntu

On Ubuntu 12.04, 12.10, 13.04, 13.10, 14.04, or 14.10, use the `sl-pm-install` command with no options to install Process Manager as an Upstart 1.4 service:

```
$ sudo sl-pm-install
```

Then run the service with:

```
$ sudo /sbin/initctl start strong-pm
```

If an application fails to deploy, or you encounter any other problems, see the log file `/var/log/upstart/strong-pm.log` to diagnose the issue.

RHEL 7+, Ubuntu 15.04 or 15.10

On RHEL 7+/CentOS, Ubuntu 15.04 - .10, and other distributions that support systemd, use this command to install Process Manager as a systemd service:

```
$ sudo sl-pm-install --systemd
```

Then start the service with:

```
$ sudo /usr/bin/systemctl start strong-pm
```

On RHEL, by default `/usr/local/bin` is not on the path so if you encounter problems running `sl-pm-install`, use this command:

```
$ sudo env "PATH=$PATH" sl-pm-install
```

If an application fails to deploy, or you encounter any other problems, see the log file `/var/log/upstart/strong-pm.log` to diagnose the issue.

RHEL Linux 5 and 6, Ubuntu 10.04-.10, 11.04-.10

For RHEL 5 and 6 and other distributions that don't support Upstart 1.4, use this command to install Process Manager as an Upstart 0.6 service:

```
$ sudo sl-pm-install --upstart=0.6
```

Then run the service with:

```
$ sudo /sbin/initctl start strong-pm
```

Set up HTTP authentication

To secure StrongLoop Process Manager with HTTP authentication, use the `--http-auth <credentials>` option to the `sl-pm-install` command; for example for username "admin" and password "foobar":

```
$ sudo sl-pm-install --http-auth admin:foobar
```

This will enable HTTP basic authentication, and require the specified credentials for every request sent to the StrongLoop PM REST API.

For more information, see Securing Process Manager.

Enable monitoring

To enable monitoring of application metrics, add the `--metrics` option:

```
$ sudo sl-pm-install --metrics <url>
```

Where `<url>` specifies the metrics collection URL (StatsD, Graphite, Splunk, log file, or syslog). See Monitoring with slc for more information.

Enable Dockerizing apps

⚠ **Prerequisite**: You must have already installed and run Docker. For more information, see Docker documentation.

To install StrongLoop PM so that it automatically converts deployed applications into Docker images and installs them in the Docker Engine running on your deployment system, use this command:

```
$ sudo sl-pm-install --driver docker
```

For more information, see Deploying apps to Docker.

Set environment variables

To set one or more environment variables when you install StrongLoop PM as a service, use the `--set-env` option for example:

```
$ sudo sl-pm-install -set-env NODE_ENV=production
```

Install and run using Docker

⚠ **Prerequisite**: You must have already installed and run Docker. For more information, see Docker documentation.

If you want to set up HTTP authentication or enable monitoring application metrics, you must install with a slightly different command. See the relevant sections below.

Run the script below to download the image, start a StrongLoop PM container, and create an Upstart or systemd config file. The app will use port 3000; StrongLoop PM will use port 8701 and will automatically restart if your server reboots.

```
$ curl -sL https://strong-pm.io/docker.sh | sudo /bin/sh
```

For more information on Docker and Docker Hub, see https://www.docker.com.

Set up HTTP authentication

To set up HTTP authentication with Docker, set the STRONGLOOP_PM_HTTP_AUTH environment variable to your username and password when you install:

```
$ curl -sL https://strong-pm.io/docker.sh | sudo bash -s
STRONGLOOP_PM_HTTP_AUTH=myuser:mypass
```

For more information, see Securing Process Manager.

Enable monitoring

To enable monitoring of application metrics, set the STRONGLOOP_METRICS environment variable:

```
$ curl -sL http://strong-pm.io/docker.sh | sudo bash -s STRONGLOOP_METRICS=<url>
```

Where `<url>` specifies the metrics collection URL (StatsD, Graphite, Splunk, log file, or syslog). See Monitoring with slc for more information.

Check Process Manager status

To check to see the status of the Process Manager service, use the status command: the exact syntax depends on the type of service (Upstart or Systemd).

For Upstart systems, use this command:

```
$ sudo /sbin/initctl status strong-pm
```

You'll see output like this:

```
strong-pm start/running, process 1057
```

For Systemd systems, use this command:

```
$ sudo /usr/bin/systemctl status strong-pm
```

You'll see output like this:

```
strong-pm.service - StrongLoop Process Manager
   Loaded: loaded (/etc/systemd/system/strong-pm.service; enabled)
   Active: active (running) since Mon 2015-03-09 20:51:33 UTC; 4 min 13s ago
 Main PID: 1628 (node)
```

Logging

With Upstart 0.6, logs go to syslog so that syslog can handle log rotation, which means you have to configure syslog to redirect log entries if you want them rotated.

With Upstart 1.4, log rotation and direction is handled by Upstart itself and logs are saved to `/var/log/upstart/strong-pm.log`.

Everything run under systemd is logged to `journald`, which is accessible through the `journalctl` command. See the journalctl man page for more information about `journalctl`. For convenience, StrongLoop systemd service definitions *also* log to syslog.

Upgrade Process Manager

To upgrade Process Manager, first reinstall from npm with:

```
$ npm install -g strong-pm
```

Then reinstall the Process Manager service:

```
$ sudo sl-pm-install
```

Then reload the configuration.

For Upstart, use these commands:

```
$ sudo /sbin/initctl reload-configuration
$ sudo /sbin/initctl restart strong-pm
```

For Systemd, use these commands:

```
$ sudo /usr/bin/systemctl daemon-reload
$ sudo /usr/bin/systemctl restart strong-pm
```

Setting up a load balancer

Using an Nginx load balancer is optional. StrongLoop integrates with Nginx and provides an Nginx Controller via CLI and Arc. You can use your own load balancer or none at all, but in that case you are responsible for routing traffic to your hosts appropriately.

See Configuring Nginx load balancer for more information.

Uninstalling Process Manager service

How you uninstall Process Manager depends on how it was installed:

- Upstart service
- Systemd service
- Docker

Upstart service

If Process Manager is installed as an Upstart service (for example, on Ubuntu 12.04+ or RHEL 5 and 6), run these commands to uninstall it:

```
$ sudo /sbin/initctl stop strong-pm
$ sudo rm /etc/init/strong-pm.conf
$ sudo /sbin/initctl reload-configuration
$ sudo rm -rf /var/lib/strong-pm
```

If you installed with Docker, then use these commands:

```
$ sudo /sbin/initctl stop strong-pm-container
$ sudo rm /etc/init/strong-pm-container.conf
$ sudo /sbin/initctl reload-configuration
```

Systemd service

If Process Manager is installed as a systemd service, (with the `--systemd` option to `sl-pm-install`) run these commands to uninstall it:

```
$ sudo /usr/bin/systemctl stop strong-pm.service
$ sudo /usr/bin/systemctl disable strong-pm.service
$ sudo rm /etc/systemd/system/strong-pm.service
$ sudo /usr/bin/systemctl daemon-reload
$ sudo rm -rf /var/lib/strong-pm
```

Docker

If you installed with Docker, then use these commands:

```
$ sudo /usr/bin/systemctl stop strong-pm-container.service
$ sudo /usr/bin/systemctl disable strong-pm-container.service
$ sudo rm /etc/systemd/system/strong-pm-container.service
$ sudo /usr/bin/systemctl daemon-reload
```

Securing Process Manager

See also:

- Using Process Manager
- Setting up a production host
- Command references: slc pm and slc pm-install

Overview

In general, you should set up secure access to a remote instance of StrongLoop Process Manager so your production host remains secure.

The general process is:

1. Set up key-based authentication for the remote server.
2. Use `http+ssh` as the protocol instead of `http` in URL arguments to `slc` commands, for example:

```
$ slc deploy http+ssh://my.host.com:8701/default
$ slc ctl -C http://my.host.com:8701
```

Using HTTP authentication

To use HTTP authentication, you must set it up when you install the StrongLoop PM service. For more information, see Setting up a production host.

Once you've set up HTTP authentication for StrongLoop PM, you can set authentication credentials directly in the URL argument to the `slc ctl` command's `-C` option; for example:

```
$ slc ctl -C http://username:password@my.host.com:7654
```

Using SSH tunneling

To connect using SSH and tunnel the HTTP requests over that connection, use the `http+ssh` protocol in the URL, for example:

```
$ slc deploy http+ssh://my-server:8701/default
```

The SSH username defaults to your current user. Override the username default with the SSH_USER environment variable.

The SSH username defaults to your current user, authentication defaults to using your current SSH agent, and port defaults to 22. To override the default:

- Username, set the SSH_USER environment variable.
- Authentication, set the SSH_KEY environment variable to the path of the existing private key to use.
- SSH port, set the SSH_PORT environment variable.

Requirements

You need set up key-based authentication for the remote server. Also, you must use an SSH agent to make your keys available. Instead of setting up an agent, you can specify the path to the private key in the SSH_KEY environment variable.

If your local username isn't the right one, use the SSH_USER environment variable to specify the username.

For example:

```
$ export SSH_USER=ubuntu
$ export SSH_KEY=~/.ssh/ec2.rsa
$ slc deploy http+ssh://ec2-host:8701/default
```

Enhancing security

To enhance security, Do all of the following:

- Block direct access to port 8701 so that it can only be accessed from the server itself.
- Use `--http-auth` when you install the StrongLoop PM service; For more information, see Setting up a production host.
- Use `--control http+ssh://user:pass@remotehost:8701/`
  - Authenticates via SSH (username: `$SSH_USER` or `$LOGNAME` from environment, key from`$SSH_AUTH_SOCK` or `$SSH_KEY` from environment).
  - Authenticates via HTTP (user and pass from URL, matching username/password set during install).

Using the SSH tunnel allows us to connect to the PM's port via its localhost, which bypasses the firewall restriction that blocks direct access.

Setting and viewing environment variables

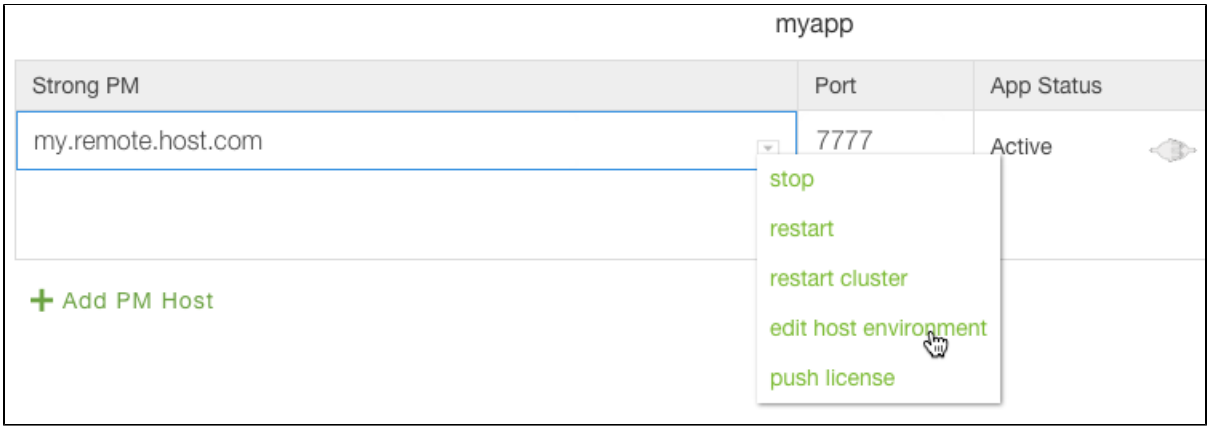
The `slc ctl` command provides sub-commands to set and view the values of variables in the StrongLoop Process Manager (PM) environment.

- Setting and viewing environment variables with Arc
- Setting and viewing environment variables with slc
- Environment variables

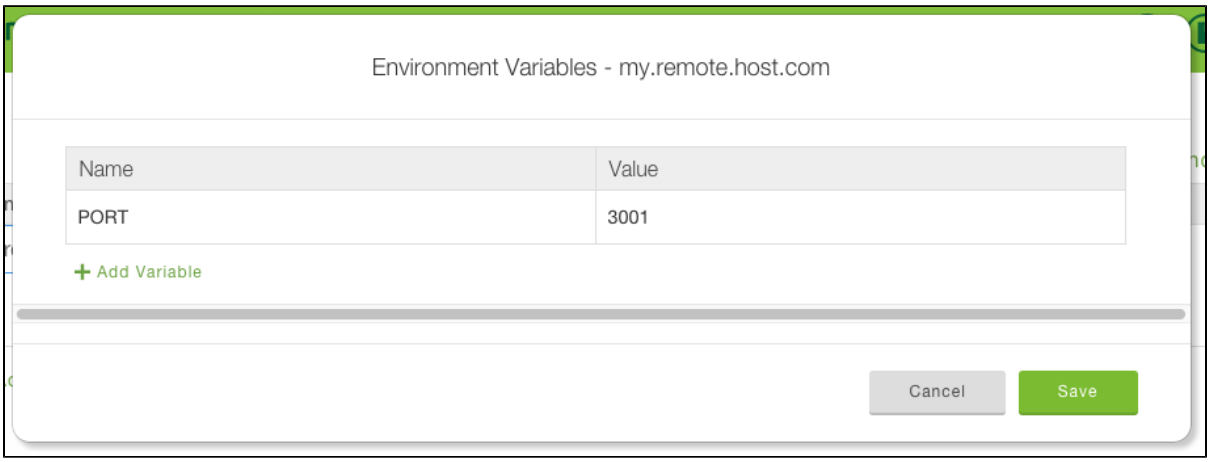
Setting and viewing environment variables with Arc

To view environment variable settings on a remote host, first connect to Process Manager running on the host.

Then click on the inverted triangle next to the host name and choose **Edit host environment**.



You'll see a list of the environment variables defined in that Process Manager's environment:



To add a new environment variable for the selected Process Manager:

1. Click **Add Variable**
2. Enter the environment variable name and value.
3. Click **Save**.

Setting and viewing environment variables with slc

Use `slc ctl env-get` to view all environment variables for a StrongLoop Process Manager environment. For example:

```
$ slc ctl -C http://my.remote.host:1234 env-get my-app
Environment variables:
 STRONGLOOP_CLUSTER=8
 STRONGLOOP_LICENSE=eyJ0eXAiOiJKV1QiLCJhbGciOiJIUz....
```

The license key string in the example above was truncated for brevity.

Use `slc ctl env-set` to set the value of one or more StrongLoop PM environment variables. When you set the value, StrongLoop PM automatically restarts the application it is managing with that new value. This enables you to set the value "on the fly" without redeploying the application. See slc ctl command reference for details.

To set environment variables *before* you deploy an app:

1. Create the service with `slc ctl create my-app`. This creates a service "container" my-app without an actual application.
2. Set environment variables for service my-app with commands like `slc ctl env-set my-app FOO=x BAR=y`.
3. Deploy your application with a command like `slc deploy -s my-app` ...

For example, for a local application with service name my-app:

```
$ cd my-app
$ slc start
$ slc ctl -C http://my.remote.host:1234 env-set my-app DB=mysql
Environment updated: "Updated environment, restarting app"
```

To set more than one variable, separate them with a space:

```
$ slc ctl env-set my-app DB=mysql PORT=8765
Environment updated: "Updated environment, restarting app"
```

For a remote application:

```
$ slc ctl -C http://your.remote.host env-set my-app DB=mysql
Environment updated: "Updated environment, restarting app"
```

Environment variables

Variable	Description
SSH_KEY	Path to the SSH private key to use when connecting with `http+ssh`. Default is current SSH agent.
SSH_PORT	port to use with `http+ssh`. Default is 22.
SSH_USER	Username when connecting with `http+ssh`. Default is your current user.
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.

Controlling Process Manager

Use the `slc ctl` command to control Process Manager at runtime. To control Process Manager remotely, use the `--control` (or `-C`) option to specify the URL where Process Manager is running.

For example, if you start Process Manager with this command:

```
$ slc pm -l 8701
```

Then you can control it with

```
$ slc ctl --control http://localhost:8701 <command>
```

where <command> is one of the sub-commands. See slc ctl command reference for details.

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.

The following sub-commands apply to Process Manager itself:

- `info` - display information on Process Manager.
- `ls` - list services under management.
- `status` (the default sub-command) - Display status information.

You can deploy multiple applications to one Process Manager and then use `slc ctl` to control and get information on the applications (or services) and also monitor individual worker processes for each application (service).

Sub-commands that apply to a specific service (application runtime instance):

- create, remove - create and remove a service.
- env-get, env-set, env-unset - view, set, and remove environment variables.
- log-dump - Dump the log buffer to the console.
- set-size - Change cluster size of the service.
- stop, soft-stop, restart, soft-restart - stop or restart a service.

Profiling commands that apply to a specific worker process:

- cpu-start, cpu-stop - Start and stop CPU profiling.
- heap-snapshot - Save a heap memory snapshot.
- objects-start, objects-stop - Start and stop Object tracking.

Setting application port

Normally, PM makes applications available on a TCP port guaranteed to be different for each app, typically allocating sequential port numbers starting with 3000 as noted above. To override this behavior and make an application available over a specific port, use this command:

```
$ slc ctl env-set <service> PORT=<n>
```

Where <*service*> is the service name or ID, and <*n*> is the port to use. For example to make "my-app" available at port 7777:

```
$ slc ctl -C http://prod.foo.com env-set my-app PORT=7777
```

Do not specify a port already in use. Doing so will cause the application to crash.

Process Manager release notes

The current version of strong-pm is .

The current release of StrongLoop Process Manager is available on npm.

For a list of the latest commits, see the change log.

Known issues

Issue Number	Title	Description

Building and deploying

Strongloop provides tools building and deploying LoopBack applications, specifically:

- Packaging and dependency management.
- Single-step deployment.
- Starting and re-starting deployed applications.

You can use Arc to build and deploy your applications, or you can use `slc`, the command line utility. Both help you to package your application for deployment, and push your application packages to StrongLoop Process Manager that will manage your deployed applications.

Deployment best practices

Overview

Best practices for deploying Node applications are still evolving. The challenges are many-fold: packaging and dependency management, single-step deploy, and starting/re-starting applications with minimal downtime.

StrongLoop Controller supports an integrated deployment workflow:

- Use `slc build` to package your application for deployment.
- Set up StrongLoop Process Manager on your deployment hosts as an application supervisor and manager.
- Finally use `slc deploy` to deploy your application packages to the Process Manager.

Using this workflow with the best practices outlined here will improve the efficiency and reliability of application deployment.

Use the workflow that suits your needs

You can use StrongLoop tools (`slc build, slc deploy`) with StrongLoop Process Manager in multiple ways, depending on your needs; for example:

- If you are deploying to Heroku or other cloud platforms, you can benefit from `slc build`, and its `git push` capability, but you don't need Process Manager.
- If you already have a build process that commits your build products into Git or creates a deployable package with the dependencies built into it, you don't have to replace it with `slc build`, but you can still use `slc deploy` and StrongLoop Process Manager.

Build dependencies into your deployable packages

While the `package.json` file describes application dependencies, it's not suitable for use at deploy-time because running `npm install` during deployment can fetch different versions of modules, leading to untested combinations and potential breakage. Although using npm shrinkwrap av

oids this issue, `npm install` still fetches dependencies live at deploy time. If your npm server or npmjs.org is down, you won't be able to deploy. This is not acceptable.

You can bundle your dependencies, but maintaining bundle specifications manually in the package.json can be problematic: When npmjs.org is down, you won't be able to deploy. So, you need to automate the bundling using `slc build`.

Bundling of Node dependencies is only part of the story. While JavaScript doesn't require compilation, many applications still require building. At deploy time, you may need to fetch front-end dependencies (using Bower, for example). You might need to minimize Javascript code, and fetch or generate other assets. All of this build output needs to be part of the deployable package. This is not trivial, because npm doesn't know what it should include in a package. It usually uses the `.gitignore` to avoid putting ephemeral files into the package, so you'll need to maintain or generate an `.npmignore` file. `slc build` can help with this, too.

And finally, while using an npm package as your deployable artifact works well for some workflows, particularly when the package is archived in something like Artifactory that prefers a single-file package, many workflows prefer Git for archiving builds. Using Git may even be required, if you are deploying to a platform such as Heroku.

Instead, `slc build` will commit your dependencies and source onto a deploy branch, not your development or production branches. It will do this for both npm-installed dependencies and the products of custom build tools such as Bower, Grunt, or Gulp. It will not commit compiled Node binaries from add-ons (unless requested).

Basic use:

```
$ cd to/your/app
$ git clean -x -d # or the equivalent if not using git, always do builds in clean
directories!
$ slc build
$ git log --stat --decorate deploy # to see what was committed to deploy
# or
$ tar -tf ../appname-appversion.tgz # to see what was packed, if app is not using git
```

Be able to push deploys

You want to push deploys when you decide your new app version is deployable, or perhaps have it automatically pushed to staging by your CI tools.

You should only push apps that have been pre-built to deployment, and staging, but alpha servers can have unbuilt apps pushed, for continuous testing against the latest matching dependencies.

And you want to push either Git branches, or npm packages, as appropriate for your workflow.

`slc deploy` does all of the above, pushing your app to the StrongLoop Process Manager. If using Git, it can also push to third-party platforms, though in this case its just a thin wrapper around `git push remote deploy:master`.

Deploy and run your app inside StrongLoop Process Manager

When running an app in deployment (as opposed to development), you need to manage:

- Logging
- Application start/stop/restart (hard and soft)
- Deploying new application versions with zero-downtime upgrade
- Clustering
- Profiling and performance monitoring, etc.

You can use StrongLoop process manager to start an app on your workstation with `slc start.` Once started, you can interact with your app through either a CLI (`slc ctl`) or a graphical interface (Arc), profile it, try out features, and so on.

StrongLoop Process Manager (StrongLoop PM):

- Runs under the control of your system process manager.
- Receives applications deployed to it (with `slc deploy`).
- Runs applications under supervision.

You can deploy both npm packages and Git branches to StrongLoop PM.

Besides exposing all the capabilities of a supervised app (run-time heap snapshot and CPU profile generation, object tracking, worker status and upgrade, and so on), it also supports different application configurations, hard/soft stop and restart, restarting the application on machine boot.

Installing a service so that it runs under your application's process manager is tedious, so the process manager also comes with an installer that

will do this for you, creating a specific user to run the manager as, setting up its run directories, and so on. The automated install currently supports systemd and Upstart 1.4 or 0.6.

