Seneca

# Definitions

**Action (in Seneca):** function identified – and run - by a JSON object.

**Action pattern:** (see patterns). examples: ***'role:math,cmd:sum'*** ***{ role:'math', cmd:'sum'}***

**Errors (in Seneca)**: what qualifies as an error: *only* invalid input or internal failures. E.g. if a database connection fails, that's a Seneca error. However, if, say, a Redis database is queried and returns no data (on account of the key not existing), that is *not* an error – merely a fact about the database – which thus should not throw or pass an error.

**Extending patterns**: (see patterns). Defining ***seneca.add*** instances that match on more specific patterns than those already defined. E.g.  ***{ role: 'math', cmd: 'sum', integer:true }*** when ***{ role: 'math', cmd: 'sum' }*** already exists.

Advantage: no need to modify / break existing code. A seneca.act instance calling '***role:math,cmd:sum'*** will still trigger the seneca.add instance attached to that same action pattern. However, a seneca.act instance calling ***'role:math,cmd:sum,integer:true'*** will always run the seneca.add instance attached to that action pattern (never the '***role:math,cmd:sum'***, since it has fewer matching params).

**initialize plugin function: init:<moduleName>**:function with a special action pattern defined for a plugin, which automatically runs when the plugin is loaded, before any other action functions, and without being explicitly messaged. The action pattern takes the form ***init:moduleNam***. E.g.

**this.add('init:math', init);**

**function init(msg, respond){**

**// \*\* Do initialization things here \*\* //**

**respond();**

**}**

-- Note that if the init fn fails, the entire plugin will fail to load. Also, the init fn must complete before anything else in the plugin can run. Even asynchronous operations run from the init fn must finish before the plugin becomes accessible.

**jsonic**: abbreviated form of JSON that allows strings to be treated as JSON objects. E.g.

***{ role: 'math', cmd: 'sum }*** *-* in JSON is equivalent to - ***'role:math,cmd:sum'*** - in jsonic

**Messages (in Seneca)**: just JSON cdocuments, with any internal structure you like. Contains both the action pattern to determine what “seneca.add” definition handles the message, and any parameters sent to the handler. e.g.

***{ role: 'math', cmd: 'sum', left:1.5, right:2.5, integer:true }***

-- note that *all* key-value pairs are attached to seneca.add's msg parameter (as properties of the msg object). All are also used for pattern matching, but extras beyond the match can be used as the parameters on which the seneca.add-linked function (seneca.add's ***action*** parameter) can act.

**Microservice architecture**: a particular way of designing software applications as suites of independently deployable services

**msrv**: microservice

**Patterns:** used to identify what kinds of messages a microservice emits; as well as what kinds of messages something is interested in receiving from Seneca. When you specify property patterns that you care about, Seneca (with a little config help) ensures that you get any messages matching those patterns, sent by other services. Patterns are lists of key-value pairs that must be matched by the top-level properties of the JSON message document.

*Examples:* 1) ***{role: 'math', cmd: 'add'}*** 2) ***'role:math,cmd:add'*** [[[in jsonic]]]

Note that they are extendable – you can add similar patterns with addition (or reduced) functionality. E.g.

***'role:math,cmd:add,integer:true'***

-- 1st matching rule: the pattern w/ the most matching params always wins. Thus, the above takes precedence over ***'role:math,cmd:add***' – i.e. this pattern is the one that runs, and ***'role:math,cmd:add***' does not.

-- 2nd matching rule: if 2 patterns have the same # of properties, they are matched in alphabetical order. Thus, e.g. ***'a:1,b:2'*** wins over ***'a:1,c:3'*** since b comes before c. Uses [patrun](https://www.npmjs.com/package/patrun) module

-- 3rd matching rule: if 2 seneca.add instances match the same pattern, whichever is most recently defined in the control flow wins. *However*, this.prior can be used to access the instance defined in the control flow directly before the “primary” matching seneca.add instance.

