How to Shiro

Complete Guide

Contents

[Getting Started 3](#_Toc28005168)

[An Introduction to the Shiro Ecosystem 3](#_Toc28005169)

# Getting Started

## An Introduction to the Shiro Ecosystem

Shiro is an interpreted, dynamic programming language (although it can be compiled, this is just really bundling the interpreter, runtime and your code together into an executable package) which looks a lot like LISP and behaves somewhat like it; I often refer to it as a LispScript. That sentence probably got rid of about 90% of the people who were considering learning Shiro, given that everyone hates dynamic languages and LISP-styled syntax. For the rest of you… welcome! Let’s begin.

The default Shiro distribution includes the following:

* shiro – a console application which can run Shiro, compile it, give you a REPL interface to play with it, install libraries and packages, etc.
* ShIDE – A windows IDE which makes writing Shiro almost pleasant. It has most of the features you’d expect of a modern IDE, and adds a lot of special commands and options to help you write and read in the LISP style effectively.
* The Shiro Standard Library – A set of libraries which provide basic functionality (math, file system manipulation, web service calling, etc.) that are packaged with Shiro. The shiro console application can easily install any Standard Library package to your project.

If you’re an old console grognard, just go into your console and type ‘shiro’ and you can immediately start playing with the REPL. This is a good way to follow along with the code samples we’ll be using to explain certain concepts in a little bit – just be careful about copy-and-pasting. The shiro REPL executes when it encounters two Enters in a row, so if your sample has a double-line break in it you might accidentally execute if halfway through.

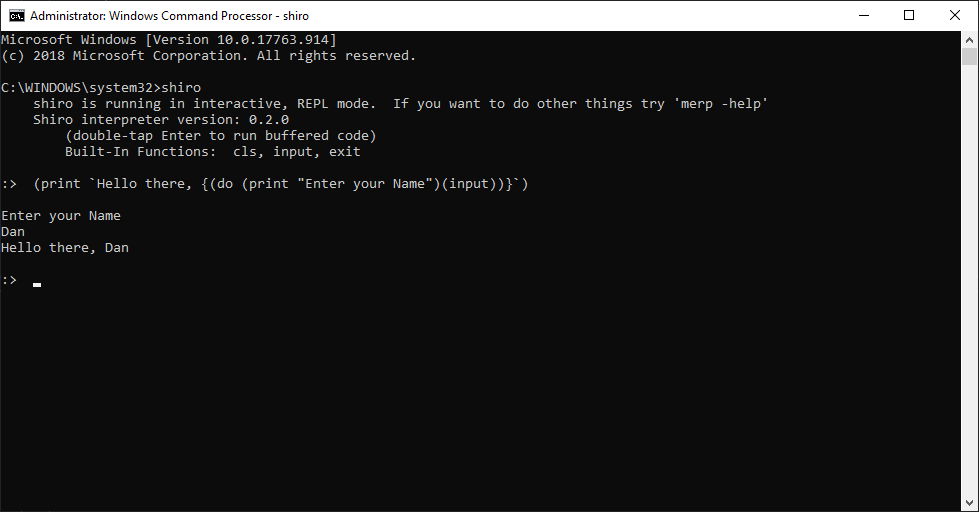


Figure 1 - The scary and ugly way to play with Shiro

Running ‘shiro help’ will show you a list of commands you can use with it if you don’t just want to run the REPL. Some example commands:

* shiro compile main library -output:program.exe
* shiro run myfile -sr
* shiro install math
* shiro uninstall math

I’m sure I’ll eventually have to come back to more-fully document these, but for right now they’re in a bit of flux and ‘shiro help’ and some messing around should get you close enough.

If you prefer something that looks like it was made after 1998, ShIDE is pretty much the way to go. The screenshot below was taken from a very early build of ShIDE (you can tell by the lack of menus and that there’s no project tree over to the left), but should give you a general idea. Notice how you’ve got all kinds of nice stuff like syntax highlighting, brace matching (this is a godsend with LISP dialects), autocomplete and multi-editing.