First, follow the instructions in Setting up a production host to install StrongLoop PM. Then you can give it a try as a transient process. in the terminal where you did `slc build`:

```
$ slc pm # Starts PM as a transient process
$ slc deploy # if using git
# or
$ slc deploy http:// ../appname-appversion.tgz # if using npm packages
$ slc ctl status # see status of the current app
$ slc ctl heap-snapshot 1 # get heap snapshot of worker 1
$ slc arc # graphical UI alternative to the CLI
# The app was already started, but you can try out a deploy:
```

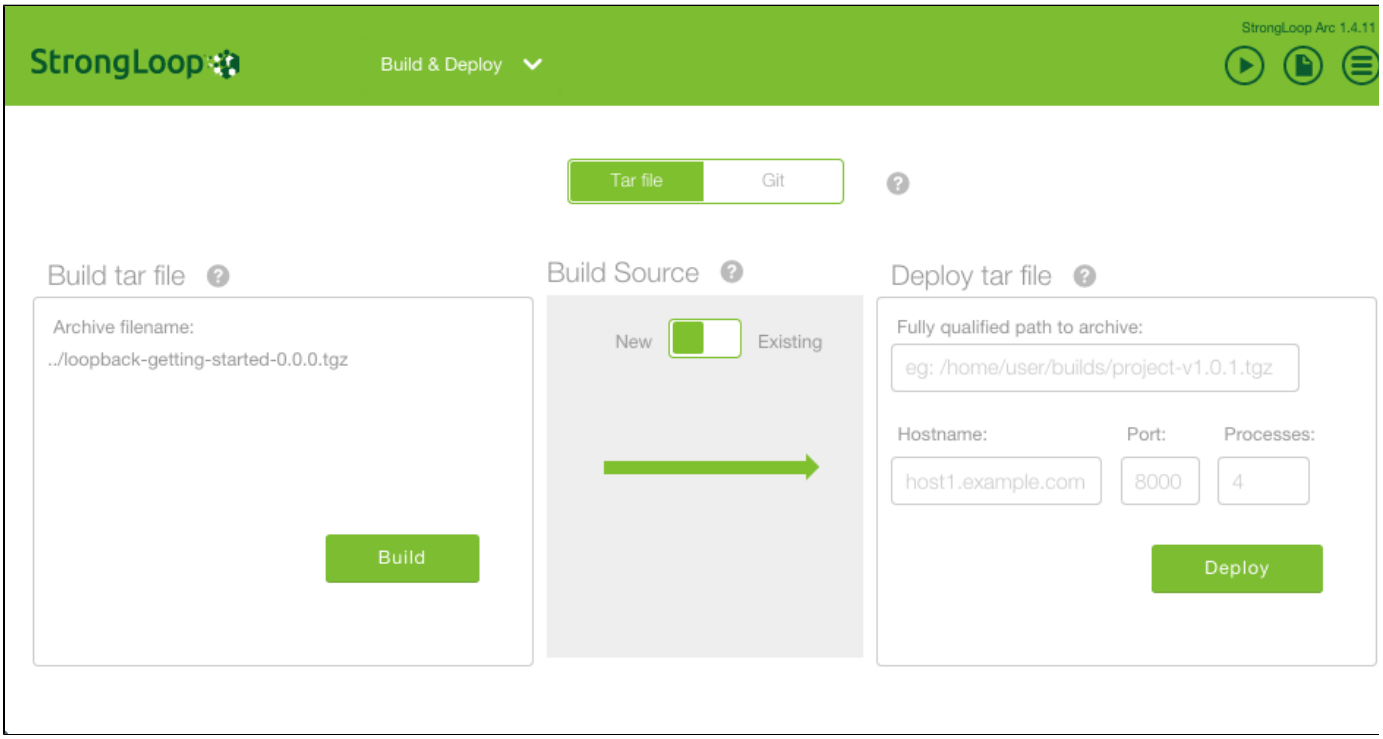
Building and deploying with Arc

StrongLoop Arc Build and Deploy enables you to build and deploy Node applications to StrongLoop Process Manager running remotely or locally.

Here, "build" simply means to create a deployable package–either a tar file or a branch of a Git repository–containing everything required to run the application: not only the Node scripts and other content, but also all the required modules (dependencies) that normally go into the `node_modules` directory, along with any binary artifacts required and compiled as part of the build. See Installing dependencies for more information.

> ⚠ You cannot deploy an application to a Windows system; however, you can build and deploy from a Windows system to a remote Process Manager running (for example) on Linux.

In the module picker, click **Build & Deploy**. You'll see the Build & Deploy module:



Build and deploy procedure

> Currently, Arc can connect to only to a Process Manager hosting an application with service ID of one (1).
>
> If you have multiple applications deployed to a PM, or to change the service ID of a deployed app, use the `slc ctl` command.

Follow these steps to build and deploy your application:

1. Choose a build type: either **Git** or **Tar file**. See Choose build type for details.
2. If you chose Git, enter the name of the branch to use (default is "deploy").
3. Click **Build**.
4. Click **Existing** to redeploy an existing Git branch or tar file. Leave the switch on **New** if you are deploying the package for the first time.
5. Start StrongLoop Process Manager on the deployment system. See Using Process Manager.
   For a production system, refer to Setting up a production host.
6. Enter deploy details, then click **Deploy**.
7. Confirm deployment by monitoring the console output from the StrongLoop Process Manager and verifying your application is running (for example, loading an API endpoint).

Choose build type

First, decide whether you want to build and deploy using a Git branch or a tar file.

Click **Git** or **Tar file**, depending on whether you want to deploy using a Git repository or a .tgz archive file.

NOTE: To use Git, you must:

- Have Git installed. Note additional requirements on Windows.
- Be working in a existing Git repository. That is, you must have created a Git repository for your application and start Arc in the application root directory.

Click **Git** to commit the build to a branch of a Git repository. Committing builds into a Git repository provides robust tracking and storage, including versioning of deployments. See Committing builds to Git for more information. You can then deploy the application from the Git branch to StrongLoop Process Manager.

Click **Tar file** to package the application into a tar file that can be installed with `npm install`. See Creating a build archive for more information. You can then deploy the tar file to StrongLoop Process Manager.

Git build and deploy

> To commit a build to a Git repository, you must have Git installed.

Build to Git

To build to Git, your application must be in a Git repository, and you must be running Arc from a valid clone of the repository.

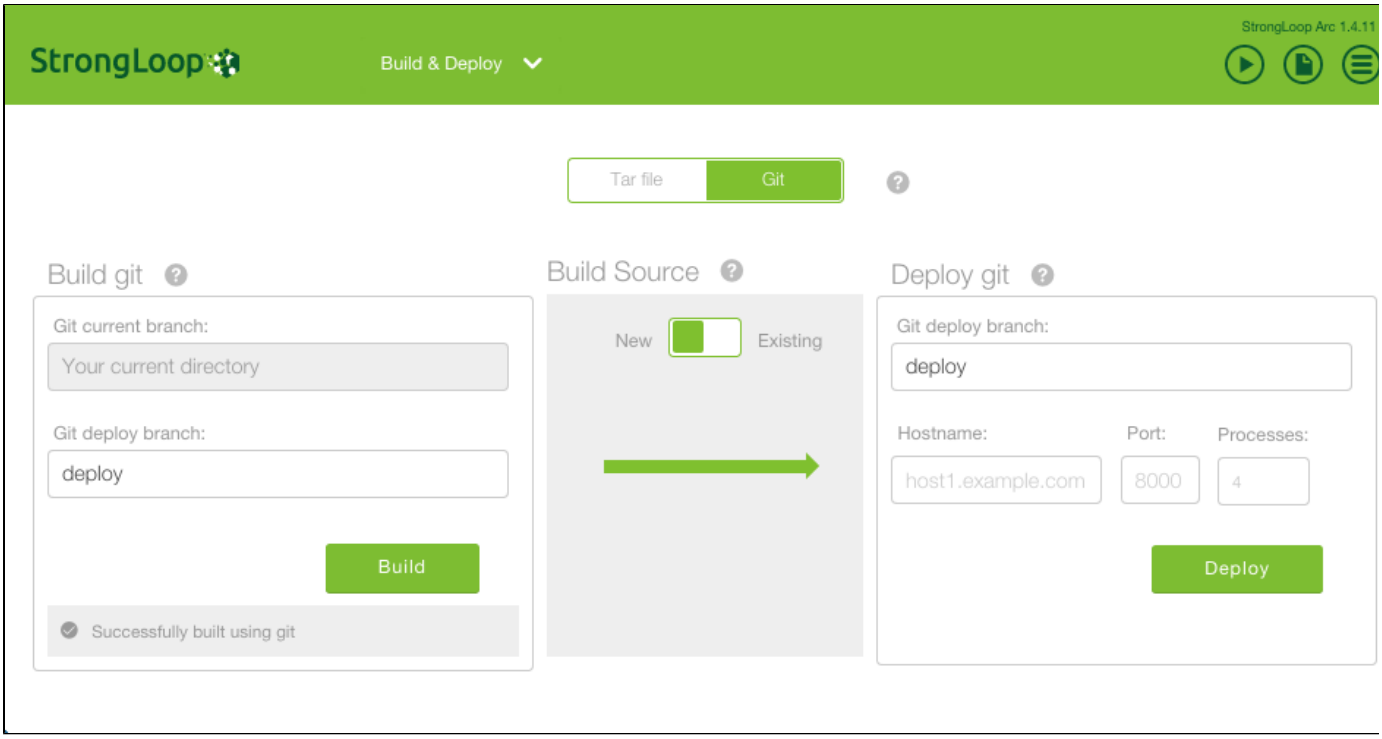
Committing builds into a Git repository provides the most robust tracking and storage, including versioning of deployments. See Committing a build to Git for more information.

StrongLoop Arc commits an exact replica of source and build products to a branch, by default named "deploy." To change the branch name, enter a different name in the **Git deploy branch** field.

After the commit, the deployment branch tip shows as a merge of the deployment and source branches. This enables you to keep a complete history of deployment builds in Git, separate from the development branches.

Click **Build** to commit the build to the specified branch. For details on what happens when you build, see Building applications with slc.

When the build completes, you'll see the message **Successfully built using git**.



Choose build source

Click **New** to create a new Git branch or archive file.

Click **Existing** to use the last Git branch or archive file created.

Deploy from Git

Once you have clicked **Build** to push an application build into a Git repository branch, you can deploy the application from that branch to StrongLoop Process Manager.

But first, you must start StrongLoop Process Manager on the deployment system. To start Process Manager as a transient process, use the `slc pm` command, for example:

```
$ slc pm -l 1234
```

This starts Process Manager listening on port 1234. **NOTE**: By default, StrongLoop Process Manager listens on port 8701; use the `-l` option to specify a different port.

On production systems, install Process Manager as a service. See Using Process Manager for more information.

Enter:

- **Git deploy branch** - Name of the branch to which you deployed the application (by default, named "deploy").
- **Hostname** - Name of the host system you're deploying to.
- **Port** - Port number where Process Manager is listening (for example, the argument to the `-l` option above).
- **Processes** - Number of processes to initially run the application in.

Click **Deploy** to deploy the application to the specified StrongLoop Process Manager instance.

Tar file build and deploy

Build to tar file

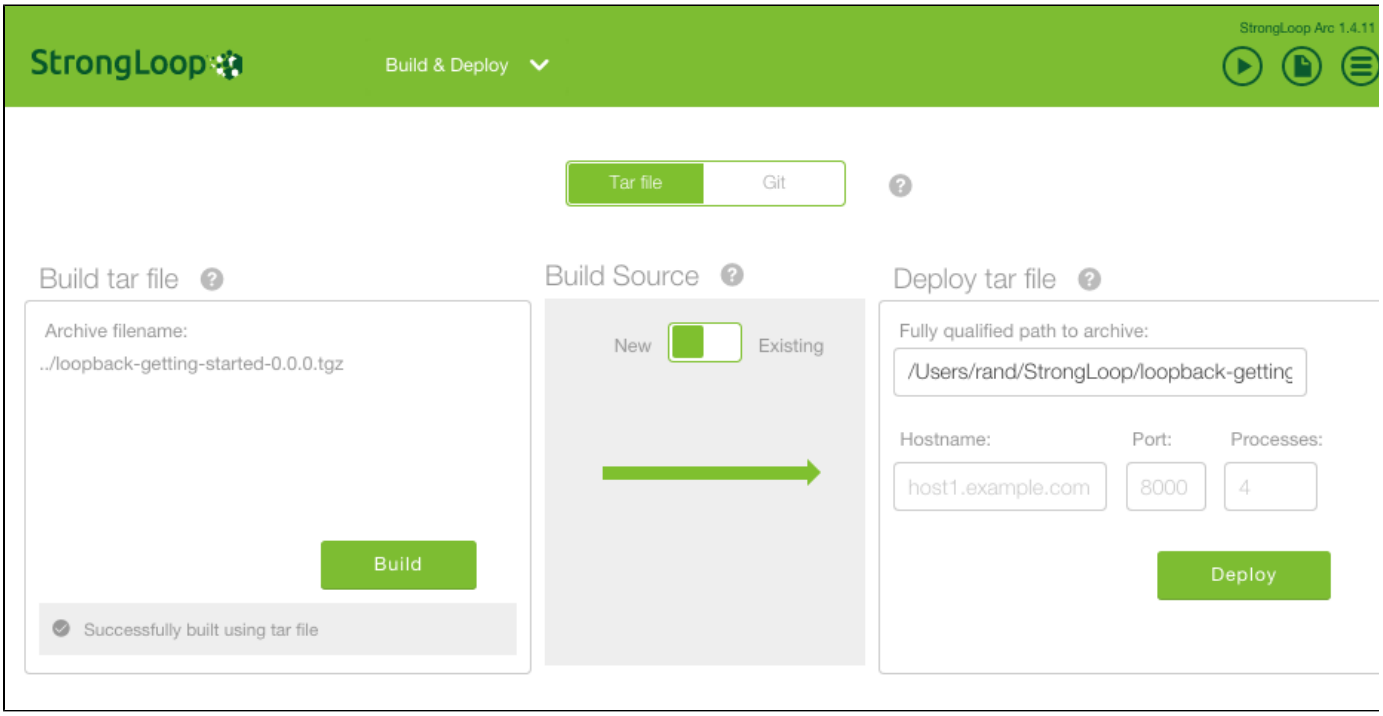
Click **Build** to create a portable archive file, a tar (.tgz) file that can then be installed to StrongLoop Process Manager.

StrongLoop Arc puts the archive file in the parent directory of the application. The archive file name starts with the application name, followed by the version number specified in `package.json`, for example, for the initial version of an application called "myapp," the file is `myapp-0.0.0.tgz`.

You must configure the `package.json` and `.npmignore` files, so building the application will ignore deployment (not development) dependencies and build script output. This is unnecessary when using Git to deploy, but mandatory when creating npm packages! See Installing

dependencies for more information.

When the build process is complete, you'll see the message **Successfully built using tar file**.



Choose build source

Click **New** to create a new Git branch or archive file.

Click **Existing** to use the last Git branch or archive file created.

Deploy from tar file

Once you have clicked **Build** to package an application to an archive (tar) file, you can then deploy the archive to StrongLoop Process Manager.

But first, you must start StrongLoop Process Manager on the deployment system. To start Process Manager as a transient process, use the `slc pm` command, for example:

```
$ slc pm -l 1234
```

This starts Process Manager listening on port 1234. **NOTE**: By default, StrongLoop Process Manager listens on port 8701; use the `-l` option to specify a different port.

On production systems, install Process Manager as a service. See Using Process Manager for more information.

Enter:

- **Fully qualified path to archive** - File path to the .tgz file containing the application package.
- **Hostname** - Name of the host system you're deploying to.
- **Port** - Port number where Process Manager is listening (for example, the argument to the `-l` option above).
- **Processes** - Number of processes to initially run the application in.

Click **Deploy** to deploy the application to the specified StrongLoop Process Manager instance.

Building applications with slc

Use the `slc build` command to build and package a Node application. It creates Git branches or `.tgz` packages that you can deploy to Process Manager with `slc deploy`.

Building and packaging an application involves three basic steps:

1. **Install**: install dependencies, run custom build steps, and prune development dependencies. See Installing dependencies.
2. **Bundle**: modify the `package.json` and `.npmignore` configuration files so dependencies will be packed. See Bundling dependencies.
3. **Create deployable package**, either:
    a. Create a deployable `.tgz` package file. See Creating a build archive.

b. Commit the build onto a Git branch. See Committing a build to Git.

Using `slc build`, you can perform all these steps in a single command, or if you prefer, multiple commands. The documentation describes each step separately, but you can use command options to perform them altogether with a single command.

When using Git, it may be useful to have several different package repositories, for example to manage public and internal releases. See Using multiple package registries for more information.

Shortcut commands

The slc build command provides shortcut options for the two basic workflows: building an npm archive (`.tgz` file) and committing a build to Git.

To create an npm archive (.tgz) file, use this shortcut in your app root directory:

```
$ slc build --npm
```

To commit a build to a Git archive, use this shortcut in your app root directory:

```
$ slc build --git
```

By default, this command commits to the "deploy" branch. Use the `--onto` option to specify a different branch.

Installing dependencies

See also:

Building an application:

1. **Install dependencies**
2. Bundle dependencies
3. Create deploy package, either:
   - Create a build archive
   - Commit a build to Git

Building with slc

The `slc build --install` command automates the common work flow for building application dependencies, specifically:

- `npm install --ignore-scripts`: Install Node dependencies without running scripts to build binary add-ons.
  To run scripts, use the `--scripts` option. See Building binary add-ons below for more information.
- `npm run build`: Specify custom build steps such as `grunt build` or `bower` in the package's `scripts.build` property, since front-end code served by Node commonly requires some amount of preparation.
- `npm prune --production`: Remove development-only tools (such as Bower or Grunt) that the package's build scripts may require, but that should not be deployed.

It runs compilation and install scripts on the deployment server using:

- `npm rebuild`: Compile add-ons for current system.
- `npm install`: Run any install scripts (not typical, but if they exist they may be required to prepare the application for running).

Building binary add-ons

If you're building an app *on exactly the same system architecture* as the deploy target, you can compile and package binary add-ons. To do this, use this command to build:

```
$ slc build --install --scripts
```

Doing this avoids having to to use a compiler on the deployment system, and is recommended when possible.

Listing application dependencies

Use the `slc ctl npmls` command to display all of the dependencies of an application under management by Process Manager. Since the dependency tree may be very large, provide an integer argument to limit the depth of the display to that number of levels; for example:

```
$ slc ctl npmls express-example-app 1
```

See slc ctl command reference for more details.

The result (for express-example-app) is:

Dependency tree for loopback-example-app Expand source

```
express-example-app@1.0.0
 eslint@0.17.1
  chalk@1.0.0
  concat-stream@1.4.8
  debug@2.1.3
  doctrine@0.6.4
  escape-string-regexp@1.0.3
  escope@2.0.6
  espree@1.12.3
  estraverse-fb@1.3.1
  estraverse@2.0.0
  globals@6.4.1
  js-yaml@3.3.0
  minimatch@2.0.7
  mkdirp@0.5.0
  object-assign@2.0.0
  optionator@0.5.0
  strip-json-comments@1.0.2
  text-table@0.2.0
  user-home@1.1.1
  xml-escape@1.0.0
 express@4.12.3
  accepts@1.2.6
  content-disposition@0.5.0
  content-type@1.0.1
  cookie-signature@1.0.6
  cookie@0.1.2
  debug@2.1.3
  depd@1.0.1
  escape-html@1.0.1
  etag@1.5.1
  finalhandler@0.3.4
  fresh@0.2.4
  merge-descriptors@1.0.0
  methods@1.1.1
  on-finished@2.2.1
  parseurl@1.3.0
  path-to-regexp@0.1.3
  proxy-addr@1.0.7
  qs@2.4.1
  range-parser@1.0.2
  send@0.12.2
  serve-static@1.9.2
  type-is@1.6.1
  utils-merge@1.0.0
  vary@1.0.0
 jscs@1.13.1
  chalk@1.0.0
```

```
 cli-table@0.3.1
 commander@2.6.0
 esprima-harmony-jscs@1.1.0-bin
 esprima@1.2.5
 estraverse@1.9.3
 exit@0.1.2
 glob@5.0.5
 lodash.assign@3.0.0
 minimatch@2.0.7
 pathval@0.1.1
 prompt@0.2.14
 strip-json-comments@1.0.2
 vow-fs@0.3.4
 vow@0.4.9
 xmlbuilder@2.6.2
request@2.55.0
 aws-sign2@0.5.0
 bl@0.9.4
 caseless@0.9.0
 combined-stream@0.0.7
 forever-agent@0.6.1
 form-data@0.2.0
 har-validator@1.7.0
 hawk@2.3.1
 http-signature@0.10.1
 isstream@0.1.2
 json-stringify-safe@5.0.0
 mime-types@2.0.11
 node-uuid@1.4.3
 oauth-sign@0.6.0
 qs@2.4.1
 stringstream@0.0.4
 tough-cookie@1.1.0
 tunnel-agent@0.4.0
tap@0.7.1
  buffer-equal@0.0.1
  deep-equal@1.0.0
  difflet@0.2.6
  glob@4.5.3
  inherits@2.0.1
  mkdirp@0.5.0
  nopt@3.0.1
```

```
runforcover@0.0.2
slide@1.1.6
yamlish@0.0.6
```

Next: Bundling dependencies

Bundling dependencies

See also:

Building an application:

1. Install dependencies
2. **Bundle dependencies**
3. Create deploy package, either:
   - Create a build archive
   - Commit a build to Git

The `slc build --bundle` command configures the `package.json` and `.npmignore` files, so `npm pack` ignores deployment (not development) dependencies as well as any build script output. This is unnecessary when using Git to deploy, but mandatory when creating a tar file,

Configuring bundle dependencies

You must list all non-development dependencies, *including optional dependencies*, in the `bundleDependencies` property in `package.json`.

Remember to add every new production dependency to the `bundleDependencies` property. If you don't, npm will try to install them after deployment, creating unexpected and fragile dependencies on npmjs.org.

Keeping this list up to date manually is error-prone, so it's better to let the `slc build --bundle` command to do it for you. However, `slc build` does not modify the `bundleDependencies` property if it is present, so you are free to maintain it yourself, if you wish (or to not use the bundle command).

Using a .npmignore file

Use the `.npmignore` configuration file, file to exclude files from the build package, such as build output and project ephemera such as test output. If npm does not find an `.npmignore` file, it uses `.gitignore` as a fallback. This means that if you have custom build output, such as minimized JavaScript, it will be treated as project ephemera, and not be packed by npm.

The bundle command will create an *empty* `.npmignore` file if there is a `.gitignore` file but there is not `.npmignore` file. This will work for clean repositories, but if you have any project ephemera, they will get packed.

Write and maintain you own `.npmignore` file unless your build process guarantees a clean working repository.

Next:

Create deploy package, either:

- Create a build archive
- Commit a build to Git

Creating a build archive

Use `slc build --pack` to create a portable archive, a tar (.tgz) file that you can then deploy to StrongLoop Process Manager. This is the default for `slc build` if there is no `.git` directory in the current working directory (that is, if the application is not cloned from a Git repository).

`Slc` puts the archive file in the parent directory of the application, to avoid the archive file itself getting packed by future builds, and to make it easier to clean the

See also:

Building an application:

1. Install dependencies
2. Bundle dependencies
3. Create deploy package, either:
   - **Create a build archive**

working repository. The archive file name starts with the application name and version number specified in `package.json`, for example, for the initial version of an application called "myapp," the file is `myapp-0.0.0.tgz`.

- Commit a build to Git

When you subsequently use the `slc deploy` command, it will look for a .tgz file to deploy based on the application name and version number specified in `package.json` in the current directory.

If the bundle command created a `.npmignore` file, check the pack file contents carefully to ensure build products are packed, but project ephemera are not. See Using a .npmignore file for more information.

Committing a build to Git

To commit a build to a Git repository, you must have Git 1.7.9 or later.

Committing builds into a Git repository provides robust tracking and storage and enables versioning of deployments. However, committing both build products and dependencies (`node_modules`) to Git pollutes source branches, creates massive Git commits, and huge churn on development branches and repositories.

In contrast, `slc build --commit` commits an exact replica of current branch source and build products onto a deployment branch. After the commit, the deployment branch tip shows as a merge of the deployment and source branches. This enables you to keep a complete history of deployment builds in Git, separate from development branches. You can push deployment branches to the same repository as the development branches if you wish.

See also:

Building an application:

1. Install dependencies
2. Bundle dependencies
3. Create deploy package, either:
   - Create a build archive
   - **Commit a build to Git**

You can also push branches prepared like this to platforms such as Amazon and Heroku.

The default name of the deployment branch is "deploy". You can configure it with the `--onto` option.

It may be useful to have several different package repositories, for example to manage public and internal releases. See Using multiple package registries for more information.

Using slc build

Committing a build to a Git repository enables you to track deployment states. For example:

```
$ slc build --onto deploy --install --commit
```

This creates a "deploy" branch off the master branch. The "deploy" branch is intended for one-time use for a particular deployment.

For example, in your repository root directory, perform this one-time setup:

```
$ git clone https://github.com/strongloop/loopback-example-appp
$ cd loopback-example-app
$ git checkout -b deploy
$ git checkout master
```

Then build the production branch onto the "deploy" branch:

```
$ git checkout production
$ git clean -x -d     ## optional, but highly recommended
$ slc build --onto deploy --install --commit
$ slc deploy http://deploy.example.com:8765/app
```

See slc build for more information.

Next: Deploy your application to a system running StrongLoop Process Manager; see Deploying applications with slc.

Deploying applications with slc

You can use StrongLoop Process Manager to manage *any* Node application, not just LoopBack applications.

Overview

You can use `slc` to build and package an application and then deploy to StrongLoop Process Manager:

1. Run the StrongLoop Process Manager on the target deployment system. See Using Process Manager.
2. Use `slc build` to install and bundle an application's Node dependencies and create a deployable package. See Building applications with slc and Installing dependencies for more details.
3. Use `slc deploy` to deploy the application locally for testing or to a remote server for staging or production. See Deploying apps to Process Manager.