-- Note: *all* action pattern params must be matched by a msg for the seneca.act -triggered fn linked to the action pattern to run; but not all params emitted by the msg must have a match

**Plugins (in Seneca)**: (also see seneca.use) minimalist notion of a 'plugin', intended to help w/ logging & debugging. A plugin is essentially just a collected set of action patterns which can be given a name to annotate logs. Plugins also take in an 'options' param used to 'instantiate' the plugin. ***seneca.use*** is used for plugin instantiation: the plugin name/path is passed in as the first param; the options object as the 2nd.

They also provide a mechanism for executing init fns in the correct order – e.g. if you want the db to connect before trying to get data from it, you can order these without depending on separate microservices being called in the correct order outside the module. Basic example:

**function basic\_plugin( options ) {**

**console.log(options)**

**}**

...then to 'include' it later (outside the module!)

**require('seneca').use( basic\_plugin, {foo:'bar'} ); //-->** instantiation fn emits **{foo: 'bar'}** if run (from console)

Note that you should not call seneca.act at all in the plugin definition, just seneca.add. Module files are meant to *define* plugins, not *use* plugins.

An init action fn can be used to init a plugin, in the form ***init:<plugin-name>****.* It must call its respond() fn. e.g.

**this.add( 'init:math', init );**

**function init( msg, respond ) {**

**fs.open( options.logfile, 'a', function( err, fd ) { // log to a custom file**

**if( err ) return respond( err ) // can't open for writing, so fail. This err is fatal to Seneca**

**log = make\_log(fd)**

**respond()**

**});**

**}**

**Pattern overrides**:

**seneca.\*\*.listen**: (see also seneca.client) launches a microservice process that listens on a given port (10101 by default) for http requests containing an action pattern matching one of the plugins (or simple action functions) hosted on the given seneca instance. Not a web server by default. Example:

**require( 'seneca' )()**

**.use( 'math' )**

**.listen();**

* henceforth, any action patterns in the math plugin can be requested via port 10101, or through ***seneca.client().act***.
* To see this in action, run this in the terminal:

***curl -d '{"role":"math","cmd":"sum","left":1,"right":2}' http://localhost:10101/act***

* + It should return a response in JSON form, such as ***{"answer":3}***

**seneca.act:** (see **seneca.add** first). Used to submit a message to act on. The first registered action pattern to match the message then runs (if there is a match). Takes 2 params: ***msg {Object||String}*** (the message object emitted by seneca.act), & ***response\_callback {Function}*** (receives the message response if there is one). ***response\_callback*** takes 2 params: ***err {Error}*** (null if there is no error) & ***result*** ***{Object}*** (result of function linked to the action pattern emitted, as well as the seneca.act call's ***response\_callback***). Note that these are chainable. Example:

**seneca.act(**

**'role:math,cmd:sum,left:123,right:27',** <-- msg (an action pattern)

**function( err, result ) {** <-- response\_callback (handles msg response)

**if (err) return console.error(err);** <-- handling of **err** param

**console.log( result );** <-- usage of ***result*** param

**})**

**.act( ... etc ...)**

**seneca.add**:Used to add a new 'action pattern' to the Seneca instance, which is matched against any JSON msgs the Seneca instance receives. Takes 2 params: ***pattern {Object||String}*** (property pattern to match in msgs), & ***action {Function}*** (fn to run if a msg matches). ***action*** takes 2 params: ***msg {Object}***(matching inbound msg), & ***respond {Function}*** (callback used to provide a response to the msg). Note that these are chainable. Example:

**seneca.add(**

**'role:math,cmd:multiply',** <-- pattern {String} – but could be {Object} too

**function( msg, respond ) { <--** action {Function}

**var result = msg.left \* msg.right <--** use of msg {Object}

**respond( null, { answer: result }) <--** use of respond callback

//--> emits {answer: {{{numGeneratedByTheCalc}}}

**})**

**.add( ... etc ...)**

**seneca.client:**

http://localhost:10101/act?role=math&cmd=sum&left=1&right=2

**seneca.use**: Used to include a plugin in a Seneca instance. Example:

**seneca.use(require('path/to/plugin/module', { option1: 'optionVal1', option2: 'optionVal2' });**

--- where the plugin module ('**path/to/plugin/module.js'**) could be e.g.