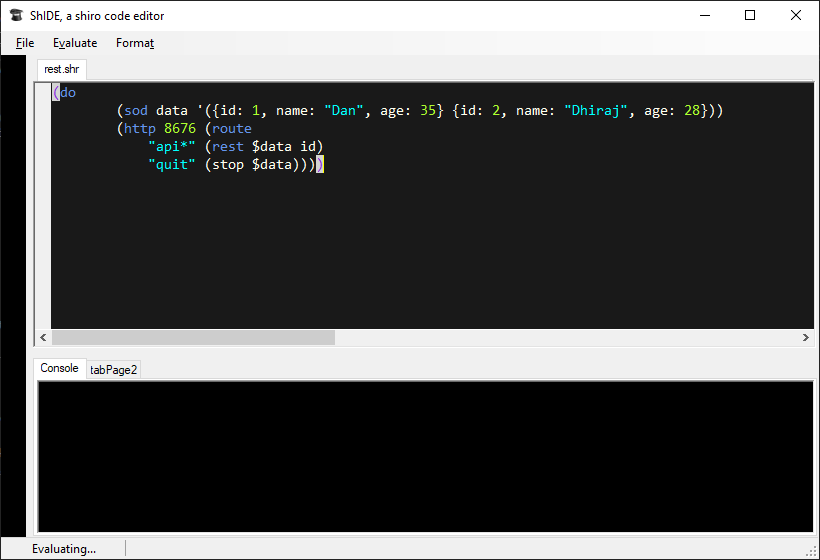


Figure - An early build of ShIDE, looking all cool

Don’t worry, we’ll learn all about how awesome ShIDE is in a later chapter.

If you bothered to check out that code in the second picture, you might be a little intrigued. Surely that can’t be what it looks like, right? A fully functional REST server in five lines of code, without even importing any libraries or extra modules? That’s like, unheard of in a programming language – seriously, try it in node. You’re 5 libraries, multiple source files and a hundred lines of code deep before you even get started.

Which brings up a good point…

## Why Learn Shiro?

Shiro is an obscure programming language that uses a syntax everyone hates and offers none of the type safety and general comfort that most people prefer in their development environments. So why on Earth should you learn it?

I'm only going to spend this one paragraph telling you about the benefits of learning some flavor of LISP, and how it makes you a better C# or Java or node or Scala programmer. LISP's syntax is an expression tree written out in code, it's what your compiler or interpreter builds to execute your code. Once you grok it, if you can write things faster and funner in shiro than in any other language.

Okay, evangelism over. Specifically, shiro was designed for a handful of purposes and it's very, very efficient and good at it. Making small web services (either for internal tools, or “mocked” services for automation testing or development), recurring DevOps processes, terminal/telnet style applications where you want to connect remotely and issue instructions, raw TCP/IP servers, and (because I love them), MUDs and MUSHes. Those last two are an obscure kind of online text game if you're curious.

I can make a full REST service in shiro with GET, PUT, POST, DELETE and PATCH in 3-4 lines of code depending on how you format it. No libraries are involved, none of the .NET stack or any other complex web host is involved, a very low level TCP/IP server with sufficient HTTP understanding to handle this is built right in to shiro. You don't need IIS, or even any third party packages. Seriously, here it is:

(do

(def data '({id: 1, name: "Dan", age: 35}

{id: 2, name: "Dhiraj", age: 28}))

(http 8676 (route

(rest $data id))))

Shiro also lets you define objects right in-line, as you see in this example. Despite being a dynamic language shiro has some pretty fun object-oriented capabilities built in. As we'll see later you could add validation, persistence and other things right to your data store just by adding some implementors and mixins.

So basically, learn shiro if you want to be able to make simple web services practically instantly for some reason (it's great for web UI development. No more waiting on the back end guys!) or stand up small microservices without having a large infrastructure or reliance on a web host. If you want to write anything that you talk to using Telnet or a MUD client... definitely learn shiro. It's also great for scheduled tasks and middleware services.

But if you're just looking for a fun new language that will let you make all the same kinds of little utilities and helpful programs you'd made in your stodgy other languages... consider giving shiro a try. You'll never find a language that takes less syntax to get some pretty complicated things done.

Alright, on to the learning!