Zero-downtime updates

StrongLoop Process Manager enables zero-downtime application updates. You can roll out updates to a live application without having to halt it.

When you push an application update with `slc deploy`, Process Manager shuts down workers one at a time, while simultaneously bringing up the new workers with re-deployed code. The old app version runs until it handles all pending TCP connections and afterward closes all existing client connections are gracefully. At the same time, the new app version accepts new client connections. Thus, the application is updated with no downtime or interruption in service.

Setting up a service before deployment

Before you deploy an application, you may want to set up the environment and configure the service. Then, when you deploy the application to the service, it will run as you've configured it immediately.

To do this, first create a service in Process Manager as follows:

```
$ slc ctl create <your-service-name>
```

You don't have to create the service if you've already deployed an application to it: Deploy auto-creates the service (with default environment, and service name from the `package.json`), so you won't have to do `slc create` unless this is a brand new PM to which the app has never been deployed.

Then you can configure the environment, set the cluster size, and so on, before deploying:

```
$ slc ctl env-set <your-service-name> NODE_ENV=production
$ slc ctl set-size <your-service-name> 8
```

Then deploy your application to the service you previously created:

```
$ slc deploy --service=<your-service-name> http://your-pm-instance:port
```

Next time you deploy, the service already exists, and if there is something you want to change, you simply use the following commands:

```
$ slc ctl env-set <your-service-name> FU=BAR
$ slc deploy --service=<your-service-name> http://your-pm-instance:port
```

Deploying apps to Process Manager

See also: slc deploy.

Overview

Once Process Manager is up and listening, you can deploy your application to it. Remember, `slc build` did not build in your binary dependencies, so you could compile your binary add-ons based on your server OS.

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.

Using the slc deploy command

Go to your system and repository that has the deployable application and enter the `slc deploy` command.

Its syntax is:

slc deploy [[-s | --service] <*service*>] http://<*server*>:<*port*> [*package* | *branch*]

Where:

- If you use the -s (or --service) option, the <*service*> argument is the service name, a string identifier for the application runtime instance.
- <*server*> is the server name or IP address.
- <*port*> is the TCP port on which Process Manager is listening.
- Either:
  - *branch* - a branch in the current Git repository. Default is "deploy"
  - *package* - name of a local .tgz file or npm package to deploy. Default is `../<package_name>-<package_version>.tgz`, where the package name and version come from the `package.json` in the current working directory.
    NOTE: Generally, you'll deploy a `.tgz` file created by `slc build` or the Arc Build & Deploy module, though it can be any .tgz file with the format created by npm pack.

For example, this deploys an application from the default "deploy" branch:

```
$ slc deploy http://myserver.myco.com:8701
```

For more information, see slc deploy.

You cannot deploy an application to a Windows system. However, you can deploy from a Windows system to a Linux or Mac OS system.

Deploying multiple applications to one PM

Use the `-s` (or `--service`) option to deploy more than one application to a single instance of Process Manager.

When you deploy an application, simply supply the option and a unique string identifier (the *service* argument). Subsequently, you can use this identifier when using the `slc ctl` command to manage and view status of the application.

Once deployed, the application is available over the network at port number 3000 + ID, where ID is the integer identifier that Process Manager assigns the app. To find the ID, use the `slc ctl` command as shown in the example below.

Process Manager currently assigns consecutive integer IDs to each application, starting with one (1); so the first application you deploy has ID one (1), the second ID two (2), and so on. However, as this assignment algorithm may change, you should always confirm the ID with the `slc ctl` command as illustrated in the example below.

Example

For example, suppose you have StrongLoop Process Manager running on a host, `prod.foo.com` and listening on port 7777 (for simplicity, without secure configuration for this example).

Suppose also you have two applications: app1 and app2 from which you've created build archives (.tgz files) with `slc build`. Let's assume the .tgz files are in the default location (the parent directory of the application root).

Now, assuming in each case that your working directory is the application root directory, you can deploy these two applications as follows:

Deploy app1

```
$ slc deploy -s appone http://prod.foo.com:7777 ../app1-1.0.0.tgz
Deployed `../app1-1.0.0.tgz` to `http://prod.foo.com:7777`
```

And:

Deploy app2

```
$ slc deploy -s apptwo http://prod.foo.com:7777 ../app2-1.0.0.tgz
Deployed `../app2-1.0.0.tgz` to `http://prod.foo.com:7777`
```

The above examples use "appone" and "apptwo" as the identifiers (or service names) for the applications. By default, if you don't supply a name argument, Process Manager will use the name property in the application's `package.json` file.

Now, view status of all applications being run by this Process Manager:

```
$ slc ctl -C http://prod.foo.com:7777
Service ID: 1
Service Name: appone
Environment variables:
  No environment variables defined
Instances:
    Version  Agent version  Cluster size
     4.0.30      1.4.15           4
Processes:
        ID      PID   WID  Listening Ports  Tracking objects?  CPU profiling?
    1.1.22555  22555   0
    1.1.22741  22741   5     prod.foo.com:3001
    1.1.22748  22748   6     prod.foo.com:3001
    1.1.22773  22773   7     prod.foo.com:3001
    1.1.22793  22793   8     prod.foo.com:3001


Service ID: 2
Service Name: apptwo
Environment variables:
  No environment variables defined
Instances:
    Version  Agent version  Cluster size
     4.0.30      1.4.15           4
Processes:
        ID      PID   WID  Listening Ports  Tracking objects?  CPU profiling?
    2.1.22984  22984   0
    2.1.22985  22985   1     prod.foo.com:3002
    2.1.22986  22986   2     prod.foo.com:3002
    2.1.22987  22987   3     prod.foo.com:3002
    2.1.22988  22988   4     prod.foo.com:3002
```

You can see that the first application is available at:

`http://prod.foo.com:3001`

And the second application is available at:

`http://prod.foo.com:3001`

Deploying apps to Docker

Prerequisite: You must first install the Docker Engine on your deployment system. See Docker Installation documentation.

Once you have installed Docker on your deployment (production) host, you can set up your StrongLoop Process Manager to "Dockerize" apps deployed to it.

Running PM as a service

To install StrongLoop PM service so that it automatically converts deployed applications into Docker images and installs them in the Docker Engine running on your deployment system, use this command:

```
$ sudo sl-pm-install --driver docker
```

Then run the StrongLoop PM service, using the command appropriate for the type of Linux system on the server; see:

- Ubuntu
- Red Hat Enterprise Linux

Now, when you deploy an app to StrongLoop PM on this server, your app will be turned into a Docker image and then run in a Docker container.

Running PM as a transient process

To run StrongLoop PM as a transient process (for example, for testing or staging), use the `slc pm` command instead:

```
$ slc pm --driver docker
```

Now, as long as this StrongLoop PM is still running, when you deploy an app to it, the app will be turned into a Docker image and then run in a Docker container.

Using multiple package registries

Overview

The public npm registry at http://npmjs.org is a common way to publish open-source code. When you have proprietary code, you may also want to set up a private package registry.

Using multiple registries for a single project enables you to easily combine both open-source modules from the public npm registry and proprietary modules from your private registry. You may also want to use multiple private registries, for example a company-wide registry for stable modules and multiple team registries with nightly builds.

Use the slc registry command to easily work with multiple registries. It provides the following sub-commands:

- add - Create a new registry configuration with the given name.
- list - List configured registries.
- remove - Remove the specified registry configuration and related files.
- use - Use one of the registry configurations created with the `add` command
- promote - Promote a given package version to another registry.

Prerequisites

If you have not already done so, follow the steps Getting started with StrongLoop Controller (tutorial) to install the `slc` command-line tool.

You'll also need a private package registry. If you want help or advice setting up a private registry, please contact us at callback@strongloop.com.

Configuring slc to use a private registry

Follow these steps to configure slc to use your private npm registry instead of the public npm registry:

1. Add a new configuration entry for your private registry:

   ```
   $ slc registry add private http://your.registry.url/
   ```

   The configuration wizard will prompt you for configuration options:

```
Adding a new configuration "private"
[?] Registry URL: (http://your.registry.url/)
[?] HTTP proxy:
[?] HTTPS proxy:
[?] User name:
[?] Email: (miroslav@strongloop.com)
[?] Always authenticate? (Y/n)
[?] Check validity of server SSL certificates? (Y/n)
Configuration "private" was created.
Run `slc registry use "private"` to let the npm client use this registry.
```

2. Switch your npm client to use the private registry:

```
$ slc registry use private
Using the registry "private" (http://your.registry.url/).
```

3. To switching back to the public npmjs.org registry use the command:

```
$ slc registry use default
Using the registry "default" (https://registry.npmjs.org/).
```

Promoting a package

Assuming you already followed the steps in the previous section to set up a registry, follow these steps to promote a package from a private to a public registry:

1. Publish a package to the private registry

```
$ npm publish
```

2. Promote a package version to the public registry

```
$ slc registry promote --from private --to default my-package@2.0.3
```

More information

For more information, see:

- slc registry command reference.
- Promoting a package

Promoting a package

Once you have followed the steps in Using multiple package registries to set up a registry, follow these steps to promote a package from a private to a public registry:

1. Publish a package to the private registry

```
$ npm publish
```

2. Promote a package version to the public registry

```
$ slc registry promote --from private --to default my-package@2.0.3
```

Scaling

You can scale a Node application:

- To multiple processes on a single host, referred to as *clustering*, also known as vertical scaling.
- To multiple hosts, each running the same application, also typically clustered, also known as horizontal scaling.

Clustering

Clustering refers to running an application in multiple *worker processes* (workers), all receiving requests on the same port.

When StrongLoop Process Manager (PM) runs an application, it automatically runs it in a cluster with a number of workers equal to the number of CPU cores on the system. When you deploy an application with Arc, you can run the application in a multi-process cluster: simply enter the number of processes to run in the **Build & Deploy** module. See Building and deploying with Arc for details.

If any one process in a cluster is terminated or dies due to application error, StrongLoop Process Manager automatically starts up another process and attaches it to the cluster without having to restart the application, thus providing *zero application downtime*. If you inadvertently push a faulty deployment to a cluster, StrongLoop PM ensures rolling restarts: If the first process being restarted crashes, then the deployment is not pushed out to the remaining processes in the cluster.

You can control application clusters with Arc and with `slc`:

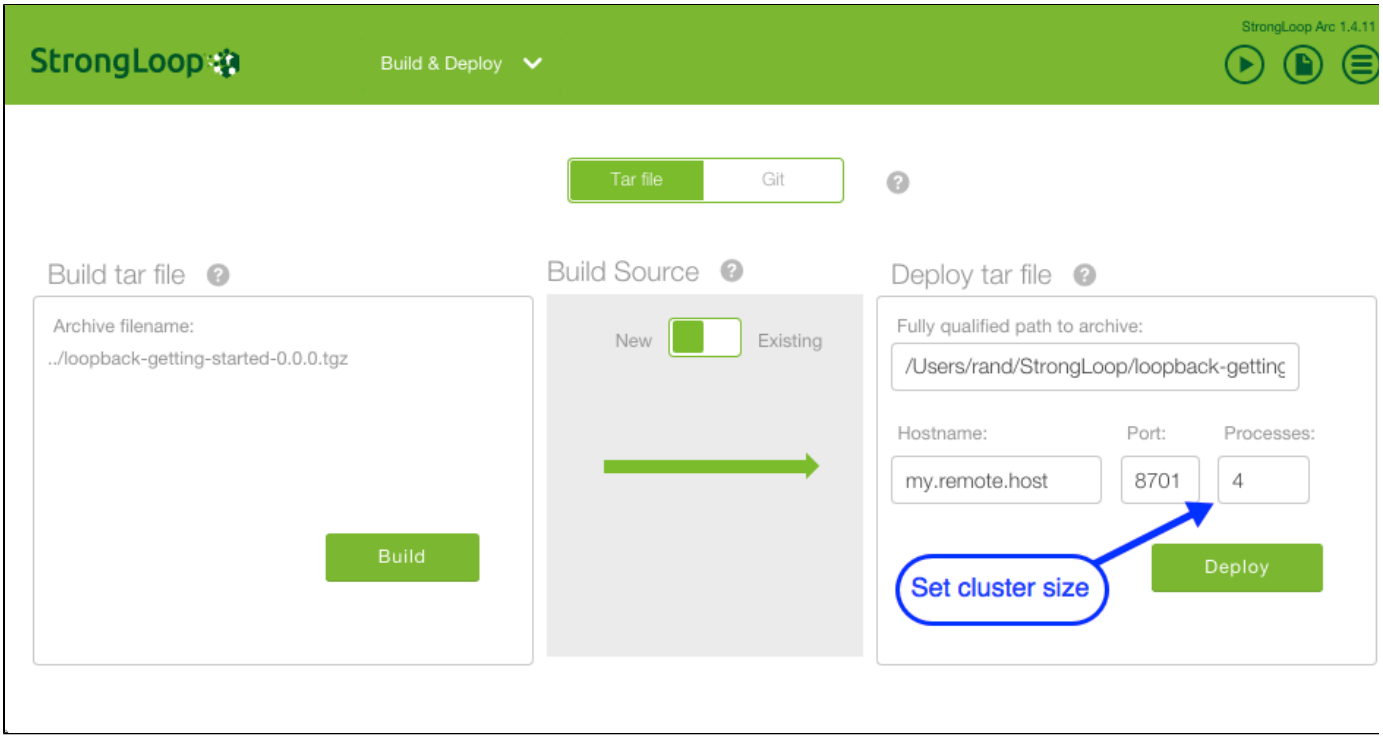
- Controlling clusters with Arc
- Controlling clusters with slc

Controlling clusters with Arc

- Setting up a cluster on deployment
- Resizing a cluster
- Restarting a cluster

Setting up a cluster on deployment

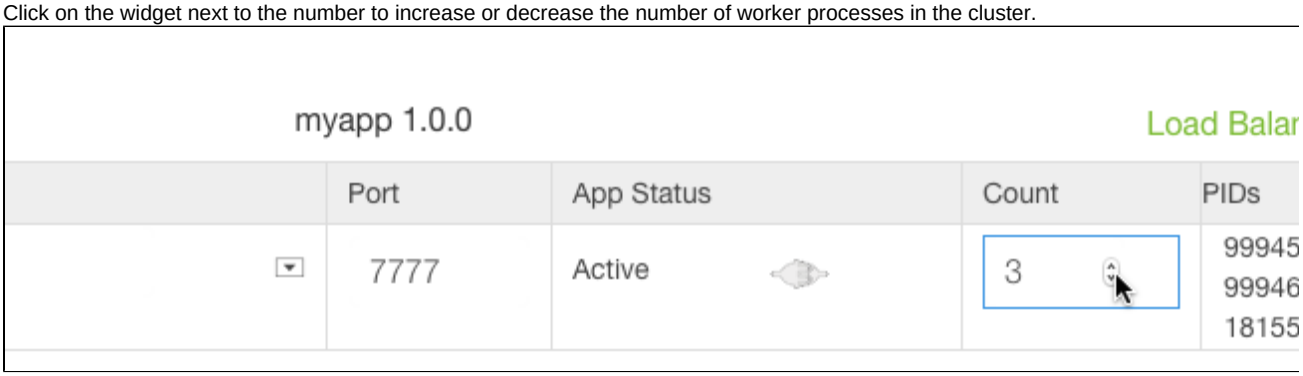
When you deploy an application with Arc, you can set the initial number of worker process in the cluster. Enter a number in the **Processes** field to specify the number of worker processes in the cluster.



Resizing a cluster

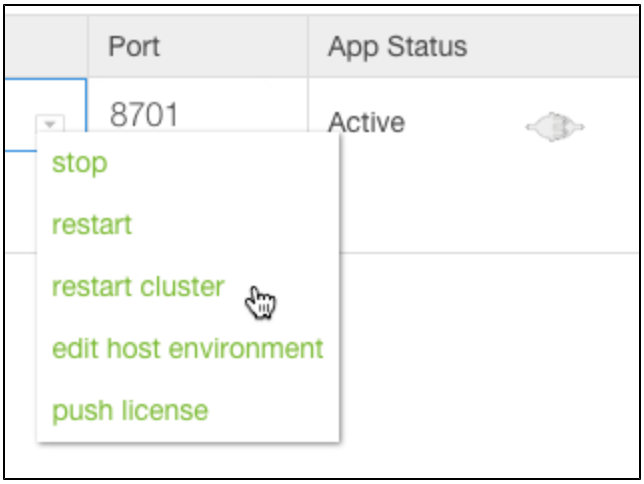
To resize a cluster:

1. Click **Process Manager** to display the Process Manager module.
2. For each Process Manager to which Arc is connected, Arc displays the number of worker processes in the cluster in the **Count** column. Click on the widget next to the number to increase or decrease the number of worker processes in the cluster.



Restarting a cluster

To restart the cluster, click the inverted triangle next to the host name, then choose **Resize Cluster**.



Controlling clusters with slc

Overview

To take advantage of multi-core systems, you can run Node applications as a cluster of *worker processes* (workers), all receiving requests on the same port.

When StrongLoop Process Manager (PM) runs an application, it automatically runs it in a cluster with a number of workers equal to the number of CPU cores on the system. When you deploy an application with Arc, you can run the application in a multi-process cluster: simply enter the number of processes to run in the **Build & Deploy** module. See Building and deploying with Arc for details.

If any one process in a cluster is terminated or dies due to application error, StrongLoop Process Manager automatically starts up another process and attaches it to the cluster without having to restart the application, thus providing *zero application downtime*. If you inadvertently push a faulty deployment to a cluster, StrongLoop PM ensures rolling restarts: If the first process being restarted crashes, then the deployment is not pushed out to the remaining processes in the cluster.

You can control an application cluster running under the control of StrongLoop PM with the `slc ctl` command, with the following sub-commands:

- `status` (the default sub-command): Report the status of the cluster workers.
- `set-size`: Set cluster size to the specified number of workers.
- `restart`: Restart all worker processes.
- `stop`: Shutdown all worker processes and stop the cluster controller.

For example, this command sets the cluster size for a service named `my-app` to four processes:

```
$ slc ctl set-size my-app 4
```

See slc ctl for complete documentation.

Viewing cluster status

Use the default `status` command to view the status of application workers:

```
$ slc ctl
Service ID: 1
Service Name: my-app
Environment variables:
  No environment variables defined
Instances:
    Version  Agent version  Cluster size
     4.1.0       1.5.1           10
Processes:
        ID      PID   WID  Listening Ports  Tracking objects?  CPU profiling?
    1.1.50320  50320   0
    1.1.50321  50321   1     0.0.0.0:3001
    1.1.50322  50322   2     0.0.0.0:3001
    1.1.50323  50323   3     0.0.0.0:3001
    1.1.50324  50324   4     0.0.0.0:3001
```

The command shows the process ID of the master (supervisor) process, the number of workers, and the workers' process IDs.

Resizing clusters at runtime

The `slc ctl` command enables you to resize a cluster at runtime with the `set-size` subcommand.

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

```
$ slc ctl set-size my-app 10
$ slc ctl
Service ID: 1
Service Name: my-app
Environment variables:
  No environment variables defined
Instances:
    Version  Agent version  Cluster size
     4.1.0       1.5.1           10
Processes:
        ID      PID   WID  Listening Ports  Tracking objects?  CPU profiling?
    1.1.50320  50320   0
    1.1.50321  50321   1     0.0.0.0:3001
    1.1.50322  50322   2     0.0.0.0:3001
    1.1.50323  50323   3     0.0.0.0:3001
    1.1.50324  50324   4     0.0.0.0:3001
    1.1.54863  54863   5     0.0.0.0:3001
    1.1.54864  54864   6     0.0.0.0:3001
    1.1.54865  54865   7     0.0.0.0:3001
    1.1.54866  54866   8     0.0.0.0:3001
    1.1.54867  54867   9     0.0.0.0:3001
    1.1.54868  54868   10    0.0.0.0:3001
```

Scaling to multiple servers

Use StrongLoop Process Manager and Arc Manager to scale applications across multiple servers as follows:

1. Install and configure Nginx load balancer (optional).
2. Provision hosts with StrongLoop Process Manager.
3. Deploy the application to each provisioned host with Arc. You can build and deploy with slc if you prefer.
4. Use Arc Manager to activate the application on each host.

Configuring Nginx load balancer

⚠ Using an Nginx load balancer is optional. StrongLoop integrates with Nginx and provides an Nginx Controller via Arc. You can use your own load balancer or none at all, but in that case you are responsible for routing traffic to your hosts appropriately.

Overview

To use Nginx load balancer to route traffic to multiple hosts, follow this procedure to set it up for use with Arc:

1. Install Nginx. See http://nginx.org/en/download.html for instructions.
2. Install the Nginx Controller on the load balancer host, as described below.
3. On the load balancer host, either:
   - Run the Nginx Controller for testing and initial setup.
   - Install the Nginx Controller as an operating system service for production use (supported only on certain Linux distributions--see sl-nginx-ctl-install for details).
4. Add the load balancer in Arc Manager. See Managing multi-server apps.

Installing the StrongLoop Nginx Controller

The StrongLoop Nginx Controller enables you to configure Nginx remotely using Arc. Install it on a load balancer host as follows:

```
$ npm install -g strong-nginx-controller
```

This enables you to run the sl-nginx-ctl and sl-nginx-ctl-install commands:

- Use sl-nginx-ctl to run the Nginx Controller as a transient process for development and testing (not recommended for production).
- Use sl-nginx-ctl-install to install the Nginx Controller as an operating system service.
  NOTE: This command is supported only on Linux distributions that support Upstart or systemd (such as RedHat, Debian, and Ubuntu). See sl-nginx-ctl-install for details.

Running as a transient process

⚠ Running the Nginx Controller as a transient process is only appropriate for development and testing. For production, install it as an operating system service.

Use the `sl-nginx-ctl` command to run the StrongLoop Nginx Controller. See the sl-nginx-ctl command reference for details.

For example, if Nginx is installed in the default location, `/usr/sbin/nginx`, you can run the controller as a transient process (for development) as follows:

```
$ sl-nginx-ctl
```

This runs the Nginx Controller with all the default options:

- Base working directory `.strong-nginx-controller`.
- Control API endpoint URL `http://0.0.0.0:0`.
- Listen endpoint URL for incoming HTTP traffic is `http://0.0.0.0:8080`.

⚠ The user account that NGINX uses must have privileges to listen on both the control and listen ports (endpoints).

Installing as a service

Use the `sl-nginx-ctl-install` command to install the StrongLoop Nginx Controller as an operating system service. See the sl-nginx-ctl-install command reference for details.

This command is supported only on Linux distributions that support Upstart or systemd (such as RedHat, Debian, and Ubuntu). See sl-nginx-ctl-install for details.

By default, the command installs the service as an Upstart (version 1.4) job and runs the Controller with user account strong-nginx-controller.

For example:

```
$ sl-nginx-ctl-install
```

This runs the command with the defaults:

- Base directory working directory `.strong-nginx-controller`.
- Control API endpoint `http://0.0.0.0:0/`.
- Listens for incoming HTTP traffic at `http://0.0.0.0:8080`.
- Path of Upstart job is `/etc/init/strong-nginx-controller.conf`.
- Path to Nginx binary `/usr/sbin/nginx`.

Managing multi-server apps

With Arc, you can manage applications running under control of StrongLoop Process Manager on multiple server hosts and configure an Nginx load balancer to route traffic to the hosts.