**module.exports = function math( options ) {**

**this.add( 'role:math,cmd:sum', function sum( msg, respond ) {**

**respond( null, { answer: msg.left + msg.right } )**

**});**

**this.wrap( 'role:math', function( msg, respond ) {**

**msg.left = Number(msg.left).valueOf()**

**msg.right = Number(msg.right).valueOf()**

**this.prior( msg, respond )**

**})**

**})**

[[[[[[[[[[[[[[[[[[[ INCOMPLETE ]]]]]]]]]]]]]]]]]

**seneca.wrap**: method that matches a set of patterns & overrides all of them w/ the same action extension fn. The same as calling seneca.add manually for each.

Takes 2 params:

***pin {Object|String}***: pattern-matching pattern e.g. ***'role:math'*** – which will match patterns '***role:math,cmd:sum'*** and '***role:math,cmd:productl'***

***action {Function}***: action extension fn -

**Seneca instance**: ultimately a set of action patterns, organized using namespacing conditions; all with a minimalist notion of a 'plugin' to help with logging & debugging (each plugin is itself just a set of action patterns – which can be given a name to annotate logs).

**this.act**: allows action patterns to trigger other action patterns – especially any where all its params match, but do not get run automatically by virtue of having fewer total matching params. Example:

**seneca.add( 'role:math,cmd:sum', function( msg, respond ) {**

**respond( null, { answer: msg.left + msg.right } )**

**})**

**.add('role:math,cmd:sum,integer:true', function( msg, respond ) {**

**//reuses role:math,cmd:sum, w an added element: the params the calc is done on are converted to integers 1st**

**this.act({**

**role: 'math',**

**cmd: 'sum',**

**left: Math.floor(msg.left),**

**right: Math.floor(msg.right),**

**}, respond)**

**})**

-- “this” is a reference to the current Seneca instance

**this.prior**: If you define 2 seneca.add instances w/ the same action pattern, this.prior lets you access the 1st match in the control flow from the 2nd match (2nd match takes precedence & runs by default).

Takes 2 params:

***msg {Object}*** (msg obj to pass to the overridden action fn – which can be modified from or identical to what the overriding action fn received); &

***reponse\_callback*** ***{Function}*** (callback fn where result of calling overridden action fn can be further changed)

Example:

//Doesn't automatically run by default on match: it gets overridden by the action fn declared later w same action pattern

**seneca.add('role:math,cmd:sum', function( msg, respond ) {**

**respond( null, { answer: msg.left + msg.right } )**

**})**

// overrides role:math,cmd:sum declared previously, with additional functionality

**.add('role:math,cmd:sum', function( msg, respond ) {**

// bail out early if there's a problem

**if( !Number.isFinite( msg.left ) || !Number.isFinite( msg.right ) ) {**

**return respond( new Error("Expected left and right to be numbers.") )**

**}**

// call previous action function for role:math,cmd:sum

**this.prior({ role: 'math', cmd: 'sum', left: msg.left, right: msg.right },**

**function( err, result ) {**

**if( err ) return respond( err );**

**respond( null, msg.left+'+'+msg.right )**

**})**

**})**

-- this.prior is truly an evil thing. Its existence means the order you include Seneca plugins matters, which makes for a deeply unmanageable type of dependency hell.

**Transport independence**: allows sending of messages betw services in multiple ways, all hidden from business logic. When your business logic doesn't need to know how messages are transported or which service will get them, there is transport independence.