The Basics

Generally speaking, shiro looks a lot like LISP, although small snippets can look more like JavaScript or some weird functional language. Shiro also works a lot like LISP, although not completely. I like to think of it as a kind of LispScript. While you use text to code in shiro, all your code is actually in the form a lists... the simple syntax the language uses is just an easy way to express lists in text. Let's mess around with a simple hello world type example. You can follow along in the REPL by just running 'shiro' in the console without any arguments. You might want to include the -sr command line argument to see the result of the code you evaluate, but that's optional.

Now type this:

print 'Hello Nurse!'

and press Enter twice. If you've ever programmed anything before you probably had a good idea what was going to happen, and lo-and-behold, it happened! But what was all that crap I said about everything being a list? That's not a list, it's just print hello world, the same thing you write in every other language whose print isn't a function call.

Well, shiro likes to be helpful, so it will wrap your top-level commands in a list for you if you forget. The actual, syntactically-correct way to do the above is this:

(print 'Hello Nurse!')

Which if you type and execute you'll see has the same result. The parenthesis tell shiro that we're starting a list, and then everything in it is parsed into a separate element. The list for this command looks like this:

print

hello world

Now the first thing in a list in shiro is a keyword (it can also be called a command), in this case 'print'. Lists are evaluated in shiro by looking at the keyword and then doing whatever that keyword is supposed to do. print is a fun keyword because it can take any number of arguments, so you can print a bunch of lines pretty easily:

(print 'This is a line' 'And this is a line' 'Guess what this is?')

Now if you're using the -sr command line argument that I talked about above, you might notice that the result of that list (which you can think of as its “return value”), is 'Guess what this is?'. print returns the last thing it printed. You won't often use the result of evaluating a print command, but keep in mind that all lists in shiro evaluate to something, even if the keyword you're using doesn't seem like one that should.

Let's say I want to print a list, I might do something like this:

(print (1 2 3)) ; wrong

And as the comment implies, I'd be very, very wrong to do so and shiro will yell at me about it. By the way, comments in shiro start with semicolons and run either until the end of the line, or the next semi-colon. There are no multi line comments, just put them in qnop lists or put semicolons at the beginning of each line.

But nevermind that, why is that example wrong? If you paid very, very careful attention to what I've taught you so far (or you know Lisp, or you read the error that the shiro REPL prints out) you can probably figure it out. I said the first thing in a list is the keyword for that list, and '1' is not a shiro keyword. The list we tried to evaluate looked like this:

print

1

2

3

And when shiro got to that second list it tried to evaluate it and threw up its hands in frustration because it couldn't. If you want a list that's just a list of stuff (like in this example), it's called a quoted list. They're awesome and useful for a lot of things, but be careful never to try to evaluate them. You can quote a list using the quote command, so:

(print (quote 1 2 3))

Which doesn't break shiro and should have the result you were expecting. You use quoted lists a lot, and typing five whole letters every time is annoying, so there's a shortcut to do it, just use the single-quote on your keyboard before the list-opening parenthesis, like so:

(print '(1 2 3))

Every shiro program (including something you evaluate on the REPL) is a single list. If you try to do something like:

(print 'hello world')

(print 'oh yeah and hello universe too')

You receive the most instructive error message of all time (well, not really. Shiro is big on instructive, helpful error messages):

[error] Sibling peered list passed for evaluation –

you are probably missing a 'do' keyword

As the error implies, you need to turn your multiple-lists into a single list, and you can use the do keyword for that. I'm sure you already figured it out, but it looks like this:

(do

(print 'hello world')

(print 'oh yeah and hello universe too'))

Like print, do evaluates to whatever the last thing in it evaluates to. Strings are pretty JavaScripty. You can use either double or single quotes for them (allowing you to use the other kind of quote inside the string). They can also have line breaks in them. There are several escape characters you can use in a string, like %n (newline). We'll list them all out later. Here's a little snippet that shows off a few of these attributes:

(print "Strings can have%s

line breaks in them , and also%s

can include the 'other kind' of quote. You can escape the%s

quote you used for the string %"like this%"")

Alright, that's literally all you need to know to start learning shiro. There aren't any other weird syntaxes you need to learn, no other rules of grammar. You just make lists, and then evaluate each other and ultimately result in doing something.

In Which Things Start to Become a Little Bit Scary

Alright, let's learn some shiro keywords. We're going to go over a lot of core concepts, so you might have to re-read this section or play around with the code samples a bit to really understand what's going on. We're going to cover math, control flow, functions, variables and a bit about objects.