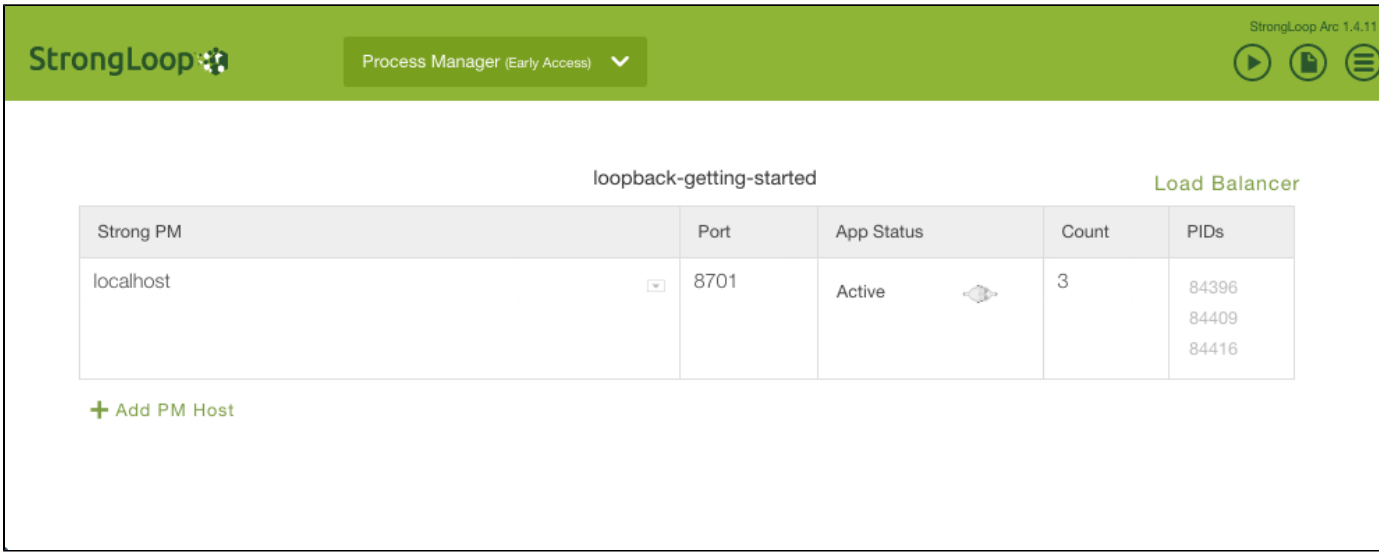
Prerequisites

Install and run StrongLoop Process Manager on each remote host where you want to run applications. See Setting up a production host for details.

If you want to use an Nginx load balancer (optional), see Configuring Nginx load balancer for setup instructions.

Adding hosts

Click **Process Manager** in module picker to display Arc Process Manager:



The page lists all the known hosts running StrongLoop PM.

To add a new host:

1. Click **Add PM Host**.
2. Enter the fully-qualified domain name of the host, or select from the list of previously-used hosts.

Make sure you enter *only* the host name. Don't include "http://".
For example, to connect to a Process Manager running on the local system, just enter "localhost", not "http://localhost".

3. Enter the port number on which StrongLoop PM is running on the host.
   App Status for the host displays “Inactive.”
4. Click **Activate** to connect to Process Manager on that server.

Once the host has been found, **App Status** changes to “Active”, and the Processes column lists the number of processes running on the host and their process IDs.

The name of the app running on the host appears in the header.

- The first PM Host activated will set the application for the Process Manager. All subsequent PM Hosts added MUST be running the same application and version. The application and version are read from the Node application's `package.json` file.
- Click the context menu button next to the app name, to display controls for managing the app (Start, Stop, and Restart).

Deleting a host

To delete a host, highlight the row by clicking on the row in a non-input area: A “delete” button (x) appears at the end of the row. Click to delete the host from the list.

Deleting a host here simply removes it from Arc Process Manager; it does not shut down any processes or services on the host.

Adding a load balancer

Using an Nginx load balancer is optional. StrongLoop integrates with Nginx and Arc provides an Nginx Controller.

You can use a different load balancer or none at all, but in that case you are responsible for routing traffic to your hosts appropriately.

Once you have installed and configured Nginx and the StrongLoop Nginx Controller, add the load balancer in Arc Manger.

Click **Load Balancer** in the Arc Process Manager module:

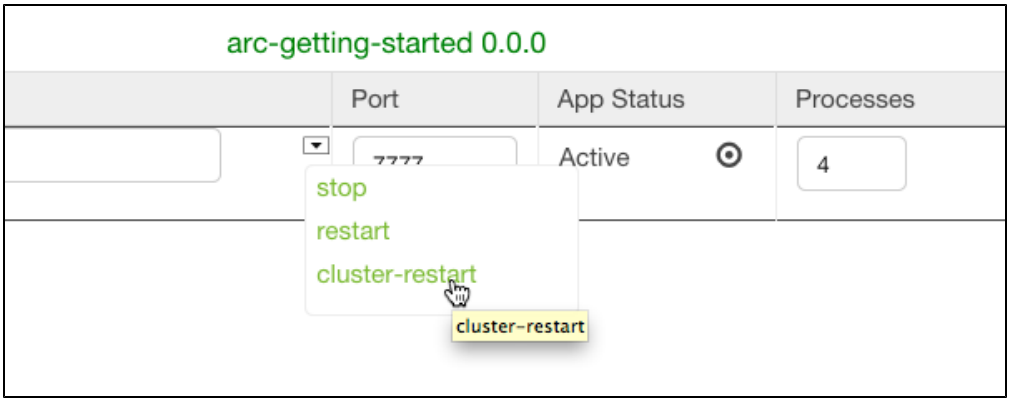
Enter the host name and port number where the StrongLoop Nginx controller is listening for control messages, then click **Save**. These are the hostname and port specified with the `-c` option to the sl-nginx-ctl and sl-nginx-ctl-install commands. See Managing multi-server apps (Adding a load balancer) for more information.

NOTE: You must have already installed and configured the Nginx load balancer and StrongLoop Nginx controller on the host. See Configuring Nginx load balancer for more information.

Once you've add an Nginx load balancer as shown above, it will automatically distribute client requests evenly to your application (API) running under StrongLoop Process Manager on multiple hosts. You can subsequently add additional hosts, and the configured load balancer will then route requests to the new hosts as well.

Controlling applications

Click next to a Process Manager instance to control the application that Process Manager is running.



On the menu that appears, click:

- **Stop** to stop the application running in that Process Manager.
- **Restart** to restart the application; this is a "hard restart" that will stop and then start all the application worker processes. If changes were done to the application's configuration, the new configuration settings would take affect.
- **Cluster-restart** to perform a "rolling restart" of the application; this enables you to restart the application with **zero downtime with the current application configuration**.

When you re-deploy an application, StrongLoop Process Manager will automatically perform a rolling restart so that the application is updated with no downtime. You don't have to manually restart it.

Developing clustered apps

Use the following modules when developing clustered applications, for example to share state between application instances:

- Socket IO store for clusters
- Strong Cluster Connect Store
- Strong Cluster TLS Store
- Strong Store for Cluster
- Strong MQ

Socket IO store for clusters

You cannot use this module with StrongLoop Process Manager.

See also Socket IO store API

Overview

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

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

Installation

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

Using Socket IO Store

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

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

Configuration

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

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

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

Setting up the master process

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

```
// master.js

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

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

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

Strong Cluster Connect Store

You cannot use this module with StrongLoop Process Manager.

See also Strong-cluster-connect-store API

Overview

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

Strong Cluster Connect Store:

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

Installation

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

Configuration

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

Configuration for Connect

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

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

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

Configuration for Express 3.x

Express 3.x is a web application framework that uses Connect; therefore, Strong Cluster Connect Store also works with Express. Use the following code to configure Strong Cluster Connect Store for use with Express 3.x:

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

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

Configuration for Express 4.x

Express 4.x moves middleware like session and cookie-parser to standalone packages. Use the following code to configure Strong Cluster Connect Store for use with Express 4.x:

```
var express = require('express');
var cookieParser = require('cookie-parser');
var session = require('express-session');
var ClusterStore = require('strong-cluster-connect-store')(session);

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

Setting up the master process

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

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

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

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

Using Strong Cluster Connect Store

The following example assumes you have set up Strong Cluster Connect Store for Express and have run the steps in "Setup for for the Master Process."

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

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

  ClusterStore.setup();

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

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

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

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

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

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

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

Strong Cluster TLS Store

You cannot use this module with StrongLoop Process Manager.

See also Strong-cluster-tls-store

Overview

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

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

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

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

You should therefore monitor the performance of your application to determine the performance boost (if any) in your specific scenario.

Installation

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

Configuration

TLS server

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

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

HTTPS Server

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

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

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

Connect and Express

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

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

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

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

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

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

Using multiple servers

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

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

Setting up the master process

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

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

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

Setting up the client

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

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

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

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

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

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

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

Using Strong Cluster TLS Store

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

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

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

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

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

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

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

} else {
  // workers execute this code

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

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

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

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

Strong Store for Cluster

You cannot use this module with StrongLoop Process Manager.

See also Strong-store-cluster API

Overview

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

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

Session management

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

How Strong Store for Cluster works

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

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

Example

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

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

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

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

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

Strong MQ

To use Strong MQ with StrongLoop Process Manager, you must use AMQP as the messaging protocol provider, not the native provider .

Overview

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

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

Message Patterns

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

Installation

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

Example

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

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

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

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

Known error

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

Event: 'error'

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

Messages

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

Queues

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

Messaging protocol providers

Strong MQ supports the following protocol providers:

- Native
- AMQP
- STOMP

Native

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

It has no options.

The URL format is:

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

AMQP

This provider is based on the node-amqp module and provides support for RabbitMQ using the AMQP protocol. See its documentation for more information.

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

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

The URL format for specifying the options above is:

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

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

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

STOMP

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

The options are:

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

The URL format for specifying the options above is:

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

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

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

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

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

Strong MQ API

Error rendering macro 'markdown-url' : Error 503 retrieving server data from URL.

View docs for strong-mq in GitHub

Logging

Overview

Logging is important for debugging and auditing applications. StrongLoop provides several options for logging:

- Basic logging using the Node `console.log()` function.
- Using `slc ctl log-dump` command to dump the last 1 MByte of log file content .
- Logging with third-party libraries such as Winston and Bunyan.

In production, when an application is running under control of the StrongLoop PM service, logging is controlled by Upstart or Systemd.

Best practices

To create effective application logs, follow these guidelines:

- Ensure that logs contain a timestamp and process ID from the process that generated the event.
- Log one event per line.
- Log as much as possible within reason.
- Leave log-level-based filtering to the external log processor/viewer.
- Leave aggregation of logs to the supervisor process.
- If logging to a file, just log to `stdout` or `stderr` and leave the file management to the supervisor process.

The StrongLoop command-line tool `slc` simplifies logging by taking care of some of the concerns for you.

Logging to the console

The simplest way to do logging is to use the standard `console.log()` function that prints logs to stdout or stderr; for example:

```
console.log('This is a log entry')
```

You can also use printf-style formating options; for example:

```
console.log('This is %s', 'log entry 1')
```

For more information on `console.log()`, see the Node.js documentation.

Logging for a local application

Run a local application named my-app under control of StrongLoop Process Manager:

```
$ slc start
```

Then, you can dump the last 1 MByte of log file content to the console with this command:

```
$ slc ctl log-dump my-app
```

Use the `--follow` option to continuously dump the logs to the the console:

```
$ slc ctl log-dump my-app --follow
```

The `slc ctl log-dump` command actually *consumes* the log messages, so if more than one person calls it for one application, only one of them will see the logs.

Logging in production

Follow the instructions in Setting up a production host to install and run StrongLoop PM as a service. Logging behavior then depends on the operating system (whether you're using Upstart or systemd).

With Upstart 0.6, logs go to syslog so that syslog can handle log rotation, which means you have to configure syslog to redirect log entries if you want them rotated.

With Upstart 1.4, log rotation and direction is handled by Upstart itself and logs are saved to `/var/log/upstart/strong-pm.log`.

Everything run under systemd is logged to `journald`, which is accessible through the `journalctl` command. See the journalctl man page for more information about `journalctl`. For convenience, StrongLoop systemd service definitions *also* log to syslog.

Using logging libraries

Using Winston

Winston is one of the most popular Node logging libraries. It is simple and supports logging to multiple transports.

Install Winston:

```
$ npm install winston
```

Use Winston:

```
...
var winston = require('winston');

winston.log('info', 'This is a log event', {timestamp: Date.now(), pid: process.pid});
winston.info('This is another log event', {timestamp: Date.now(), pid: process.pid});
...
```

Transports

Winston comes with file- and console-based transports, but many others are available on npmjs.org. For example:

```
...
winston.add(require('winston-irc'), {
  host: 'irc.somewhere.net',
  nick: 'logger',
  pass: 'hunter2',
  channels: {
    '#logs': true,
    'sysadmin': ['warn', 'error']
  }
});
...
```

Winston also has a Splunk transport:

```
...
winston.add(require('winston-splunk').splunk, {
  splunkHostname: 'thishost.somewhere.net',
  splunkHost: 'splunk.somewhere.net',
  splunkPort: 54321,
  splunkFacility: nodejs
});
...
```

For information on additional transports and other options, see https://github.com/flatiron/winston.

Using Bunyan

Bunyan provides structured machine-readable logging.

Install Bunyan:

```
$ npm install bunyan
```

Use Bunyan; for example:

```
...
var bunyan = require('bunyan');
var log = bunyan.createLogger({name: 'myapp'});
log.info('hi');
log.warn({lang: 'fr'}, 'au revoir');
...
```

The above code would produce the output such as the following:

```
...
{"name":"myapp","hostname":"localhost.localdomain","pid":17095,"level":30,"msg":"hi","
time":"2014-06-15T03:18:15.707Z","v":0}
{"name":"myapp","hostname":"localhost.localdomain","pid":17095,"level":40,"lang":"fr",
"msg":"au revoir","time":"2014-06-15T03:18:15.709Z","v":0}
...
```

Child loggers

Bunyan enables you to create separate child loggers for each sub-component of your application.

For example, in Express you might use a child logger per request.

```
...
var bunyan = require('bunyan');
var log = bunyan.createLogger({name: 'myapp'});

app.use(function(req, res, next) {
req.log = log.child({reqId: uuid()});
next();
});

app.get('/', function(req, res) {
req.log.info({user: ...});
});
...
```

This would produce a log such as the following:

```
{"name":"myapp","hostname":"localhost","pid":4501,"level":30,"reqId":"XXXX-XX-XXXX","u
ser":"...","time":"2014-06-16T07:34:26.345Z","v":0}
```

Using Splunk

Overview

Splunk searches, monitors, analyzes and visualizes machine-generated big data from websites, applications, servers, networks, sensors and mobile devices.

Run Splunk locally to log to a local file or syslog and configure Splunk to read the logs from there. Run Splunk on a remote server and configure the Splunk universal forwarder to securely forward logs events to the Splunk server.

Configure application logging

First, install a logging middleware module; for example:

```
$ npm install --save morgan
```

Alternatively you can also use Bunyan, Winston, Strong-logger or other popular Node logging frameworks.

Then, add the logging middleware to your application.

Adding logging middleware to a LoopBack application

For a LoopBack application, add logging middleware as described in Defining middleware:

Edit `server/middleware.json` and add the logging middleware in the initial phase:

```
...
  "initial": {
    "compression": {},
    "morgan" : {},
    "cors": {
    ...
  },
...
```

Running the application

This example is for a LoopBack application.

Start the application locally:

```
$ slc start
```

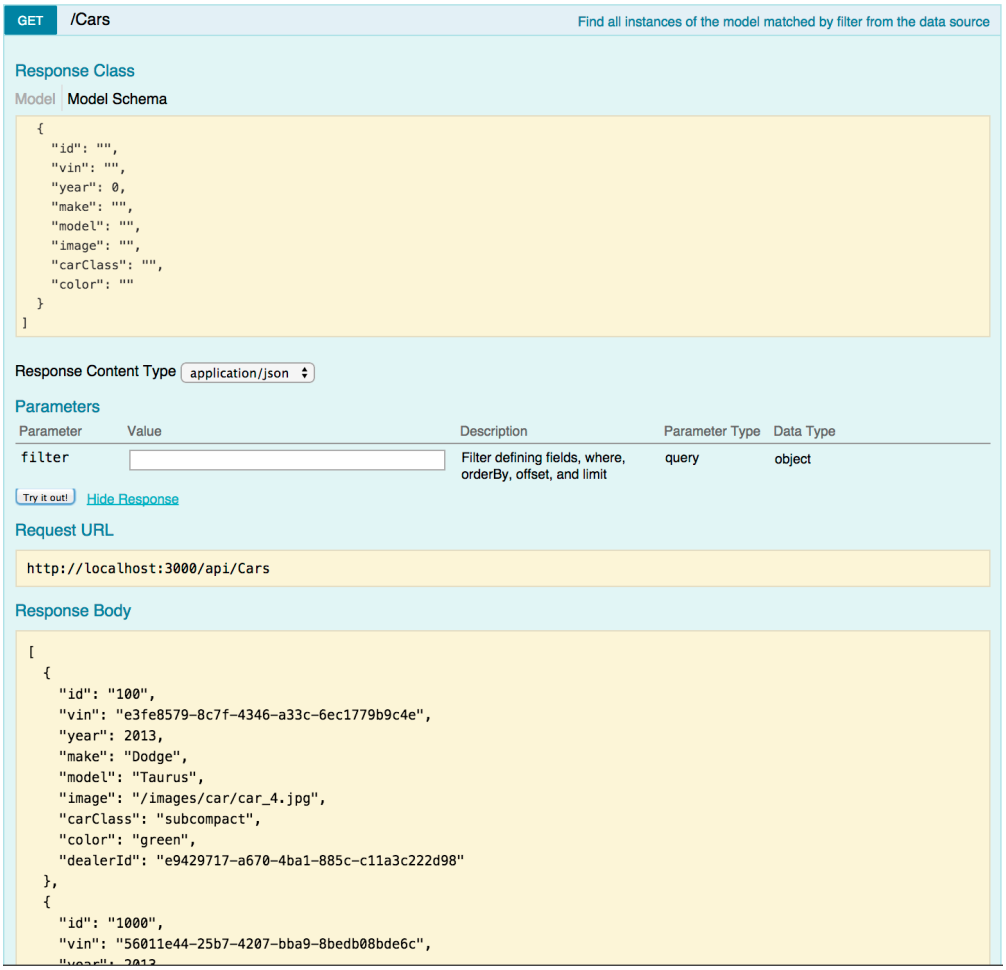
If you're running on a remote host, then build and deploy to Process Manager running on your remote host.

Then dump the log to the console as follows:

```
$ slc ctl log-dump
```

This dumps the last MB of logs to the console.

Now execute a `GET` query from the API explorer and check the generated logging on the console log:



The default middleware logger logs to console output. However since the application is running in a cluster, it will log to a managed log file called `supervisor.log` under `<application root>`

```
2014-09-15T22:50:40.109Z pid:1431 worker:1 GET /api/Cars 304 60.964 ms - -
```

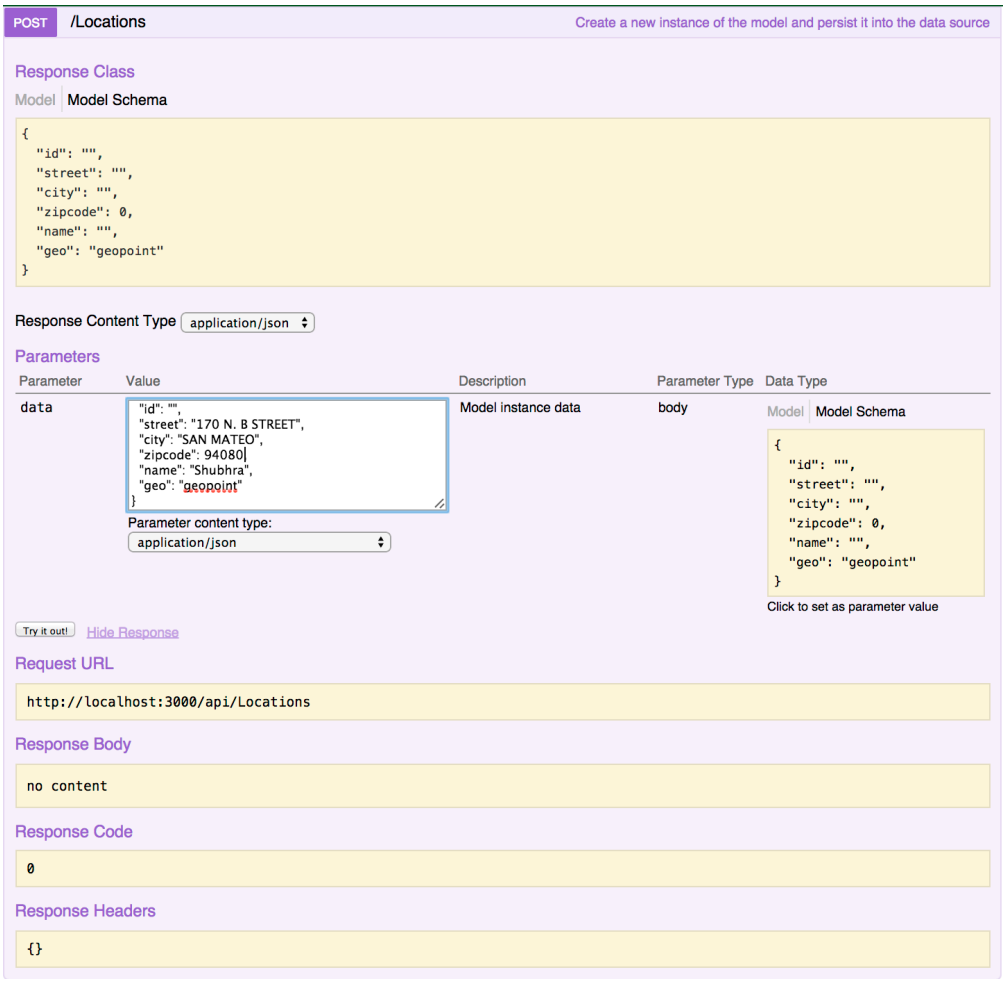
Here you see the logged entry of the API call with the `Timestamp, Process ID (1431), Worker ID, Response Code and Response Time Logged`

Execute a few other API endpoint calls from the Loopback API explorer and check associated log entries.

```
2014-09-15T23:36:24.792Z pid:1431 worker:1 GET /api/Cars/count 304 2.933 ms - -
2014-09-15T23:36:35.759Z pid:1431 worker:1 GET /api/Locations 304 3.902 ms - -
2014-09-15T23:38:35.289Z pid:1431 worker:1 AssertionError: must provide a string
"lng,lat" creating a GeoPoint with a string
2014-09-15T23:38:35.289Z pid:1431 worker:1     at new GeoPoint
(/Users/shubhrakar/Logging_Example/node_modules/loopback-datasource-juggler/lib/geo.js
:124:5)
2014-09-15T23:38:35.289Z pid:1431 worker:1     at GeoPoint
(/Users/shubhrakar/Logging_Example/node_modules/loopback-datasource-juggler/lib/geo.js
:119:12)
2014-09-15T23:38:35.289Z pid:1431 worker:1     at
ModelConstructor.Object.defineProperty.set [as geo]
(/Users/shubhrakar/Logging_Example/node_modules/loopback-datasource-juggler/lib/model-
builder.js:399:81)
2014-09-15T23:38:35.291Z pid:1431 worker:1     at new ModelConstructor
(/Users/shubhrakar/Logging_Example/node_modules/loopback-datasource-juggler/lib/model-
builder.js:165:22)
2014-09-15T23:38:35.291Z pid:1431 worker:1     at Function.DataAccessObject.create
(/Users/shubhrakar/Logging_Example/node_modules/loopback-datasource-juggler/lib/dao.js
:146:11)
2014-09-15T23:38:35.291Z pid:1431 worker:1     at SharedMethod.invoke
(/Users/shubhrakar/Logging_Example/node_modules/loopback/node_modules/strong-remoting/
lib/shared-method.js:207:17)
2014-09-15T23:38:35.301Z pid:1430 worker:0 ERROR supervisor worker id 1 (pid 1431)
accidental exit with 8
2014-09-15T23:38:37.350Z pid:1430 worker:0 INFO supervisor started worker 2 (pid 1516)
2014-09-15T23:38:46.653Z pid:1516 worker:3 GET /api/Locations 304 18.060 ms - -
2014-09-15T23:39:09.785Z pid:1516 worker:3 GET /api/Locations/count 200 3.140 ms - 12
```

The above log shows two “Get()” operations, `/api/Cars/count` and `/api/Locations`, and then a POST operation, where the application encountered an assertion error:

```
AssertionError: must provide a string "lng,lat" creating a GeoPoint with a string
```



This error resulted in the running worker process crashing. However, because the cluster is being managed by StrongLoop Controller, another worker process (1516) is spun up and added to the master. Consecutive API calls are automatically routed to the new worker:

```
GET /api/Locations
GET /api/Locations/count
```

Logging to Syslog

When running an application in production, the StrongLoop Process Manager service logs directly to syslog.