# Create microservice

**seneca.add(**

**{role:'math', cmd:'sum'},**

**function( msg, respond ) {**

**var sum = msg.left + msg.right**

**respond( null, { answer: sum } )**

**})**

## Components

**seneca** <-- call to the framework

**.add(** <-- request to add a new microservice (msrv)

**{role:'math',cmd:'sum'},** <-- name of command (the whole json is the name)

**function( msg, respond ) {** <-- fn launched when msrv run; msg has params passed to msrv

**var sum = msg.left + msg.right;** <-- code the microservice runs on the received params

**respond( null, { answer: sum } )** <-- result passed to response fn. No err occurred so 1St param null

**})**

1. call framework and request add
2. Define command
3. define function launched when command called
4. give response at end of function via **respond( nullOrError, response )**;

# Call microservice

**seneca.act(**

**{role:'math', cmd:'sum', left:1, right:2},**

**function( err, result ) {**

**if( err ) return console.error( err )**

**console.log( result )**

**})**

--> 2

# Patterns – how services are called

## Pattern override rules (deciding which of many similar action patterns run)

* More properties win
* If 2 patterns have an equal number of properties, they resolve alphabetically
  + e.g.:
    - a:1, b:2 wins over a:1 – the 1st has more properties
    - a:1, b:2, d:5 wins over a:1, c:3, d:4 - the 1st wins alphabetically, as b comes before c in the alphabet
* If 2 patterns are precisely identical, the most recent pattern takes precedence. However, the overriding pattern's function has access to the prior pattern as follows:

**seneca**

**.add(**

**'role:math,cmd:sum',**

**function( msg, respond ){**

**var sum = msg.left + msg.right;**

**respond( null, { answer: sum });**

**})**

**//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* THE OVERRIDING FUNCTION \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* //**

**.add(**

**'role:math,cmd:sum',**

**function( msg, respond ) {**

// **RUNS BEFORE** OVERRIDDEN SERVICE RUN

if (!Number.isFinite( msg.left ) || !Number.isFinite( msg.right )) {

return respond( new Error("Expected left & right to be numbers") );

}

**// RUNS OVERRIDDEN SERVICE**

this.prior({ // <-- key point

role: 'math',

cmd: 'sum',

left: msg.left,

right: msg.right

**},**

**// RUNS AFTER OVERRIDDEN SERVICE**

**function(err, result){**

if (err) return respond(err);

result.info = msg.left + '+' + msg.right;

respond( null, result );

**})**

**})**

**//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* THE OVERRIDING FUNCTION \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* //**

# **Plugins**

* A plugin is just a set of action patterns, grouped under a name for organization. In practice: a fn w a single param: “options”
  + plugin def passed to the seneca.use method. Example of simplest plugin:

function minimal\_plugin( options ) {

console.log(options);

}

*[[[ ... elsewhere, perhaps even in another file if a “require” is used for minimal\_plugin ...* ]]]:

require( 'seneca' )()

.use( minimal\_plugin, { foo:'bar' } );

* has mechanism to execute init fns (basically constructors) in order (eg. db connect should be done b4 db data access)
* Intended to help w/ logging & debugging

## Plugin action patterns

### **Example plugin with action patterns**

**function math( options ) {**

**this.add( 'role:math,cmd:sum', function( msg, respond ) {**

**respond( null, { answer: msg.left + msg.right } )**

**})**

**this.add( 'role:math,cmd:product', function( msg, respond ) {**

**respond( null, { answer: msg.left \* msg.right } )**

**})**

**}**

***... elsewhere, perhaps another file if “require” used to call math plugin ...***

**require( 'seneca' )()**

**.use( math )**

**.act( 'role:math,cmd:sum,left:1,right:2', console.log ); //---> null { answer: 3 }**

**.act( 'role:math,cmd:product,left:4,right:3', console.log ); //---> null { answer: 12 }**

* note that a plugin should never have seneca.act called within it - seneca.add only

## Initializing plugins

* Special {init:'insertNameOfPluginHere'} pattern called to initialize a plugin
* init function must call its respond callback without errors
* All plugins must complete initialization before any actions are executed

### Initialization pattern

this.add({ init:'nameofplugin' }, init) //special initialization pattern

function init( msg, respond ) {

// log to a custom file

fs.open(options.logfile, 'a', function(err, fd){

if(err) return respond(err); // cannot open for writing === fail -- this err is fatal to Seneca

log = make\_log(fd) ;

respond() ;