For example, below is shown the tailing output of syslog (`/var/log/system.log`). You can see the two worker processes 1553 and 1554 log to syslog, where as the master process 1551 logs to `supervisor.log` as usual.

```
Sep 15 17:08:37 MacBook-Pro.local node[1552]: pid:1553 worker:1 GET
/explorer/resources/inventory 304 1.512 ms - -
Sep 15 17:08:37 MacBook-Pro.local node[1552]: pid:1554 worker:2 GET
/explorer/resources/Customers 304 4.048 ms - -
Sep 15 17:08:37 MacBook-Pro.local node[1552]: pid:1554 worker:2 GET
/explorer/resources/Cars 304 1.360 ms - -
Sep 15 17:08:37 MacBook-Pro.local node[1552]: pid:1553 worker:1 GET
/explorer/resources/Customers 304 2.394 ms - -
Sep 15 17:08:37 MacBook-Pro.local node[1552]: pid:1554 worker:2 GET
/explorer/resources/inventory 304 1.185 ms - -
Sep 15 17:08:37 MacBook-Pro.local node[1552]: pid:1553 worker:1 GET
/explorer/resources/Locations 304 2.074 ms - -
Sep 15 17:08:37 MacBook-Pro.local node[1552]: pid:1553 worker:1 GET
/explorer/resources/notes 304 1.418 ms - -
Sep 15 17:08:37 MacBook-Pro.local node[1552]: pid:1553 worker:1 GET
/explorer/images/throbber.gif 304 0.490 ms - -
```

Invoking one `get()` Operation from the API explorer routed the call to worker process 1 as indicated by the logging in Syslog.

```
Sep 15 17:09:48 MacBook-Pro.local node[1552]: pid:1553 worker:1 GET /api/Locations 304
14.850 ms - -
```

Using Splunk

If you don't already have it, download Splunk Enterprise and install the version for your operating system.

Once installed, boot the `splunkd` daemon process with the Splunk start scripts. For example, if you installed Splunk to `/opt/splunk` then start Splunk as follows:

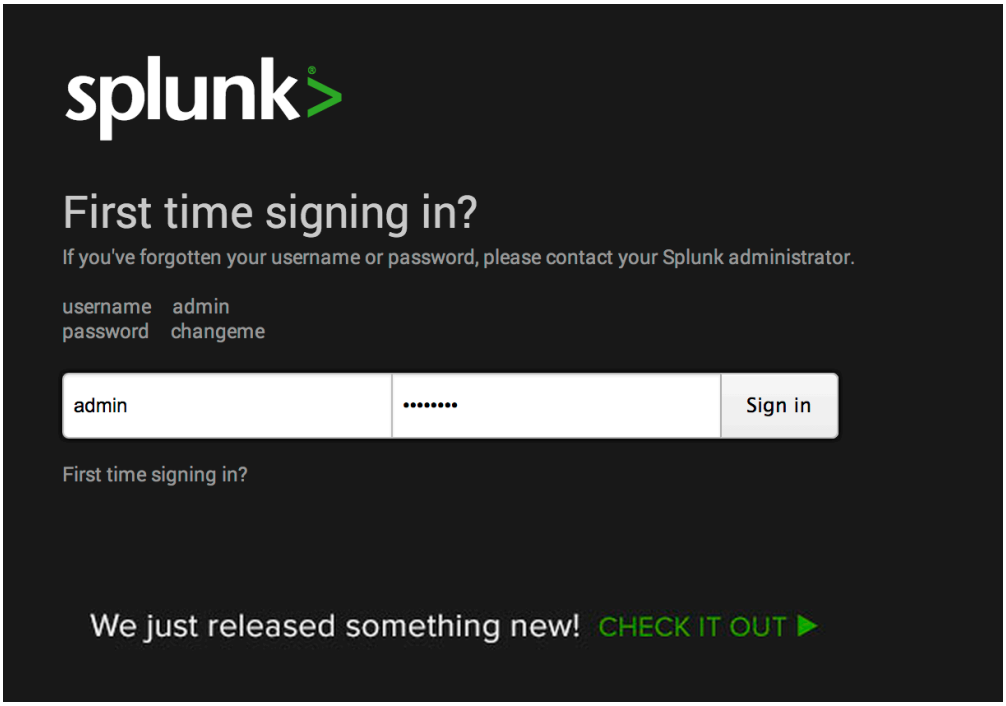
```
$ sudo /opt/splunk/bin/splunk start
$ echo "User/pass: admin/changeme"
$ sudo /opt/splunk/bin/splunk enable listen 9997
```

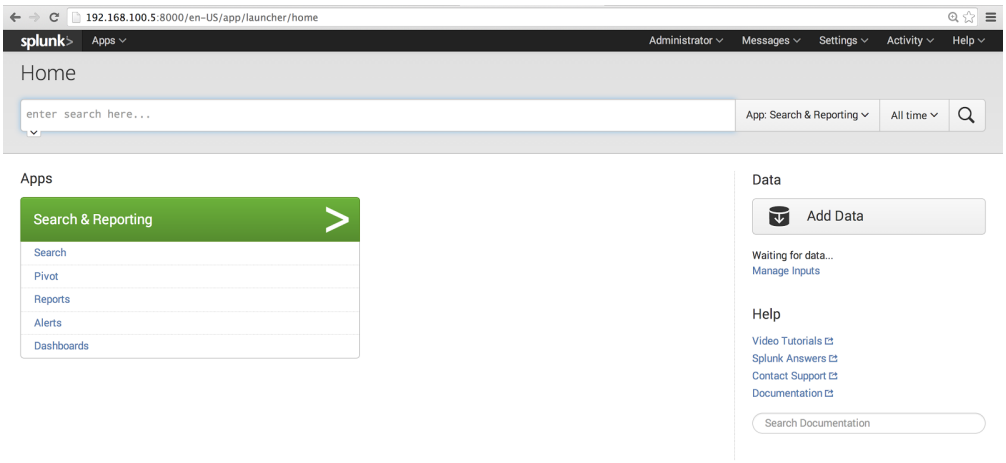
The last command configures the listener port as 9997.

After startup, the Splunk server provides:

- Web interface : available usually on http://hostname:8080
- Management interface usually on http://hostname:8089
- Listening interface on supplied port 9997

Log in into Splunk using the admin console:

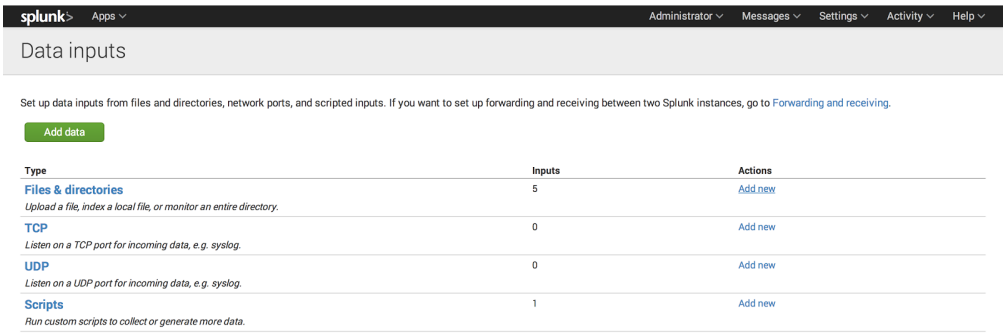


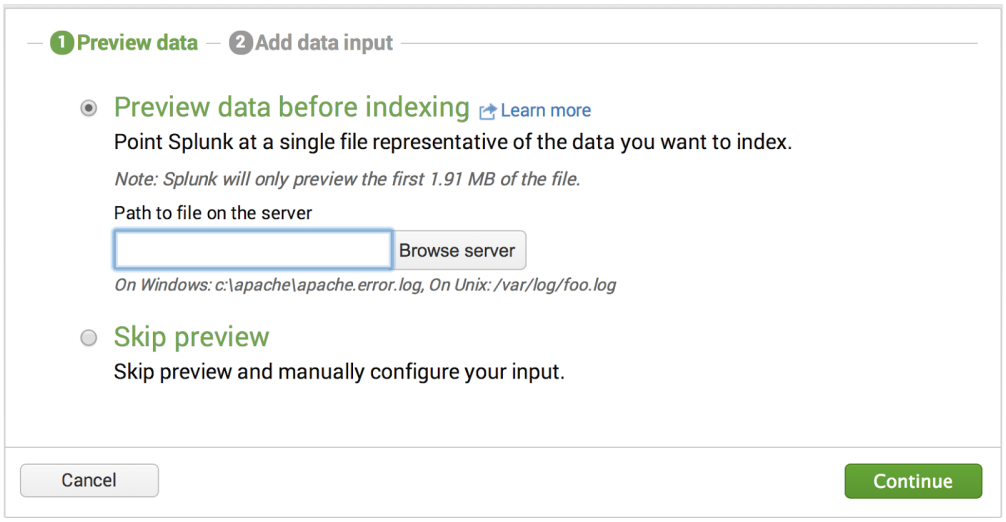


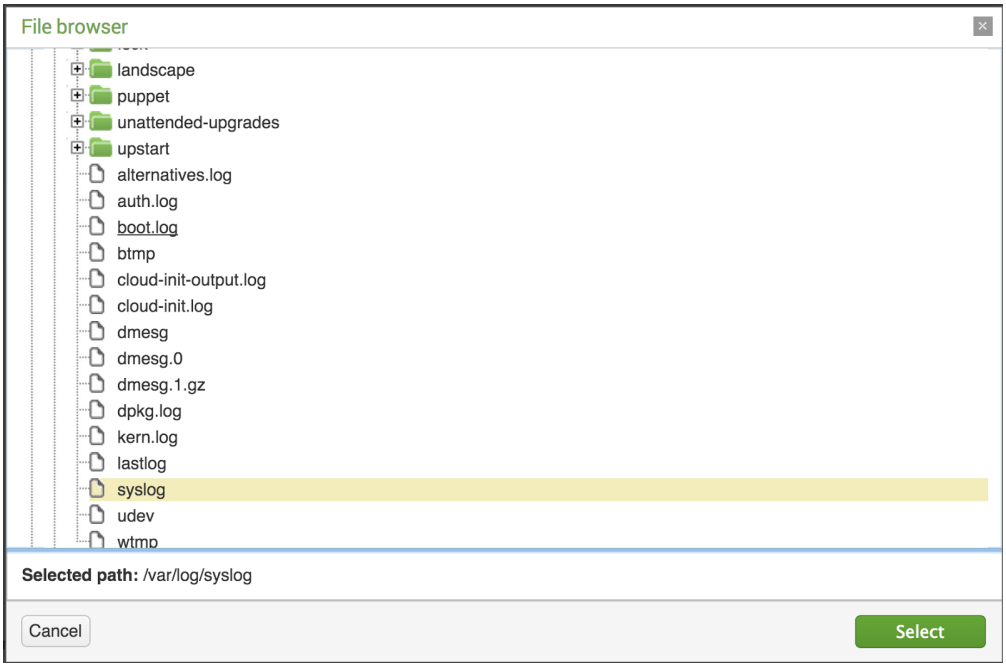
Configure the data input

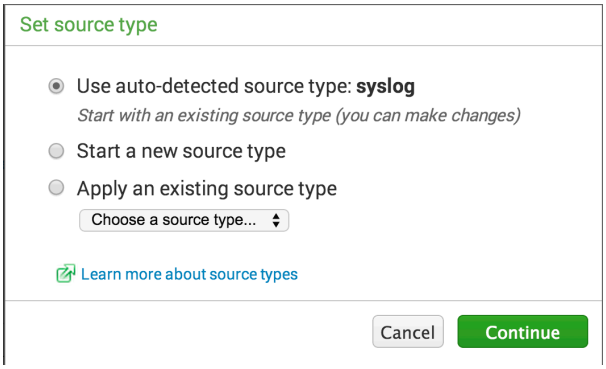
Splunk can receive the log data using either a direct file input or via a listening TCP or UDP port. It has a pre-built parser for standard logs like syslog. If we were logging from the Node application to a remote syslog file on the Splunk server itself, then we can simply setup the remote syslog file as the managed input on Splunk. The screenshots below depict a file system monitoring of syslog.

This workflow is triggered by selecting the “manage data input” menu item on the Splunk console.









Using the Universal Forwarder

Alternately you can use the Splunk Universal forwarder to forward the logs from the server hosting the Node application to a remote Splunk server.

Download the Universal Forwarder and install the version for your operating system.

You can write a small shell script to configure the Universal Forwarder on the Node server as shown below. For example, if the Universal Forwarder is installed under`/Applications`, the script would look like:

```
$ sudo /Applications/SplunkForwarder/bin/splunk add forward-server 192.168.100.5:9997
$ sudo /Applications/SplunkForwarder/bin/splunk start
$ echo "User/pass: admin/changeme"
$ sudo /Applications/SplunkForwarder/bin/splunk add monitor /var/log/system.log
```

Executing this shell script creates the monitor for autoforwarding events in syslog to Splunk. This feature can be used to read any logs including the supervisor log which stores aggregated Node logs managed by StrongLoop Controller. You just have to make sure that we point at the path for the `supervisor.log` instead of `syslog`, if you prefer not logging to `syslog`.

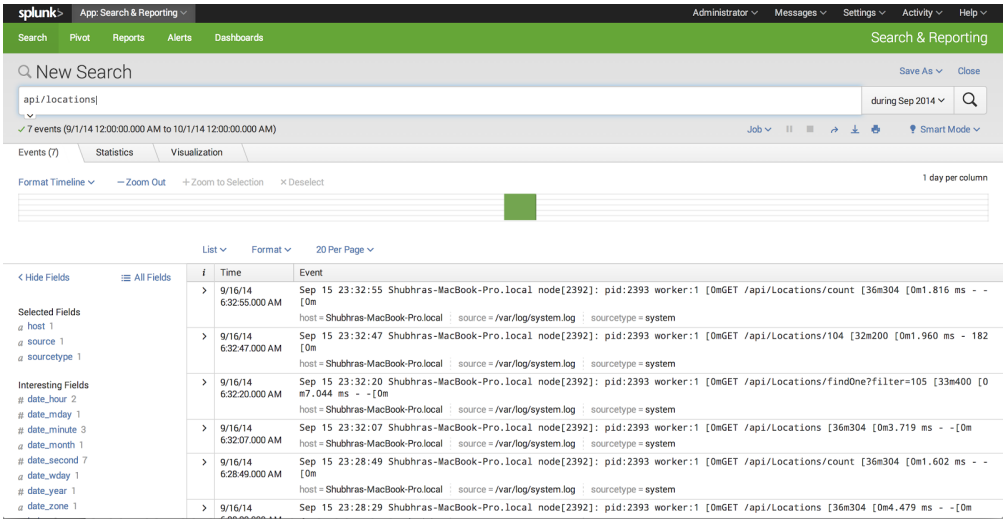
```
./splunk_fwd_setup_statsD.sh
..
..
Added monitor of '/var/log/system.log'.
```

This tells us that the monitor for reading the syslog has been setup successfully.

View and configure Node.js event data

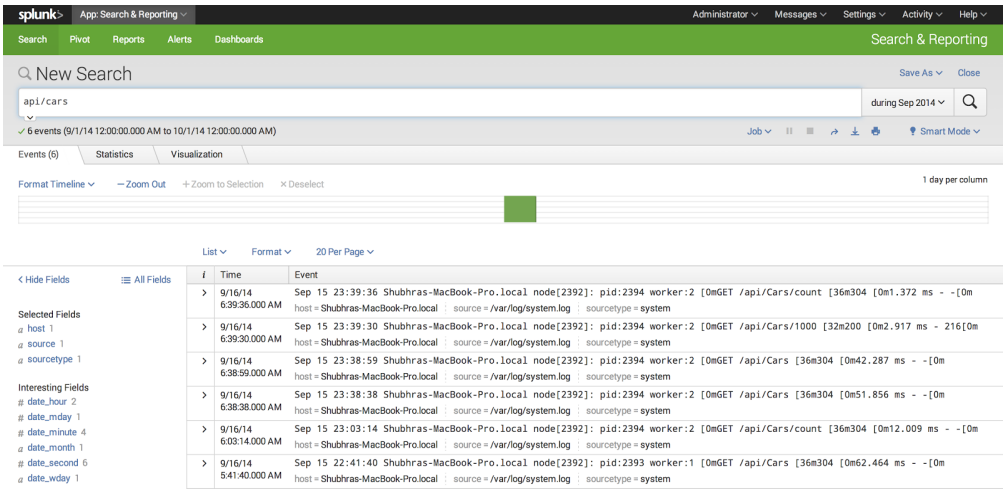
You can execute API calls from the API explorer of Loopback and then we can simply log in into Splunk and search for the API calls of interest

For example, to check all location-related API calls, search for “api/locations”…



You can see the timestamp, process ID, and worker ID as well as specific API endpoint calls along with response times, host and source. Some endpoints are simply GetCounts(), while others are findOne() for locations and so on.

Similarly search for “api/cars” to find all the APIs endpoints executed for the cars model along with associated metrics and `server/process/app` information.



Splunk has multiple ways of aggregating the event data reported into Dashboards and reports. For more information, see Splunk Dashboards and Visualization.

Monitoring app metrics

To generate and view metrics, you must have:

- A standard set of C++ compiler tools on the system running the application. See Installing compiler tools for more information.
- A valid StrongLoop license key on the system running the application.
  You automatically get a free trial license valid for 30-days from when you first run Arc that applies to apps running locally within Arc. See Managing your licenses for more information.

Then read the following to generate and view metrics:

- Setting up monitoring
- Viewing metrics with Arc
- Monitoring with slc
- Available metrics

Setting up monitoring

To generate and view metrics, you must have:

- A standard set of C++ compiler tools on the system running the application. See Installing compiler tools for more information.
- A valid StrongLoop license key on the system running the application.
  You automatically get a free trial license valid for 30-days from when you first run Arc that applies to apps running locally within Arc. See Managing your licenses for more information.

Prerequisites

Run Arc to get license key

You must run StrongLoop Arc at least once to get a license key. The first time you run Arc, it contacts the StrongLoop license server to request a free 30-day trial key.

You automatically get a free 30-day trial StrongLoop license key that starts the first time you run StrongLoop Arc. You can renew the trial license for certain features; see Renewing a license.

To manage your StrongLoop product licenses in Arc go to the **Licenses** page that shows the license keys for specific features.

For more information on licenses and subscriptions, see https://strongloop.com/node-js/subscription-plans/.

See Managing your licenses for more information.

Identify application name

If you created your application with `slc loopback`, this requirement will automatically be met.

The application name must be specified in one of the following ways:

- The application's `package.json` file **name** property. If you create your application with the `slc loopback` command, it creates this file and sets the application name appropriately.
- Environment variable STRONGLOOP_APPNAME.

The value of the environment variable supersedes the value set in `package.json`.

For example:

Excerpt from package.json

```
{  ...
 "name": "loopback-example-app",
 ...
}
```

Install compiler tools

To be able to view all metrics for an application (including memory and CPU profiling), you must have a small set of standard compiler tools on the machine running the application. See Installing compiler tools for more information.

If you don't install compiler tools, you will see warnings when you run the application.

Enable monitoring with third-party consoles

To enable monitoring with third-party consoles, either:

- Set the STRONGLOOP_METRICS environment variable to a StatsD, Graphite, or Splunk collector URL
- Install StrongLoop Process Manager as a service with `sl-pm-install --metrics <url>` where <url> is a StatsD, Graphite, or Splunk collector URL.

Then simply run your application in StrongLoop Process Manager or with Arc: To run a local application in StrongLoop PM, use `slc start`. For production, start PM as described in Setting up a production host and then deploy your application to it.

Viewing metrics with Arc

Prerequisites

See Setting up monitoring for important prerequisites.

Get a demo application

If you don't have your own application, download the profiler-demo-app application. It's designed to showcase StrongLoop metrics collection and CPU / heap profiling, and comes with a load generator script. Clone it from GitHub as follows:

```
$ git clone https://github.com/strongloop/profiler-demo-app.git
$ cd profiler-demo-app
$ npm install
```

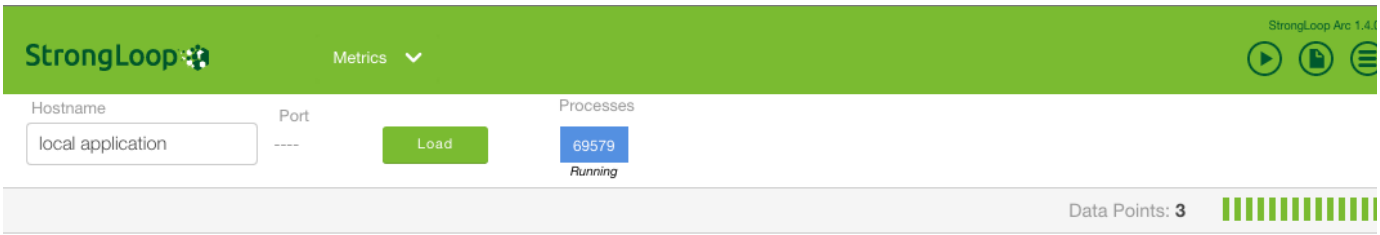
See Profiling for more information on the demo application.

View application metrics

By default, Arc starts a local instance of StrongLoop Process Manager, and displays "local application" in the Hostname field. It automatically displays the process IDs (PIDs) of the local application processes and select the first PID by default. Then, it displays metrics for that PID.

StrongLoop Arc Metrics provides live statistics on CPU use, event loop execution, and heap memory consumption for any Node application.

In the module picker, click **Metrics** to display the Metrics module:



To monitor an application running locally:

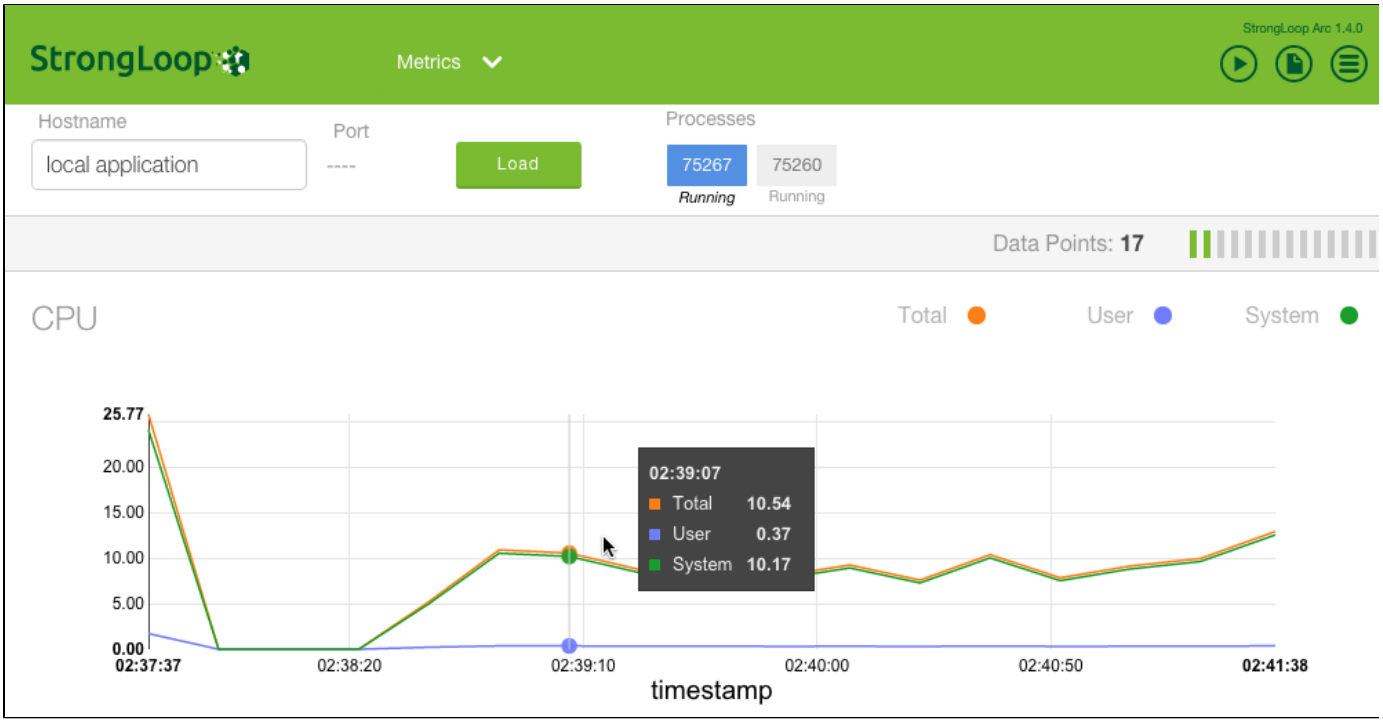
1. Run it with the app controller.
   Leave "local application" as the **Hostname** and **Port** value un-set.
2. Click **Load**.
   Arc will show the application process ID in a blue box.
3. Select a process ID (by default, the first one will be selected).
   Arc will start to display metrics graphs.