})

}

function make\_log(fd){

return function(entry){

fs.write( fd, new Date().toISOString() + ' ' + entry,

null, 'utf8', function(err) {

if(err) return console.log( err );

fs.fsync(fd, function(err) { // ensure log entry is flushed

if(err) return console.log(err);

});

});

};

}

## Examining plugins

**node math-plugin.js --seneca.log.all | grep math**

* ...run this on the plugin from the command line to get intensive logging information
  + including inbound and outbound messages – which is of particular interest

## Transport plugins

* There are many ways to pass messages, many of which require plugins, such as:
  + seneca-redis-transport: <https://github.com/rjrodger/seneca-redis-transport>

[[[ RETURN TO THIS LATER ]]]

# Built-in plugins (included with main Seneca module)

## Seneca-basic {{{BUILT-IN}}}

* Utilities to send values around in an internal hashtable
  + set a value in the seneca hashtable (marked by a key), get a value
  + create a list (marked by a key) or push a value to a list with the given key (if it exists already), or pop the last list item
* built-in patterns:::

### {role:basic, note:true, cmd:set}

#### *params: key:'keyName' value:'valToAssign2Key*

#### *returns: (err)*

* Set 'keyName' inside internal Seneca hashtable to have value 'valToAssign2Key'
* Example:

**seneca.act({role:'basic', note:true, cmd:'set', key:'key1', value:'value1'}, console.log);**

### {role:basic, note:true, cmd:get}

#### *params: key:'keyName'*

#### *returns: (err, value previously assigned to key)*

* Get the value in the internal hashtable at key 'keyName'
* Example:

**seneca.act({role:'basic', note:true, cmd:'get', key:'key1'}, function(err, value){**

**console.log(value);**

**}); //--> 'value1' -- assuming we ran the previous example, where 'key1' was set**

### {role:basic, note:true, cmd:push}

#### *params: key:'keyName', value:'valToAddToArr'*

#### *returns: (err)*

* Create array at key 'keyName', containing value 'valToAddToArr'. If an array already exists at 'keyName', add 'valToAddToArr' to the existing array
* Example

**seneca.act({role:'basic', note:true, cmd:'push', key:'list1', value:'list1\_item1'});**

**seneca.act({role:'basic', note:true, cmd:'push', key:'list1', value:'list1\_item2'});**

* + - key 'list1' now contains [ 'list1\_item1', 'list2\_item2' ]

### {role:basic, note:true, cmd:list}

#### *params: key:'keyName'*

#### *returns: (err, array of values previously assigned to key)*

* Display all values assigned to key 'keyName' via *cmd:push* (see previous command). Emits array.
* Example:

**seneca.act({role:'basic', note:true, cmd:'list', key:'list1'}, function(err, value){**

**console.log(value);**

**}); //--> *[ 'list1\_item1', 'list2\_item2' ]*** --- assuming the previous example was run

### {role:basic, note:true, cmd:pop}

#### *params: key:'keyName'*

#### *returns: (err, item most recently added to array at key 'keyName')*

* Remove most recently added item from array at key 'keyName', and return it
* Example:

**seneca.act({role:'basic', note:true, cmd:'pop', key:'list1'}, function(err, value){**

**console.log(value);**

**});** //--> { value: 'list1\_item2' } -- assuming the *cmd:push* examples ran

## seneca-transport plugin {{{BUILT-IN}}}

* provides HTTP and TCP transport channels for transmitting & receiving Seneca messages
* lets you use the ***.listen*** and ***.client*** methods on the Seneca object
  + .listen sets a Seneca instance to listen for messages passed to it via a certain port
  + .client sets a Seneca instance to transmit messages over a certain port, to see a response from another Seneca instance (that is listening on that port)

## seneca-web plugin {{{BUILT-IN}}}

## seneca-mem-store plugin {{{BUILT-IN}}}

* in-memory storage engine with a set of action patterns allowing you to store data

# Plugins – third-party

## seneca-redis-store plugin {{{BUILT-IN}}}

* in-memory storage engine with a set of action patterns allowing you to store data

# Instantiating Seneca – magic options

## Defining Seneca options

***var seneca = require('seneca')({***

***//options definable in here***

***});***

## See options defined already

***var seneca = require('seneca')({***

***debug: {***

***print: {***

***options: true***

***}***

***}***

***});***

* This will emit a giant JSON showing all options currently defined, and all available to set.