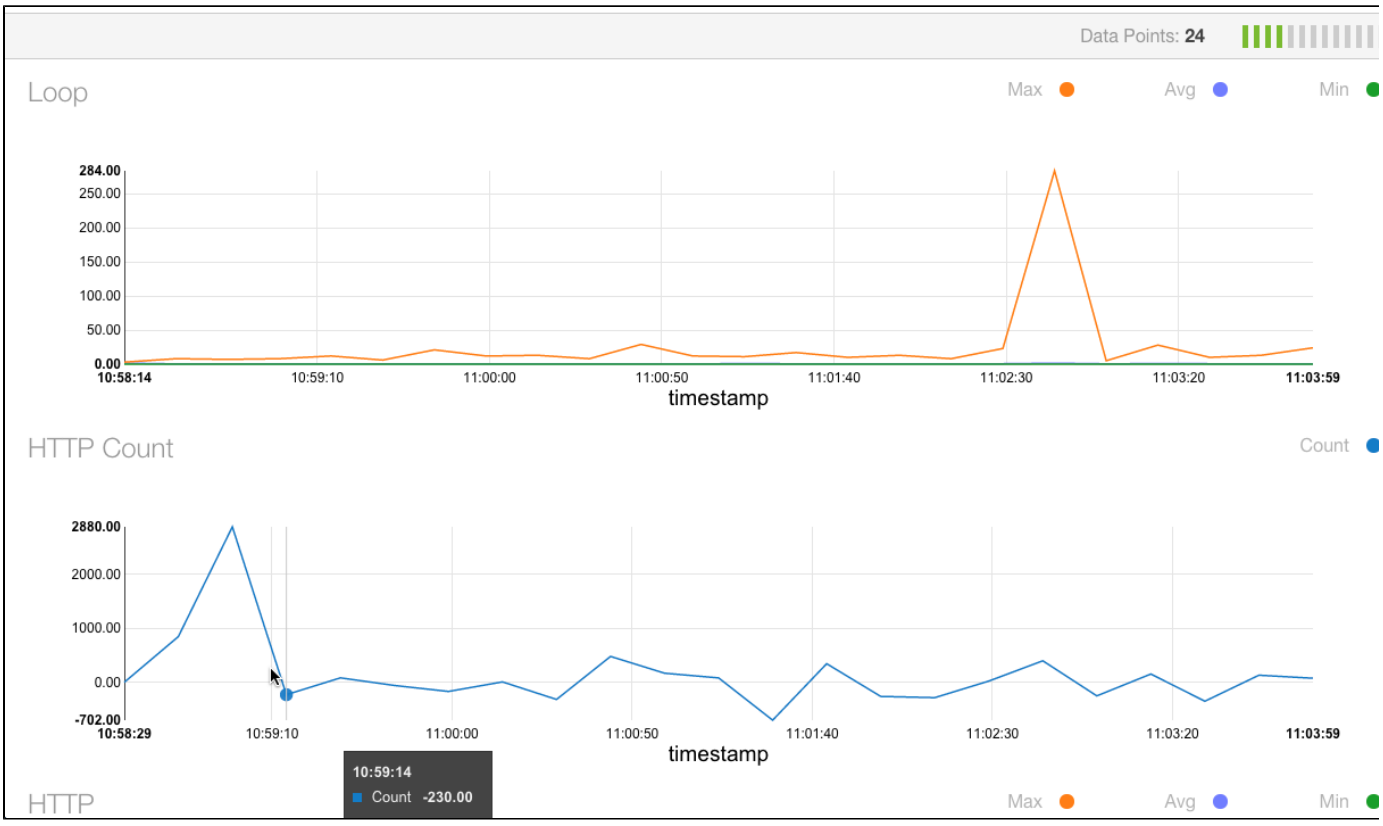
To monitor an application running remotely:

1. Rung StrongLoop Process Manager on the remote host. See Using Process Manager for more information.
2. Build and deploy an app to the remote host. See Building and deploying for more information.
3. Enter:
   - **Host name**: the name of the system where StrongLoop Process Manager is running the application; for example, `myhost.foo.com`.
   - **Port**: Enter the port on which StrongLoop Process Manager is running, by default 8701.
4. Click **Load**.
   Arc will show the application process IDs under the **Processes** heading.
5. Select a process ID (by default, the first one will be selected).
   Arc will start to display metrics graphs.

You can see a variety of metrics: CPU, Loop, HTTP Count, HTTP, Heap and Loop Count, as illustrated in the screenshots below.

See Available metrics below for a complete list.





Available metrics

Metric Name	Description
CPU metrics	
cpu.user	User CPU time. CPU time directly attributable to execution of the userspace process, expressed as a percentage of wall clock time. Can exceed 100% on multi-core systems.
cpu.total	Total CPU time. Sum of user and system time, expressed as a percentage of wall clock time. Can exceed 100% on multi-core systems.
cpu.system	System CPU time. CPU time spent inside the kernel on behalf of the process, expressed as a percentage of wall clock time. Can exceed 100% on multi-core systems. CPU time is elapsed wall clock time when the CPU is executing instructions on behalf of the process, that is, when the process or its corresponding kernel thread is actually running.
Event loop metrics	
loop.count	Number of event loop ticks in the last interval.
loop.average	Mean average tick time in milliseconds.

loop.minimum	Shortest (fastest) tick in milliseconds.
loop.maximum	Longest (slowest) tick in milliseconds.
Heap metrics	
heap.total	Total size of the V8 heap, expressed in bytes. The V8 heap stores JavaScript objects and values, excluding integers in the range -2,147,483,648 to 2,147,483,647. Note: the exact range depends on the processor architecture.
heap.used	The part of the V8 heap that is currently in use. Expressed in bytes. The V8 heap stores JavaScript objects and values, excluding integers in the range -2,147,483,648 to 2,147,483,647. Note: the exact range depends on the processor architecture.
gc.heap.used	The part of the V8 heap that is still in use after a minor or major garbage collector cycle, expressed in bytes. Strong agent reports this metric only when the garbage collector actually runs (periodically, as needed). The V8 heap stores JavaScript objects and values, excluding integers in the range -2,147,483,648 to 2,147,483,647. Note: the exact range depends on the processor architecture. *Not available in StrongLoop Arc*.
HTTP metrics	
http.connections.count	Number of new HTTP connections in the last interval. *Not available in StrongLoop Arc*.
Message queue metrics	
messages.in.count	Number of received strong-mq or axon messages. *Not available in StrongLoop Arc*.
messages.out.count	Number of sent strong-mq or axon messages. *Not available in StrongLoop Arc*.

Third-party module metrics

StrongLoop Process Manager collects metrics for a number of third-party modules, reported as:

- `<module>.count`
- `<module>.average`
- `<module>.maximum`
- `<module>.minimum`

The following table lists supported modules and recorded metrics:

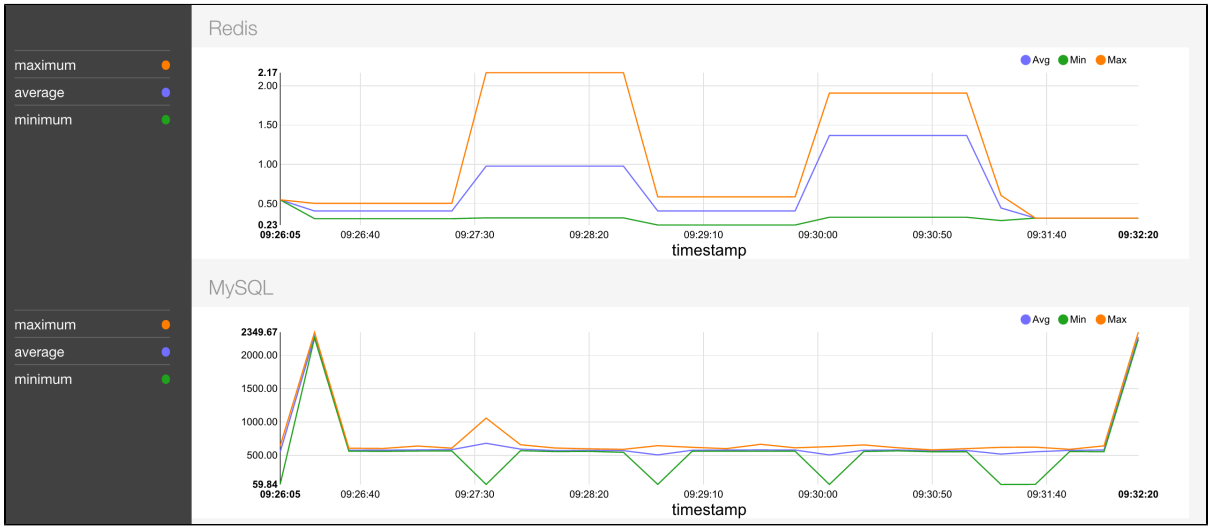
module	metric
loopback-datasource-juggler	Query time in ms (reported as `dao.<metric>`)
leveldown	Query time in ms
memcache	Query time in ms (reported as `memcached.<metric>`)
memcached	Query time in ms
mongodb	Query time in ms
mysql	Query time in ms
postgres	Query time in ms
redis	Query time in ms
riak-js	Query time in ms (reported as `riak.<metric>`)
oracle	Query time in ms (reported as `oracle.<metric>`)

Process Manager reports metrics once per time interval. For modules with more than one command or query method, PM reports the sum of query times for individual methods.

For example, the strong-oracle module has `.execute()`, `.commit()` and `.rollback()` methods. PM sums reponse times for those methods and reports them as one metric, where:

- `oracle.count` is the number of calls in the last interval
- `oracle.average` the mean average query response time
- `oracle.maximum` the slowest query response time
- `oracle.minimum` the fastest query response time

Examples:



Monitoring with slc

Overview

To generate and view metrics, you must have:

- A standard set of C++ compiler tools on the system running the application. See Installing compiler tools for more information.
- A valid StrongLoop license key on the system running the application.
  You automatically get a free trial license valid for 30-days from when you first run Arc that applies to apps running locally within Arc. See Managing your licenses for more information.

Setting the metrics URL

To set up metrics monitoring, specify a metrics URL, which can specify either a third-party console (StatsD, Graphite, or Splunk) or a log file. You can specify the metrics URL:

- With the STRONGLOOP_METRICS environment variable.
- When you install StrongLoop Process Manager as a service (for production use).
- With the `slc ctl env-set` command.

Setting the metrics URL with an environment variable

You can set the environment variable on the system running StrongLoop PM:

```
$ export STRONGLOOP_METRICS=<url>
```

Where `<url>` is one of the metrics URLs described below.

Then deploy the application to the system running StrongLoop PM.

Setting the metrics URL during Process Manager service installation

To set the metrics URL when you install StrongLoop Process Manager as a service, use the command:

```
$ sl-pm-install --metrics <url>
```

If installing with Docker, use:

```
$ curl -sL http://strong-pm.io/docker.sh | sudo bash -s STRONGLOOP_METRICS=<url>
```

Where `<url>` is one of the metrics URLs described below. See Setting up a production host for more information.

Setting the metrics URL at runtime

Once the application is already running or deployed, use the `slc ctl env-set` command to set the environment variable on the system running StrongLoop PM. StrongLoop PM will automatically restart the application with the new setting; for example, if the service name is my-app:

```
$ slc ctl env-set my-app STRONGLOOP_METRICS=<url>
```

Running the app

Once you have set the metrics URL, run the application under control of StrongLoop Process Manager.

For example, to run a local application called my-app:

```
$ cd my-app
$ slc start
```

To run a remote application (typical in production):

1. Build the application with `slc build`; see Building applications with slc.
2. Install StrongLoop Process Manager as a service on the production system; see Setting up a production host.
   If you have not already set the STRONGLOOP_METRICS environment variable, you can set it as described above in Setting the metrics URL during Process Manager service installation
3. Deploy the application to the production host with `slc deploy`; see Deploying apps to Process Manager.

Metrics URLs

Metrics URLs can take the form of a StatsD, Graphite, or Splunk URL or a log file specification.

StatsD

StatsD is a simple protocol for log information along with a simple daemon (server) that aggregates and summarizes application metrics. The client communicates with the StatsD server using the StatsD protocol, and the daemon then generates aggregate metrics and relays them to a graphing or monitoring backend. For more information on StatsD, see StatsD, what it is and how it can help you.

The StatsD Node server:

- Includes a built-in Graphite backend that can be local or hosted.
- Supports other backends such Zabbix, DataDog, and so on.
- Supports custom backends.

Other metrics consumers, such as DataDog, have agents that support the StatsD-protocol.

To use StatsD, set the STRONGLOOP_METRICS environment variable to a StatsD receiver URL of the form:

statsd://[*host*[:*port*]][/*scope*]

Where:

- *host* is the name of the host where the StatsD server is running; default is "localhost".
- *port* is the TCP port the StatsD server is using; default is 8125.
- *scope* is a string to scope or identify metrics; for example this might be the name of the application or module.

You can set the environment variable before you deploy your application to Process Manager.

To set it after you deploy the application, use the `slc ctl env-set` command, which will automatically restart the application with that environment variable. For example:

```
$ slc ctl env-set my-app STRONGLOOP_METRICS=statsd://localhost:8125/app1
```

Example output:

```
my-app.cpu.user:0.00907|g
my-app.cpu.system:0.01664|g
my-app.cpu.total:0.0257|g
my-app.heap.used:10698193|g
my-app.heap.total:27423366|g
my-app.loop.count:127|c
```

Hosted Graphite

To use Graphite, set a metrics URL of the form:

graphite://[*host*[:*port*]]

Where:

- *host* is the name of the host where the Graphite server is running; default is "localhost".
- *port* is the TCP port the Graphite server is using; default is 2003.

This specifies to forward metrics data to hosted Graphite.

For example:

```
$ slc ctl env-set STRONGLOOP_METRICS=graphite://myhost.foo.com:1234
```

Splunk

To use Splunk, set a metrics URL of the form:

splunk://[*host]*:*port*

Where:

- *host* is the name of the host where the Splunk server is running; default is "localhost".
- *port* is the TCP port the Splunk server is using; you must provide a value since the protocol has no assigned port.

This specifies to write to Splunk using a UDP key-value protocol.

For example:

```
$ slc ctl env-set my-app STRONGLOOP_METRICS=splunk://myhost.foo.com:1234
```

Log file

To send metrics information to a log file, set a metrics URL of the form:

```
log:[file]
```

If you omit the file name, by default metrics information is send to the console (stdout).

For example:

```
$ slc ctl env-set my-app STRONGLOOP_METRICS=log:myapp.log
```

Syslog

To write metrics information using syslog, set a metrics URL of the form:

```
syslog:[?[application=appname][&priority=level]]
```

Where:

- *appName* is any string; default is "statsd".
- *priority* is any of the following values:
  - LOG_DEBUG
  - LOG_INFO (the default)
  - LOG_NOTICE
  - LOG_WARNING
  - LOG_CRIT

For example:

```
$ slc ctl env-set my-app
STRONGLOOP_METRICS=syslog:[?[application=<application>][&priority=<priority>]]
```

Object tracking

StrongLoop Process Manager can track the creation and reclamation of JavaScript objects over time. Tracking objects helps to hunt down memory leaks, but since it imposes some CPU and memory overhead, it's disabled by default.

Use the `slc ctl objects-start` command to start tracking objects and `slc ctl objects-stop` to stop tracking; see slc ctl for more information.

Running your application

Start by running your application under control of StrongLoop Process Manager; for example, to run locally:

```
$ slc start
```

You can also deploy your application to a remote Process Manager using `slc deploy`. and then use the `-c` option to slc ctl to specify the remote Process Manager URL.

Then display the PIDs and worker IDs for all worker processes:

```
$ slc ctl
```

For example:

```
Service ID: 1
Service Name: express-example-app
Environment variables:
  No environment variables defined
Instances:
    Version  Agent version  Cluster size
     4.1.0       1.5.1            4
Processes:
        ID      PID   WID  Listening Ports  Tracking objects?  CPU profiling?
    1.1.49401  49401   0
    1.1.49408  49408   1     0.0.0.0:3001
    1.1.49409  49409   2     0.0.0.0:3001
    1.1.49410  49410   3     0.0.0.0:3001
    1.1.49411  49411   4     0.0.0.0:3001
```

Setting metrics URL

Object tracking information is published to the metrics URL defined for the Process Manager. See Setting the metrics URL for more information.

For example, to set the metrics URL to a log file named `express-example.log`:

```
$ slc ctl env-set my-app STRONGLOOP_METRICS=log:express-example.log
```

```
$ slc ctl
Service ID: 1
Service Name: express-example-app
Environment variables:
    Name                Value
    STRONGLOOP_METRICS  log:myapp.log
Instances:
    Version  Agent version  Cluster size
     4.1.0       1.5.1            4
Processes:
        ID      PID   WID  Listening Ports  Tracking objects?  CPU profiling?
    1.1.49401  49401   0
    1.1.49408  49408   1     0.0.0.0:3001
    1.1.49409  49409   2     0.0.0.0:3001
    1.1.49410  49410   3     0.0.0.0:3001
    1.1.49411  49411   4     0.0.0.0:3001
```

Starting object tracking

You can track objects for any of these processes. For example, to track objects for the worker ID 1:

```
$ slc ctl objects-start 1.1.1
```

Let it run for a while under typical load to get object tracking data.

When enabled, StrongLoop agent reports the following metrics at 15 second intervals:

- `object.<type>.count`: Number of objects created and reclaimed in the last interval, where *<type>* is the name of the constructor that created the object, for example `Array`, `Date`. If the count is less than zero, it means more objects have been reclaimed than created.

- `object.<type>.size`: Total size of the objects created and reclaimed in the last interval, where *<type>* is the name of the constructor that created the object, for example `Array`, `Date`. If the size is less than zero, it means more objects have been reclaimed than created.

Stopping object tracking

Then to stop object tracking, use the command:

```
$ slc ctl objects-stop 1.1.1
```

Example output:

objects.log Expand source

```
2015-05-08T22:42:06.000Z statsd.bad_lines_seen=0 (count)
2015-05-08T22:42:06.000Z statsd.packets_received=10 (count)
2015-05-08T22:42:06.000Z express-example-app.Rands-MacBook-Air.0.loop.count=39 (count)
2015-05-08T22:42:06.000Z express-example-app.Rands-MacBook-Air.0.loop.minimum=0
(gauge)
2015-05-08T22:42:06.000Z express-example-app.Rands-MacBook-Air.0.loop.maximum=14
(gauge)
2015-05-08T22:42:06.000Z express-example-app.Rands-MacBook-Air.0.loop.average=0.94872
(gauge)
2015-05-08T22:42:06.000Z express-example-app.Rands-MacBook-Air.0.gc.heap.used=9198268
(gauge)
2015-05-08T22:42:06.000Z express-example-app.Rands-MacBook-Air.0.heap.used=10450150
(gauge)
2015-05-08T22:42:06.000Z express-example-app.Rands-MacBook-Air.0.heap.total=35944581
(gauge)
2015-05-08T22:42:06.000Z express-example-app.Rands-MacBook-Air.0.cpu.total=0.40734
(gauge)
2015-05-08T22:42:06.000Z express-example-app.Rands-MacBook-Air.0.cpu.system=0.32576
(gauge)
2015-05-08T22:42:06.000Z express-example-app.Rands-MacBook-Air.0.cpu.user=0.08158
(gauge)
2015-05-08T22:42:21.000Z statsd.bad_lines_seen=0 (count)
2015-05-08T22:42:21.000Z statsd.packets_received=49 (count)
2015-05-08T22:42:21.000Z express-example-app.Rands-MacBook-Air.0.loop.count=46 (count)
2015-05-08T22:42:21.000Z express-example-app.Rands-MacBook-Air.1.loop.count=2 (count)
2015-05-08T22:42:21.000Z express-example-app.Rands-MacBook-Air.2.loop.count=2 (count)
2015-05-08T22:42:21.000Z express-example-app.Rands-MacBook-Air.3.loop.count=2 (count)
2015-05-08T22:42:21.000Z express-example-app.Rands-MacBook-Air.4.loop.count=2 (count)
2015-05-08T22:42:21.000Z express-example-app.Rands-MacBook-Air.0.loop.minimum=0
(gauge)
2015-05-08T22:42:21.000Z express-example-app.Rands-MacBook-Air.0.loop.maximum=1
(gauge)
2015-05-08T22:42:21.000Z express-example-app.Rands-MacBook-Air.0.loop.average=0.13043
(gauge)
2015-05-08T22:42:21.000Z express-example-app.Rands-MacBook-Air.0.gc.heap.used=9198268
(gauge)
2015-05-08T22:42:21.000Z express-example-app.Rands-MacBook-Air.0.heap.used=10796490
(gauge)
2015-05-08T22:42:21.000Z express-example-app.Rands-MacBook-Air.0.heap.total=36060545
(gauge)
2015-05-08T22:42:21.000Z express-example-app.Rands-MacBook-Air.0.cpu.total=0.11375
(gauge)
2015-05-08T22:42:21.000Z express-example-app.Rands-MacBook-Air.0.cpu.system=0.09507
(gauge)
```

```
2015-05-08T22:42:21.000Z express-example-app.Rands-MacBook-Air.0.cpu.user=0.01868
(gauge)
2015-05-08T22:42:21.000Z express-example-app.Rands-MacBook-Air.1.loop.minimum=0
(gauge)
2015-05-08T22:42:21.000Z express-example-app.Rands-MacBook-Air.1.loop.maximum=18
(gauge)
2015-05-08T22:42:21.000Z express-example-app.Rands-MacBook-Air.1.loop.average=9
(gauge)
2015-05-08T22:42:21.000Z express-example-app.Rands-MacBook-Air.1.gc.heap.used=8881704
(gauge)
2015-05-08T22:42:21.000Z express-example-app.Rands-MacBook-Air.1.heap.used=12943437
(gauge)
2015-05-08T22:42:21.000Z express-example-app.Rands-MacBook-Air.1.heap.total=34411277
(gauge)
2015-05-08T22:42:21.000Z express-example-app.Rands-MacBook-Air.1.cpu.total=1.44555
(gauge)
2015-05-08T22:42:21.000Z express-example-app.Rands-MacBook-Air.1.cpu.system=1.17609
(gauge)
2015-05-08T22:42:21.000Z express-example-app.Rands-MacBook-Air.1.cpu.user=0.26947
(gauge)
2015-05-08T22:42:21.000Z express-example-app.Rands-MacBook-Air.2.loop.minimum=0
(gauge)
2015-05-08T22:42:21.000Z express-example-app.Rands-MacBook-Air.2.loop.maximum=16
(gauge)
2015-05-08T22:42:21.000Z express-example-app.Rands-MacBook-Air.2.loop.average=8
(gauge)
2015-05-08T22:42:21.000Z express-example-app.Rands-MacBook-Air.2.gc.heap.used=8878365
(gauge)
2015-05-08T22:42:21.000Z express-example-app.Rands-MacBook-Air.2.heap.used=12939553
(gauge)
2015-05-08T22:42:21.000Z express-example-app.Rands-MacBook-Air.2.heap.total=34329673
(gauge)
2015-05-08T22:42:21.000Z express-example-app.Rands-MacBook-Air.2.cpu.total=1.24815
(gauge)
2015-05-08T22:42:21.000Z express-example-app.Rands-MacBook-Air.2.cpu.system=1.13068
(gauge)
2015-05-08T22:42:21.000Z express-example-app.Rands-MacBook-Air.2.cpu.user=0.11747
(gauge)
2015-05-08T22:42:21.000Z express-example-app.Rands-MacBook-Air.3.loop.minimum=0
(gauge)
2015-05-08T22:42:21.000Z express-example-app.Rands-MacBook-Air.3.loop.maximum=2
(gauge)
2015-05-08T22:42:21.000Z express-example-app.Rands-MacBook-Air.3.loop.average=1
(gauge)
2015-05-08T22:42:21.000Z express-example-app.Rands-MacBook-Air.3.gc.heap.used=8875527
(gauge)
2015-05-08T22:42:21.000Z express-example-app.Rands-MacBook-Air.3.heap.used=12939427
(gauge)
2015-05-08T22:42:21.000Z express-example-app.Rands-MacBook-Air.3.heap.total=34200824
(gauge)
2015-05-08T22:42:21.000Z express-example-app.Rands-MacBook-Air.3.cpu.total=1.04327
(gauge)
2015-05-08T22:42:21.000Z express-example-app.Rands-MacBook-Air.3.cpu.system=0.94033
(gauge)
2015-05-08T22:42:21.000Z express-example-app.Rands-MacBook-Air.3.cpu.user=0.10295
(gauge)
2015-05-08T22:42:21.000Z express-example-app.Rands-MacBook-Air.4.loop.minimum=0
(gauge)
2015-05-08T22:42:21.000Z express-example-app.Rands-MacBook-Air.4.loop.maximum=1
```

```
(gauge)
2015-05-08T22:42:21.000Z express-example-app.Rands-MacBook-Air.4.loop.average=0.5
(gauge)
2015-05-08T22:42:21.000Z express-example-app.Rands-MacBook-Air.4.gc.heap.used=8608397
(gauge)
2015-05-08T22:42:21.000Z express-example-app.Rands-MacBook-Air.4.heap.used=12917944
(gauge)
2015-05-08T22:42:21.000Z express-example-app.Rands-MacBook-Air.4.heap.total=34127810
(gauge)
2015-05-08T22:42:21.000Z express-example-app.Rands-MacBook-Air.4.cpu.total=0.75981
(gauge)
2015-05-08T22:42:21.000Z express-example-app.Rands-MacBook-Air.4.cpu.system=0.68941
(gauge)
2015-05-08T22:42:21.000Z express-example-app.Rands-MacBook-Air.4.cpu.user=0.07039
(gauge)
2015-05-08T22:42:21.000Z statsd.timestamp_lag=0 (gauge)
```

```
2015-05-08T22:42:36.000Z statsd.bad_lines_seen=0 (count)
2015-05-08T22:42:36.000Z statsd.packets_received=56 (count)
...
```

Monitoring the Node event loop

In addition to monitoring event loop statistics continuously using the metrics API, you can also use *Smart Profiling* to monitor Node processes and automatically start CPU profiling when the event loop stalls.

Smart profiling is supported only for applications running on Linux x86_64 systems with StrongLoop Process Manager 5.0.1+.

You don't get a free 30-day trial license for Smart Profililng, as you do for other features. To try it out, please email **sales@strongloop.com**.

Use Smart Profiling to start CPU profiling automatically when the Node event loop stalls.

Simply add a *timeout* argument to the slc ctl command to specify the threshold; if the event loop stalls for longer than that time, then CPU profiling starts. CPU profiling stops when the event loop resumes execution.

For example, for an application with service ID one (1) running locally:

```
$ slc ctl cpu-start 1.1.76901 12
```

This command will start CPU profiling for worker process ID 76901 if the event loop on that process stalls for more than 12ms (12 milliseconds). CPU profiling stops when then event loop resumes execution, that is, when it "becomes un-stuck".

Here's an example of smart CPU profiling for an app with service ID one (1) running on a remote host:

```
$ slc ctl -C http://my.remote.host cpu-start 1.1.76901 12
```

For a sample application to demonstrate Smart Profiling in action, see https://github.com/strongloop/smartprofiling-example-app.

Defining custom metrics

Overview

You can define custom metrics and then "patch" them dynamically into a running Node application to view using the Metrics API.

You can define the following custom metrics:

- Counts - can be incremented and decremented, and require no context.
- Timers - have a start, and a stop, and require a unique context capable of having a timer property created.

Injecting custom metrics

Inject custom metrics with the command:

```
$ slc ctl patch <worker> <file>
```

<*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).

<*file*> is the patch file defining the custom metrics.

Patch files

Patch files are JSON descriptions of a patch set.

A single application can use multiple patches, and patch as many files as you want. You can patch the same file repeatedly.

A patchset is:

```
{
  FILESPEC: [
    PATCH,
    ...
  ],
  ...
}
```

The `FILESPEC` is a regular expression (regex) string that uniquely identifies a JavaScript file in the application. For example, `index.js` is unlikely to be unique, but `your-app/index.js` probably is. Note that the script names matched against are fully qualified when they were required (so lengthening the file path can always result in a unique spec), and short when builtin to node (`^util.js$`).

You can apply multiple patches:

```
{
  type: TYPE,
  line: LINE,
  metric: METRIC,
  [context: CONTEXT,]
}
```

`TYPE` is one of:

- increment: increment the metric count
- decrement: decrement the metric count
- timer-start: start a metric timer
- timer-stop: stop a metric timer

`LINE` is the line number (the first line is line number `1`) at which the metrics code will be inserted.

`METRIC` is a dot-separated metric name. It will have `custom.` prepended to avoid conflict with built-in metric names, and will have either `.count` or `.timer` appended (as appropriate), to indicate the type.

`CONTEXT` is a javascript expression that must result in an object that can have a timer property created on it.

Example

Requires Node v0.10.32 and MySQL driver 2.5.1.

With the following code in your application's main JavaScript file:

```
// save as `dyninst-metrics-example.js`
var http = require('http');

http.createServer(request).listen(process.env.PORT || 3000);

function request(req, res) {
  setTimeout(function() { // line 7
    res.end('OK\n'); // line 8
  }, Math.random() > 0.1 ? 10 : 100);
}
```

You can patch this code to keep a count of concurrent requests and a timer for each request by passing the following `patch.json` to the `slc ctl patch` command:

```
{
  "dyninst-metrics-example\\.js": [
    { "type": "increment", "line": 7, "metric": "get.delay" },
    { "type": "decrement", "line": 8, "metric": "get.delay" },
    { "type": "timer-start", "line": 7, "metric": "get.delay", "context": "res" },
    { "type": "timer-stop", "line": 8, "metric": "get.delay", "context": "res" }
  ]
}
```

To design your patch set, consider what the effect of the patch insertion will be. Patches must be lexically valid at the insertion point, which is always the beginning of the line. The above patch set will, conceptually, result in the patched file looking like:

```
// save as `dyninst-metrics-example.js`
var http = require('http');

http.createServer(request).listen(process.env.PORT || 3000);

function request(req, res) {
res.___timer = start('get.delay');increment('get.delay');  setTimeout(function() { //
line 7
res.___timer.stop();decrement('get.delay');   res.end('OK\n'); // line 8
  }, Math.random() > 0.1 ? 10 : 100);
}
```

Note from the above:

- Patches must by valid when inserted at the beginning of the specified line.
- Patches can be cumulative, and previous patches don't change the line numbering of the file.
- The context is used to store a timer on start, that can be accessed on stop. The context expression does not have to be identical for start and stop, but must evaluate to the same object.
- Duplicate stops as well as non-existence of a timer are ignored.
- Metrics can have the same name, since the type is appended.

Example

1. Clone loopback-example-app and install dependencies:

```
$ git clone https://github.com/strongloop/loopback-example-app
$ cd loopback-example-app
$ npm install
```

2. Enter these commands to start StrongLoop Process Manager, run the app under its control, and set the necessary environment variables:

```
$ slc start
$ slc ctl env-set loopback-example-app DB=mysql
PORT=8765 STRONGLOOP_METRICS=statsd://
```

3. Clone etsy/statsd.
4. Configure it to dump to console: https://github.com/strongloop-forks/statsd/blob/master/exampleConfig.js#L150
5. Run

```
$ node stats.js exampleConfig.js
```

6. Create `patch.json` with the contents below
7. Wait until some metrics are showing up on Statsd console, it should happen by now 8 hit the example explorer with some queries, note there are no custom metrics
8. Patch worker 1:

```
$ slc ctl patch 1.1.1 patch.json
```

Open the example explorer with some queries to see some custom metrics information.

The patch file `patch.json` should be:

```
{
  "^http\\.js$": [
    { "type": "increment",   "line": 2062, "metric": "http.req" },
    { "type": "decrement",   "line": 1013, "metric": "http.req" },
    { "type": "timer-start", "line": 2062, "metric": "http.req", "context": "res" },
    { "type": "timer-stop",  "line": 1013, "metric": "http.req", "context": "this" }
  ],
  "mysql\\/lib\\/Connection\\.js": [
    { "type": "timer-start", "line": 184, "metric": "mysql.query", "context": "query"
}
  ],
  "mysql\\/lib\\/protocol\\/sequences\\/Sequence\\.js": [
    { "type": "increment",   "line": 25, "metric": "mysql.req" },
    { "type": "decrement",   "line": 75, "metric": "mysql.req" },
    { "type": "timer-stop",  "line": 75, "metric": "mysql.query", "context": "this" }
  ]
}
```

Viewing metrics with DataDog

The StrongLoop monitoring agent exposes an API that enables you to view application performance metrics (APM) with an on-premises console. StrongLoop also provides middleware to publish metrics via the well-known StatsD protocol. Many metrics consumers, such as Datadog, have agents that support the StatsD protocol. This makes it easy to create your own custom dashboard in DataDog powered by the StrongLoop monitoring agent.

- Monitoring with StrongLoop and DataDog
- Available metrics
- Some useful DataDog dashboards
  - CPU consumption
  - Event loop
  - Heap use during garbage collection
  - HTTP connections

Monitoring with StrongLoop and DataDog

With the following commands you can send metrics to a StatsD compatible server:

```
$ cd <your-app-root>
$ slc start
$ slc ctl env-set STRONGLOOP_METRICS=statsd://[<host>][:<port>]
```

Then start your DataDog agent as you usually do, and then check the DataDog Integration dashboard for metrics from your Node app.

Available metrics

The StrongLoop agent sends real-time performance metrics for a Node application. Once you know what's being sent and how you can best utilize each of the metrics, you can group them into as many integration dashboards as you choose.

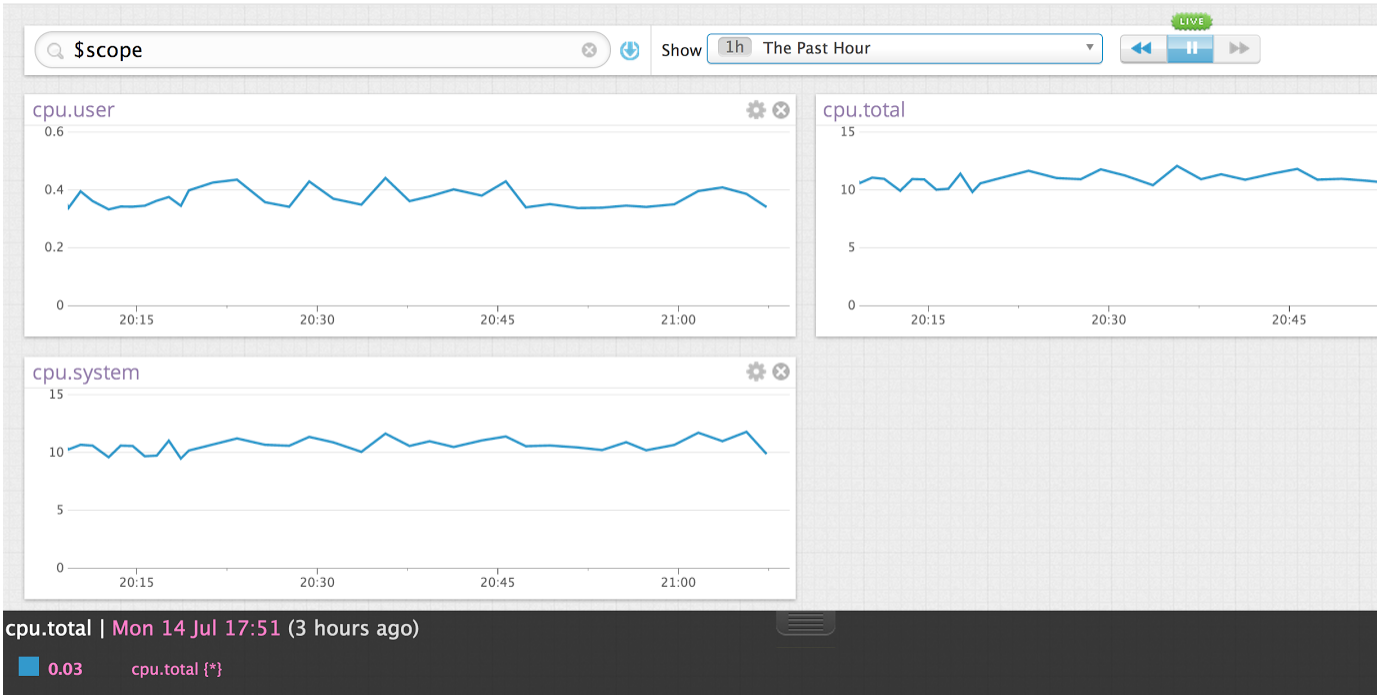
See Available metrics for the list of metrics you can monitor.

Some useful DataDog dashboards

Below are some examples of useful dashboards that you can create and view with DataDog.

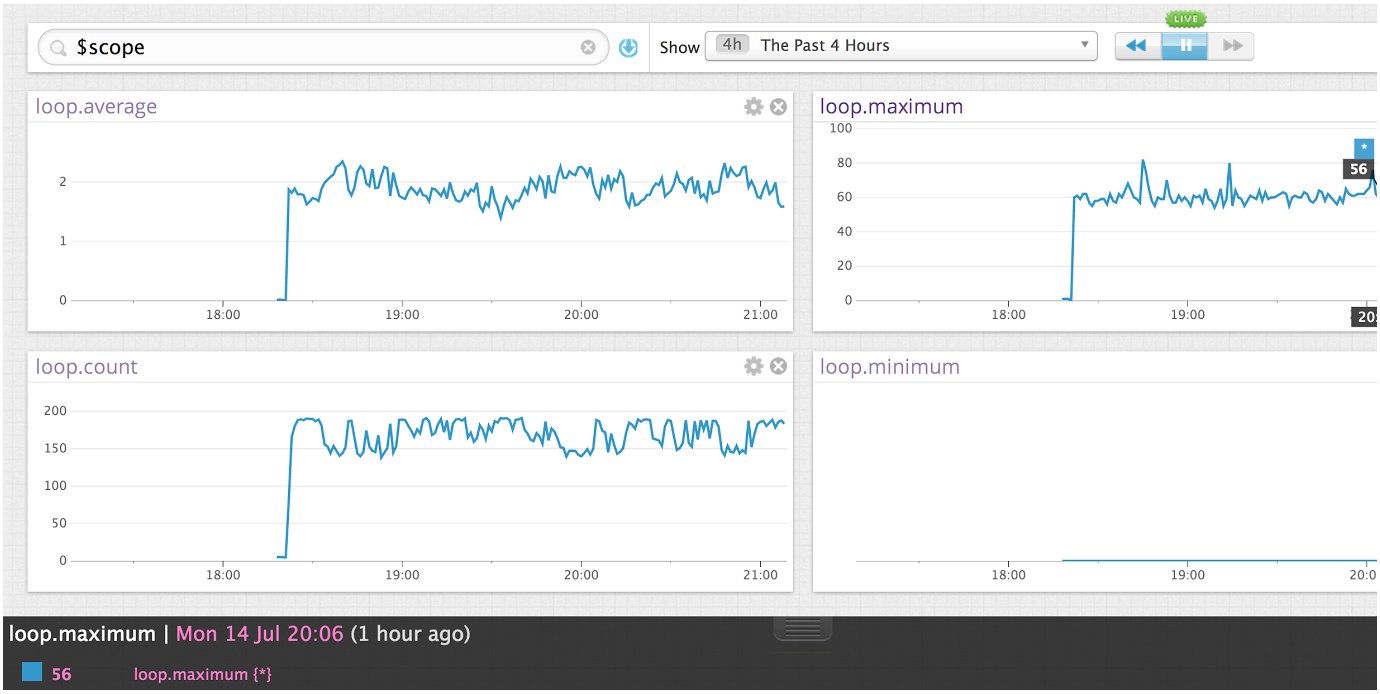
CPU consumption

The following dashboard is created from the cpu.user, cpu.total, and cpu.system metrics strings.



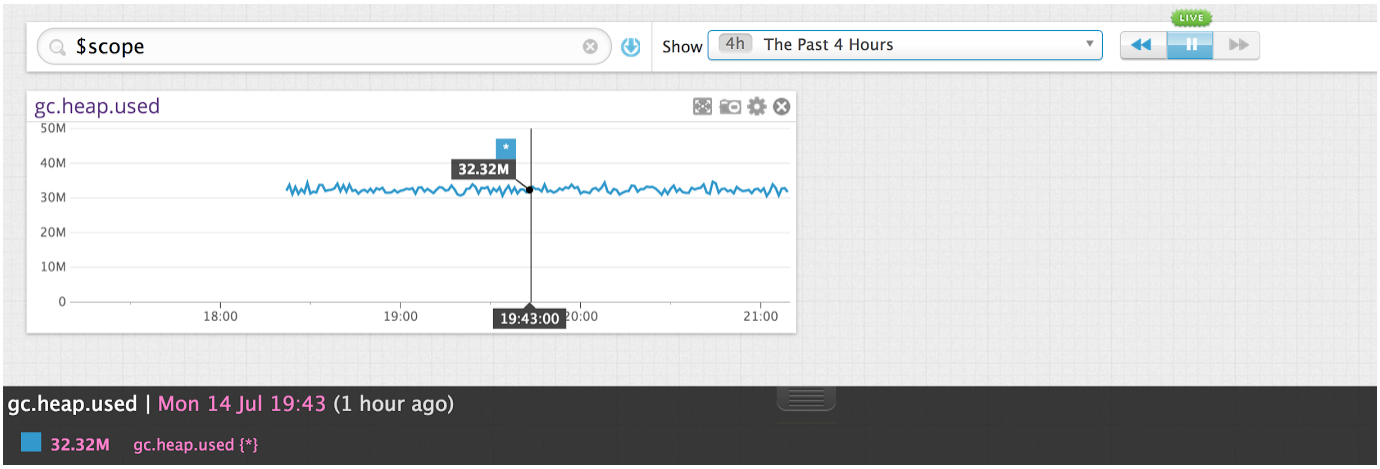
Event loop

The following dashboard is created with the loop.average, loop.maximum, loop.count, and loop.minimum metrics strings.



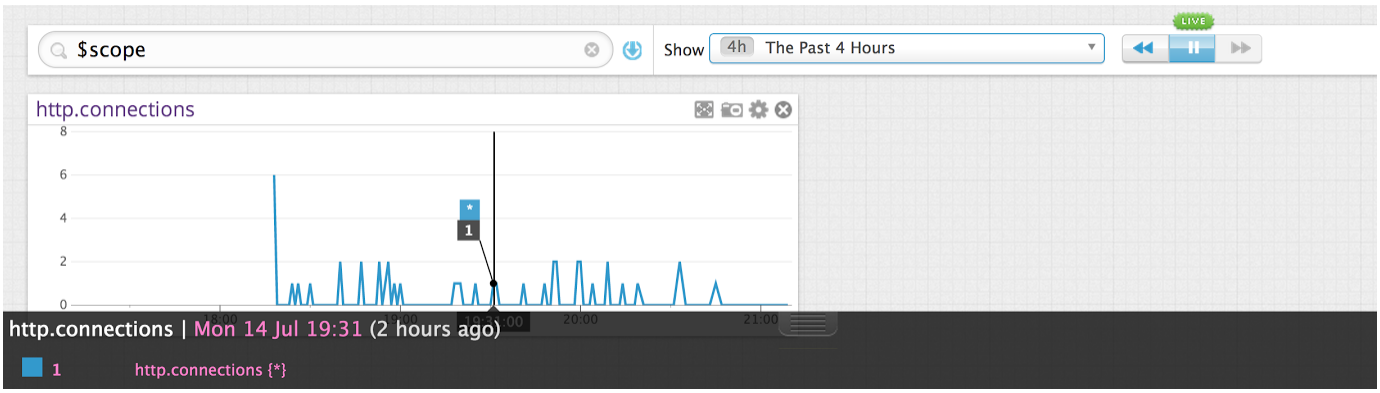
Heap use during garbage collection

The following dashboard is created with the gc.heap.used metrics string.



HTTP connections

The following dashboard is created with the http.connections metrics string.



Available metrics

Overview

StrongLoop Process Manager (PM) enables you to gather performance metrics for Node.js processes.

How to use

Follow Setting up monitoring to setup your application for monitoring.

Then, depending on whether you want to use Arc or a third-party console, follow the instructions in:

- Viewing metrics with Arc
- Monitoring with slc

Available metrics

Metric Name	Description
CPU metrics	
cpu.user	User CPU time. CPU time directly attributable to execution of the userspace process, expressed as a percentage of wall clock time. Can exceed 100% on multi-core systems.
cpu.total	Total CPU time. Sum of user and system time, expressed as a percentage of wall clock time. Can exceed 100% on multi-core systems.
cpu.system	System CPU time. CPU time spent inside the kernel on behalf of the process, expressed as a percentage of wall clock time. Can exceed 100% on multi-core systems. CPU time is elapsed wall clock time when the CPU is executing instructions on behalf of the process, that is, when the process or its corresponding kernel thread is actually running.
Event loop metrics	
loop.count	Number of event loop ticks in the last interval.
loop.average	Mean average tick time in milliseconds.
loop.minimum	Shortest (fastest) tick in milliseconds.
loop.maximum	Longest (slowest) tick in milliseconds.
Heap metrics	
heap.total	Total size of the V8 heap, expressed in bytes. The V8 heap stores JavaScript objects and values, excluding integers in the range -2,147,483,648 to 2,147,483,647. Note: the exact range depends on the processor architecture.
heap.used	The part of the V8 heap that is currently in use. Expressed in bytes. The V8 heap stores JavaScript objects and values, excluding integers in the range -2,147,483,648 to 2,147,483,647. Note: the exact range depends on the processor architecture.
gc.heap.used	The part of the V8 heap that is still in use after a minor or major garbage collector cycle, expressed in bytes. Strong agent reports this metric only when the garbage collector actually runs (periodically, as needed). The V8 heap stores JavaScript objects and values, excluding integers in the range -2,147,483,648 to 2,147,483,647. Note: the exact range depends on the processor architecture. *Not available in StrongLoop Arc*.

HTTP metrics	
http.connections.count	Number of new HTTP connections in the last interval. *Not available in StrongLoop Arc*.
Message queue metrics	
messages.in.count	Number of received strong-mq or axon messages. *Not available in StrongLoop Arc*.
messages.out.count	Number of sent strong-mq or axon messages. *Not available in StrongLoop Arc*.