# Logging – to terminal

## **node name-of-plugin.js --seneca.log.all**

* Outputs large data table
* YOU WILL NEED TO GREP THIS!!! e.g.:

**node name-of-plugin.js –seneca.log.all | grep name-of-plugin**

* My clever module runner that effectively filters the output while keeping the plugin restarting on each save:

**nodemon math\_test\_1.js launch --seneca.log.all | grepx1i2 ADD math**

* This example displays all log lines containing “math” that do not also contain the word “ADD”

### Output

* **log attributes – info displayed in the output table (in order of display):**
* **date-time**: when the log entry occurred
* **seneca-id**: identifier for the Seneca process
* **level**: one of DEBUG, INFO, WARN, ERROR, FATAL
* **type**: entry code, such as act, plugin, etc
* **plugin**: plugin name (actions without a plugin have root$)
* **case**: entry case, such as IN, OUT, ADD, etc.
* **action-id/transaction-id**: tracing identifier, stays the same over the network
* **pin**: the action pattern for this message
* **message**: the inbound or outbound message (truncated if too long)
* **case** (in log attributes) - shows which operation an item represents - from:
  + DEFINE: declaration of the module
  + ADD: adding of action functions to module
  + options: setting of options value for the module
  + DEFAULT: display of default options values
  + register ready :
  + act <pluginName> IN : shows incoming messages (messages that hit the module)
  + act <pluginName> OUT : shows outputs of action functions from module
* **action-id/transaction-id -** unique action Ids: e.g.:

***2682lsrziy1i/rst61586f7wl***

* + these can let you trace message flow. All sub-actions have the same message ID as the triggering action, so you can see the route the initial action pattern triggered (i.e. since action functions can trigger other action functions)
    - for example, if you used ***this.wrap*** when defining your function, both the wrap function and the function it calls will be displayed in the flow

## node name-of-plugin.js –seneca.print.tree

* Displays a tree of dependencies between commands and functions
* Lets you see the name/value pairs of the action patterns arranged in a tree structure, along with any overrides. Action fns are indicated by the format: ***“# plugin, (action-id), function-name”***.

## node name-of-plugin.js --seneca.log=plugin: name-of-plugin

* Narrows log output to a single plugin of interest

## node name-of-plugin.js –seneca.log=pin:role:nm\_of\_role\_\_eg\_math

* filters all log entries where the action pattern contains the specific pin

# Multiprocess

# Broadcasting

* seneca.client and seneca.listen parameters must be the same for communication to occur
* e.g.

**module.exports = ((() =>**

**(require('seneca')()**

**//register microservices**

**.use('app/microservices/plugins/emitHello', { opt1: 'val1'})**

**.use('app/microservices/plugins/math\_test\_1', { opt1: 'someoptval1'})**

**.listen(11111)** ;

... AND ...

**seneca.client({ port:11111, host:'localhost' })**

**//Perform math actions**

**.act( 'role:math,cmd:sum,' + 'left:123,right:27',**

**log.seneca.info)**

**.act('role:math,cmd:multiply,' + 'left:10,right:5',**

**log.seneca.info);**

* ...can communicate without issue
* Different types of communication exist – such as tcp

**seneca.client({ port:11111, host:'localhost', type: 'tcp' })**

*... in another module ...*

**seneca.use('some/module')**

**.listen({ port: 11111, host:'localhost', type: 'tcp' })** ;

* + - tcp is better, because it has lower overhead and keeps the client<-->'listener'(microserver?) connection continuously open
    - default is http