Third-party module metrics

StrongLoop Process Manager collects metrics for a number of third-party modules, reported as:

- `<module>.count`
- `<module>.average`
- `<module>.maximum`
- `<module>.minimum`

The following table lists supported modules and recorded metrics:

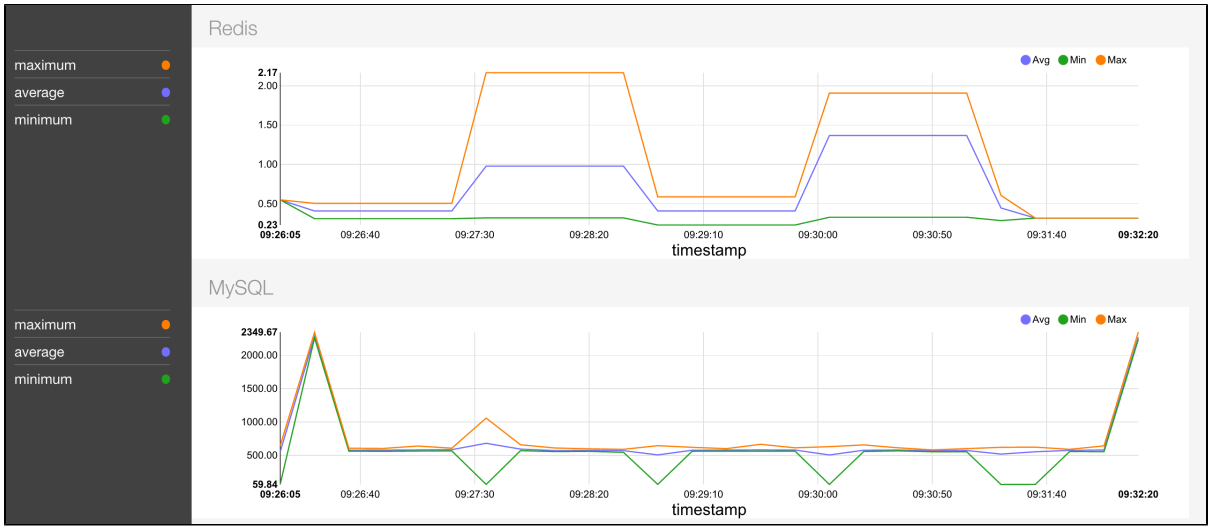
module	metric
loopback-datasource-juggler	Query time in ms (reported as `dao.<metric>`)
leveldown	Query time in ms
memcache	Query time in ms (reported as `memcached.<metric>`)
memcached	Query time in ms
mongodb	Query time in ms
mysql	Query time in ms
postgres	Query time in ms
redis	Query time in ms
riak-js	Query time in ms (reported as `riak.<metric>`)
oracle	Query time in ms (reported as `oracle.<metric>`)

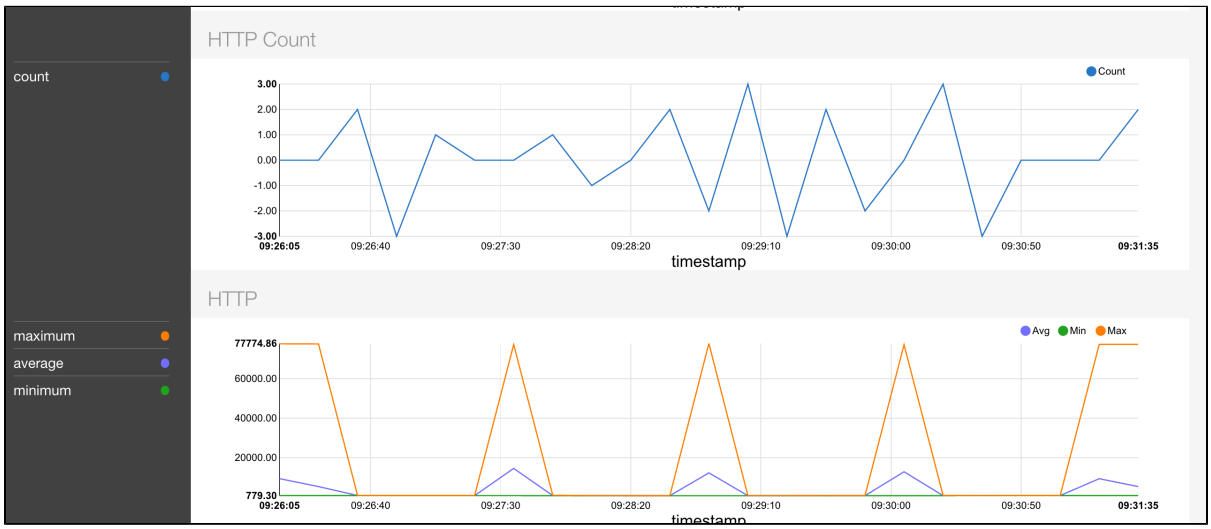
Process Manager reports metrics once per time interval. For modules with more than one command or query method, PM reports the sum of query times for individual methods.

For example, the strong-oracle module has `.execute()`, `.commit()` and `.rollback()` methods. PM sums reponse times for those methods and reports them as one metric, where:

- `oracle.count` is the number of calls in the last interval
- `oracle.average` the mean average query response time
- `oracle.maximum` the slowest query response time
- `oracle.minimum` the fastest query response time

Examples:





Tracing

Tracing is available as a beta feature.

Please contact **callback@us.ibm.com** to get a license for it.

Overview

The StrongLoop Arc **Tracing** module enables you to analyze performance and execution of Node applications to discover bottlenecks and trace code execution paths. You can display up to five hours of data to discover how applications perform over time.

See also: tracing-example-app.

You can drill down into specific function calls and execution paths for HTTP and database requests, to see how and where your application is spending time. Tracing provides powerful "flame graph" visualization of an application's function call stack and the corresponding execution times to help you track down where the application spends its time.

Tracing monitors an application by:

- Detecting the entry and exit points of JavaScript functions to record the elapsed time of each function call and the position of the function call in the source file. Tracing instruments every function in your application as well as the packages your application requires.
- Wrapping all HTTP/HTTPS requests and database operations for supported database backends.

A *trace* is the execution path that an application follows at runtime, including the entire function call stack.

The Tracing module displays data at several different levels:

- Initially, you see the **resource consumption timeline**, a chart of CPU use and memory consumption over time.
  Click anywhere on the chart to display information on the selected interval.
- You'll then see the **Trace sequences** view with detailed information on the selected time interval.
  Click on one of the trace sequences to drill down into its execution path.
- You'll then see **Waterfalls** and **Synchronous vs. asynchronous waterfall** views for that trace sequence.
- Finally, you'll see the **Synchronous code flame graph** and **Synchronous raw data**, powerful tools to enable you to pinpoint exactly where your app is spending its time.

Supported backends

The following table lists the backends for which tracing has been tested. It may work with other versions of the stated Node drivers, but has not been tested with them.

Backend	Node Driver
Memcache	memcache 0.3.0
Memcached	memcached 2.1.0
MongoDB	mongodb 1.x - 2.x
MySQL	mysql 2.9.0+
Oracle	oracle 0.3.8 oracledb 1.0.0
PostgreSQL	pg 4.4.1
Redis	redis 0.21.1
SQLite	sqlite3 3.0.10

Setup

Configure license

Contact StrongLoop for a license key to use tracing.

To view tracing data for an app running locally on the same system where you're running Arc, you don't need to do anything further.

To view tracing data for a remote application running on a remote server, you must push the license key to the remote server running Process Manager. For instructions, see Setting your license key on a remote host.

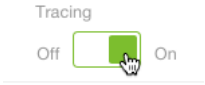
Procedure to view application tracing

Tracing does not currently work for applications run with Arc app controller. You *must* deploy an application to Process Manager (either locally or remotely) as described below.

To view tracing for an application:

1. Start StrongLoop Arc.
2. Start an instance of StrongLoop Process Manager (PM) either locally or on a remote server. For example, the easiest way to run an app locally is:

   ```
   $ cd my-app
   $ slc start
   ```

   For more information, see Using Process Manager.
3. If you ran Process Manager with slc start, as shown in the example in step 2, skip this step.
   Otherwise, for Process Manager on a remote server, deploy your application using either Arc or `slc`. For more information, see Building and deploying.
4. In Arc, choose **Process Manager** in the module selector and connect to the running PM instance as described in Connecting to Process Manager from Arc. For example, to connect to a PM running locally as shown in the example in step 2:
   - For **Strong PM**, enter `localhost` .
   - For **Port**, enter 8701.
5. In Arc, choose **Tracing**.
6. Click **On** to turn on tracing:
   

   You'll see a message like starting tracing... 0 of 4 processes.
   Once all processes are loaded, you will see tracing data. It may take a few seconds to load all the processes.

The **Host** drop-down selector lists Process Managers to which you've connected; by default the first one is selected. Click the **Host** drop-down selector to choose a different Process Manager.

By default, the first application process ID is selected. Click on another process ID to view data for it.

You will see the application resource consumption timeline.

Tracing information will not automatically update. Either refresh your browser or click on a new PID to display the latest data.

Resource consumption timeline

The information shown here is similar to Viewing CPU and heap memory consumption. However, tracing enables you to "drill down" at points where you see spikes in resource consumption to help uncover the root causes. When there is anomalous memory or CPU consumption, the chart will have an orange triangle along the timeline.

Tracing also enables you to view data over a much longer period of time (up to the last five hours of application execution) to track down issues that only occur with sustained application use.

This chart shows:

- **Blue line**: Total CPU use (standard Linux load average over 1 minute for all CPU cores).
- **Orange line**: Process memory consumption (in MBytes).

An orange triangle at the bottom of the chart indicates a statistical anomaly beyond a three-sigma variation.

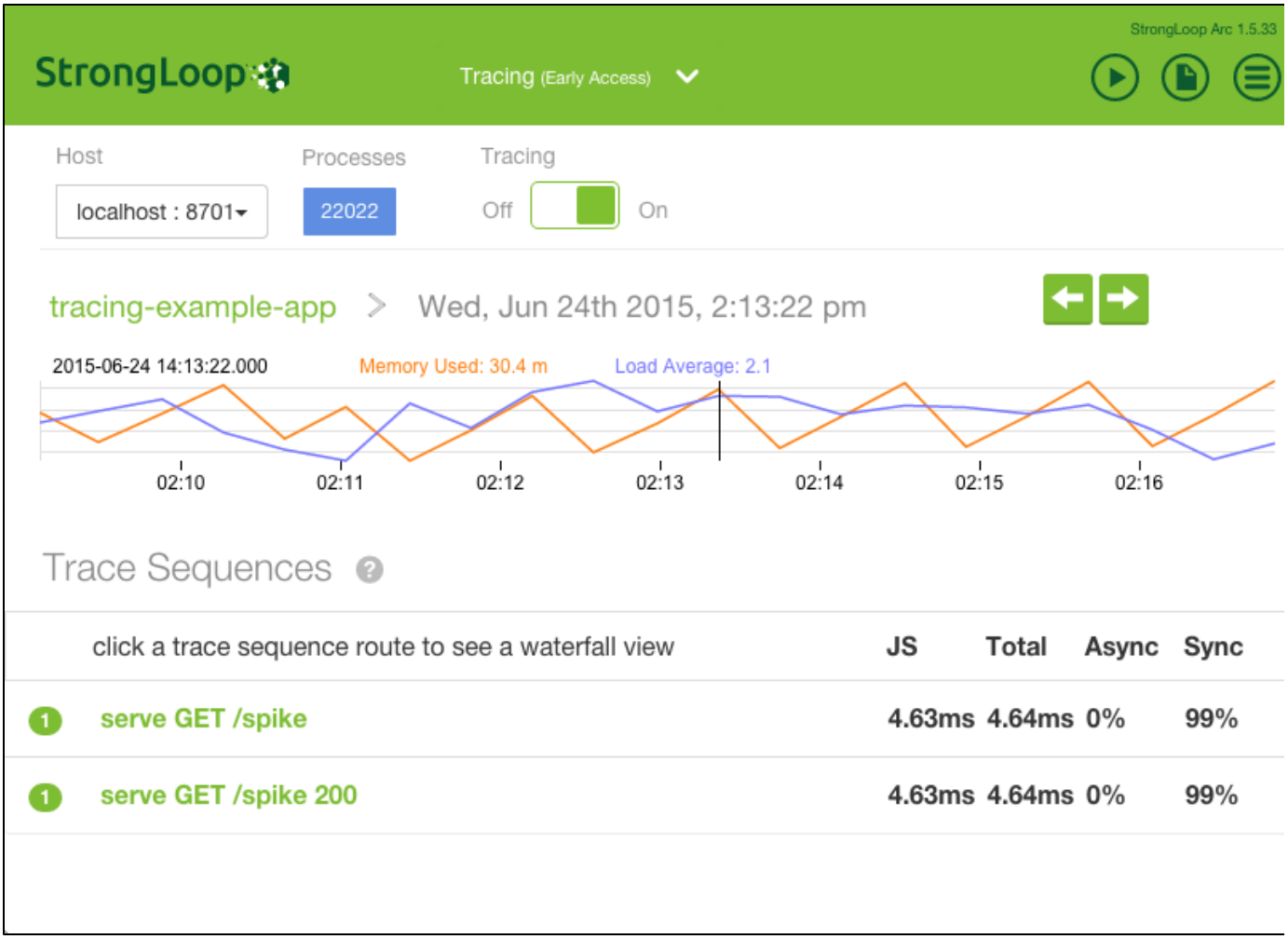
StrongLoop collects data at approximately 20s intervals for up to 5 hrs of time. Use this chart to pick a specific time interval you want to investigate.

Mouse over the chart to display the values at each point in time, and click on a point to display details of that interval.

Trace Sequences

See Release notes for important information about trace sequences for backend persistence stores.

Click a point on the resource consumption chart to display the trace sequences at that point; for example:



This table lists all the trace sequences active in the application during the selected interval, including HTTP endpoints, database operations, and other external calls such as memcache. The number in a green circle is the number of distinct code paths in the sequence.

Click on a row in the table to display details about that trace sequence.

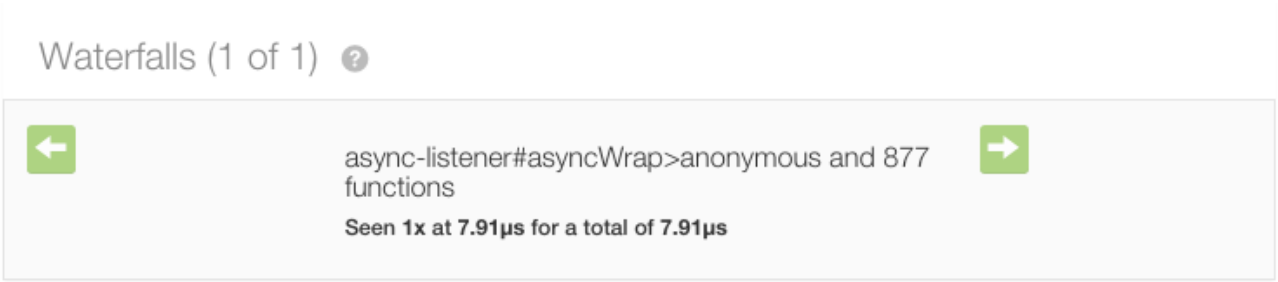
Click the green arrows at the upper right to move to the next and previous intervals.

The table displays for each trace sequence:

- **JS**: Time spent executing JavaScript code.
- **Total**: Time spent in this trace sequence.
- **Async**: Percentage of time spent waiting for a callback.
- **Sync**: Percentage of time spent executing synchronous code.

Click on one of the routes to display the code path executed when responding to HTTP requests to that route; for example:

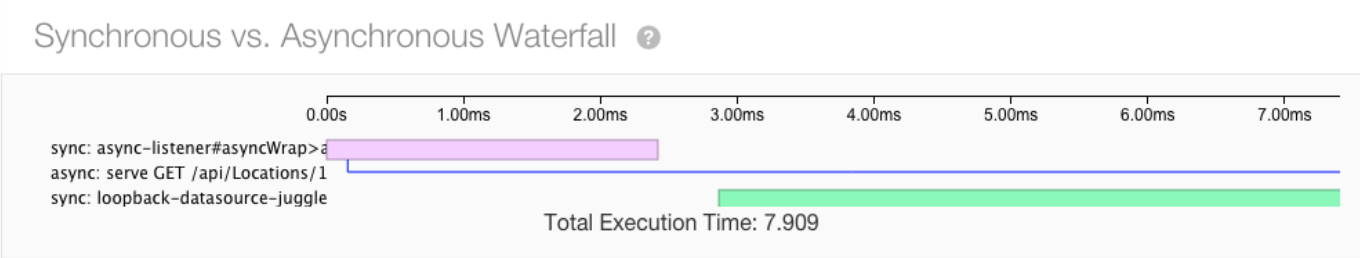
Waterfalls



Displays details for the trace sequence (HTTP route, database operation or other operation) and how much time the application spent in it.

Click the right arrow to display the next trace sequence. Click the left arrow to display the previous trace sequence. If there is only one execution path for the selected trace sequence, then the arrows are disabled.

Synchronous versus asynchronous waterfall



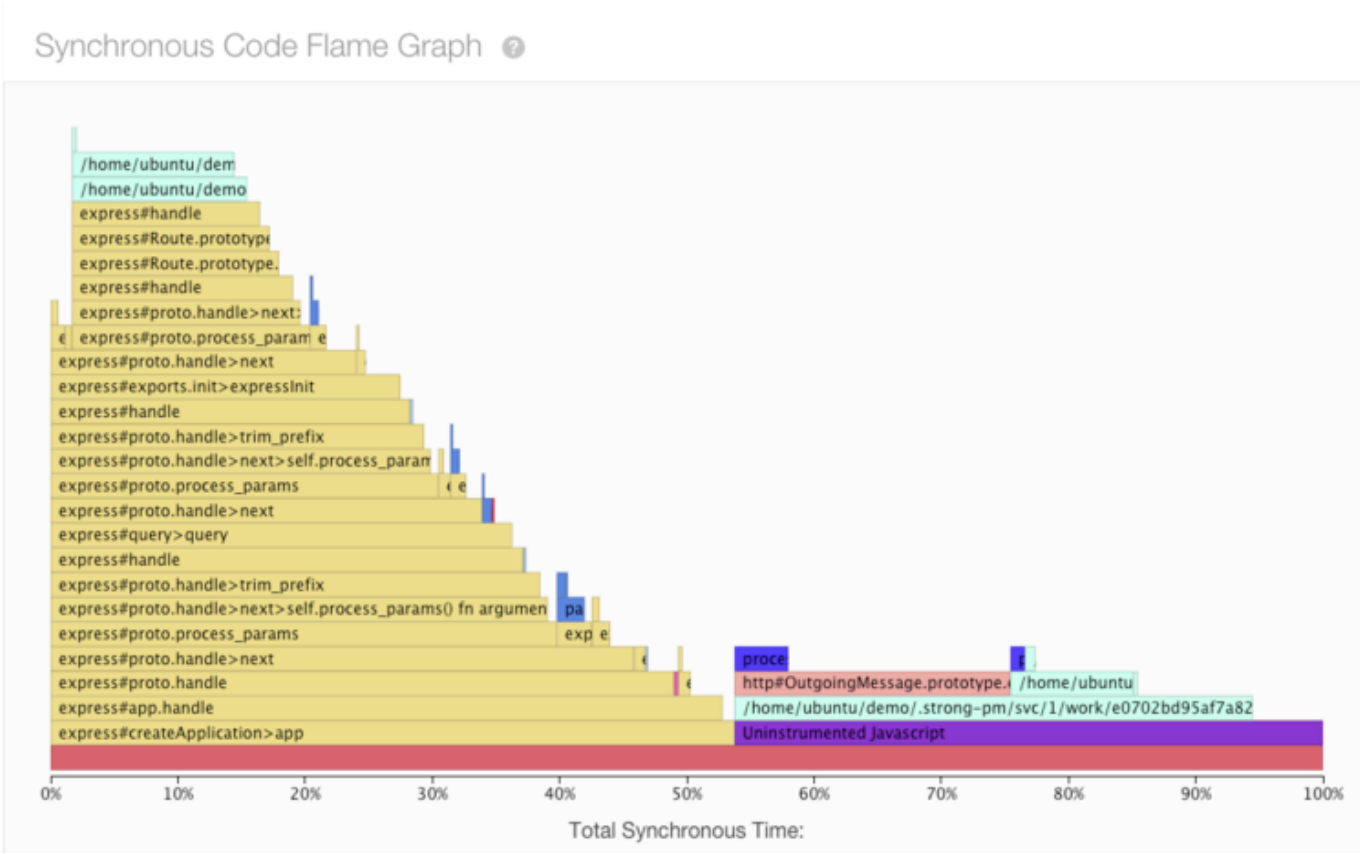
This section shows the elapsed time for the selected code path across asynchronous boundaries. Each continuous block of Javascript code is represented by a solid rectangle, whose length is proportional to the amount of time spent executing that code. The label of the block shows the most significant function in that sequence of code.

The top bar is the time spent initiating the original function call. The lower bar is the time spent executing the callback function, and the line connecting the two represents the time spent waiting for a response; for example, for a database server to return a value.

Asynchronous callbacks are represented by lines connecting the time of the callback being scheduled to when it was invoked.

Click the bar to display the top costs (in terms of time spent) in the Inspector.

Synchronous code flame graph



Provides rich information about the function call stack including application code and the third-party modules the code uses, enabling you to see where an app is spending the most time.

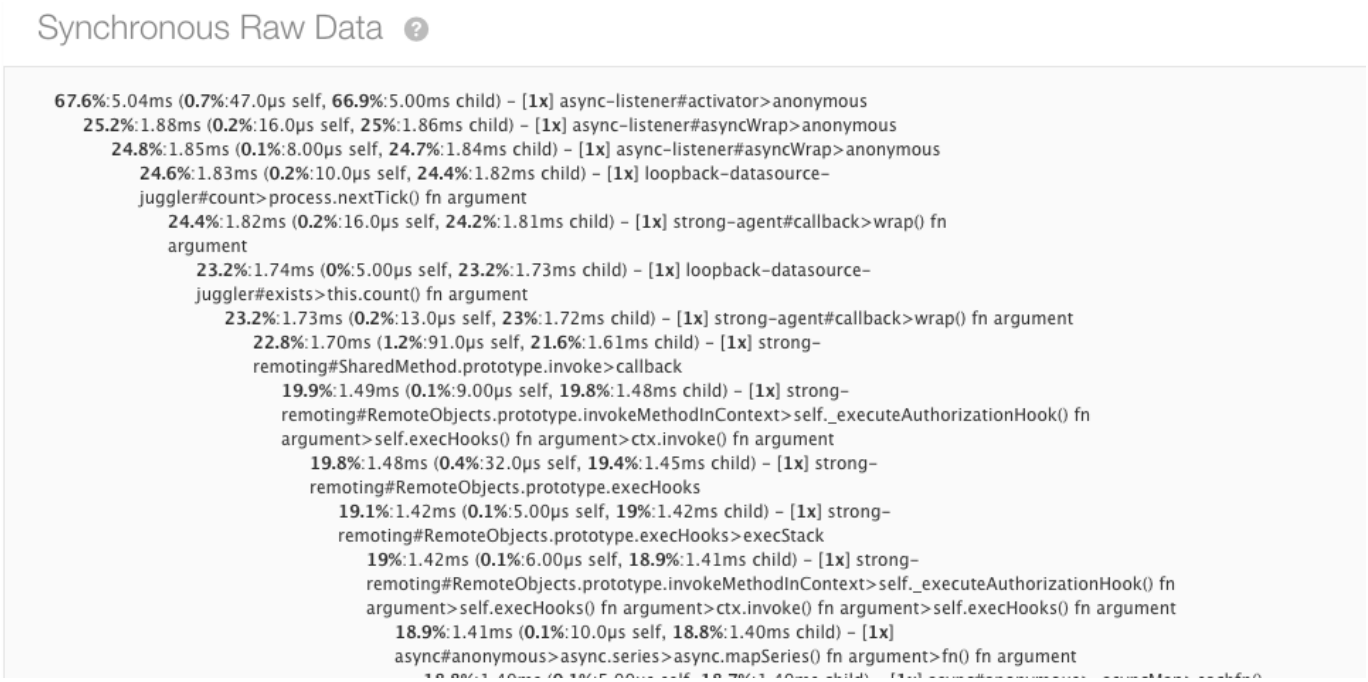
The flame graph shows each module in a different colored block. For example, `express` might be shown in green, `loopback` in purple, and `asy`

nc in purple. Blocks show function names when available. Unnamed callback functions are labeled “anonymous”. The selected call is always shown in yellow.

Read the flame graph from the bottom up. The bottom rectangle is the root of the call-stack tree, the application's initial function call, functions it calls are above that, and so on. The horizontal axis is percentage of time executing a function.

Mouse over a bar to display details of that particular call in the Inspector at right.

Synchronous raw data



This section shows the time spent executing the entire JavaScript call stack, displayed as a text-based tree illustrated graphically in the Flame Graph above.

Mouse over a bar to display details of that particular call in the Inspector at right. Click on a line to keep the display at that point.

Inspector

The Inspector displays different information, depending on whether you clicked on a bar in Synchronous vs. Asynchronous Waterfall or Synchronous Code Flame Graph / Raw Data.

Synchronous vs. Asynchronous Waterfall (Event Loop) view

Inspector shows:

- **Name**: Module and function with the largest self cost in this waterfall, that is, the function that has the longest time, not counting time to execute child functions.
- **Cost**: Execution time in microseconds.
- **Top Costs**: Percentage of time spent executing each child function.

Synchronous Code Flame Graph or Raw Data view

Inspector shows:

- **Name**: Module and function names.
- **Version**: Version of the module being used.
- **File Name**: Javascript source code file name where the function call occurs.
- **Line and Column**: Line and column in the source file where the function call occurs.
- **Total/self/child cost**: Time spent executing this call stack (total), the function (self), and functions it calls (child). In the flame graph, child calls are shown above the calling function (or "self" when you select that function call).
- **Total/self/child percentage**: Percentage of time spent this call stack (total), the function (self), and functions it calls (child).