Variables are pretty standard in shiro. Note that I've included a few extra line breaks for clarity in this code sample, which is fine if you're using the interpreter or compiler, but will cause the REPL to try and evaluate half a list and complain.

(do

(def x 1) ; declare x for the first time and set it to 1.

; A variable can only be defined once

(set x 5) ; set existing variable x to value 5

(sod y 10) ; 'sod' makes it easy to work with variables, by

; using either set or define (get it? s.o.d)

(sod y 23) ; see?

; You need to explicitly get the value of a variable:

(print (str "x = " (v x)))

; ... but there is a reader shortcut to do this using $:

(print (str "y = " $y)))

So all those variables are global. There is only ever one 'x' per shiro instance. If you're using the REPL, it will persist throughout the entire session, otherwise it will exist in all your different code files because they share the same instance of the shiro runtime. You can, however, create you own local variables whenever you want by making a scope level. You do that with the let keyword:

(do

(def i 255)

(let

(i 1 j 2)

(do

(print $i) (print $j)))

(print (def? i) (def? j) $i))

So what are we doing here? First I'm making a global variable called i and setting it to 255. We'll use this fact later, so keep it in mind. Then we encounter our let keyword. Let takes two arguments, and the first one is somewhat special because it has what's called an implicit quote. As you note it's a quoted list ('i' is not a shiro command), but we didn't have to quote it. I figured I'd save you the keypress. This first list must have an even number of things in it. They are basically paired, with the first being the variable name and the second the default value.

Variables in a let-scope hide global variables, so i inside the let is 1, not 255. Once we leave the let-scope the global i is unhidden and retains its original value. You'll notice (at least, if you will if you figured out that that def? keyword returns true if something is defined and false otherwise) that variables inside a let-scope are destroyed when the scope ends, so j ceases to exist.

You can do all the ordinary kinds of math and comparison that you're used to in other programming languages, but you do it using the rules of shiro syntax, so the command (or in this case, the operator) goes at the beginning of the list. Here are some examples:

(+ 2 2) ; 4

(+ 2 (- 3 1)) ; 4

(+ 3 3 3) ; 9

(= 2 2) ; True

(= 2 2.5) ; False

(= 2 (/ 4 2)) ; True

(= nil "nil") ; False

(! true) ; False

(! nil) ; True

(! 0) ; True

(> 3 2) ; True

(>= 2 2) ; True

Most of this is pretty straightforward, about the only really interesting things to note is nil, which is a particular value in shiro that means “nothing”, it's like NULL in other languages. Also note that shiro has truthiness like JavaScript, so you can use numbers, objects or even strings as booleans without incident – at least without incident if you knew you were doing it and intended to.

Now that you know shiro has booleans (duh), you can probably also guess it has ways to branch based on them. To do so we use the innovative keyword if:

(if true (print "Hello world"))

(if false (print "Won't Print") (print "Will Print"))

Remember how I said that every list in shiro evaluates to something? Well because of that property, the if keyword can also be used just like a ternary operator (the ? : in most languages). Like so:

(print (if false "Won't Print" "Will Print"))

And of course those strings could be lists as well, and if you keep extrapolating that you're programming in shiro! You can loop in shiro (while loops at least), although you're being kind of weird most of the time if you do so because there are much better ways to do it like the map, filter and apply keywords we'll learn about later. But if you want to be weird, here's a while loop in shiro:

(do

(sod x 10)

(while (> $x 0) (do

(print $x)

(set x (- $x 1)))))

Stunning, right?

Shiro has functions (boy oh boy does shiro has functions!). The least interesting kind are just... well functions. You define them, they have names, and you call them just like everything else in shiro, by putting that name at the beginning of a list and evaluating that list.

(defn say-hi (name)

(print (str "Hello " $name)))

Other than the str command (which is how we concatenate strings in shiro), this all reads pretty easily. Define a function named say-hi, which takes one parameter called 'name', then says hello to the name. You call it like any other first-class shiro command:

(say-hi Dan)

Functions are okay I guess... they basically let you make your own language keywords, which is neat, but they're so static and monolithic and boring, it would be much cooler if there were functions that weren't named anything and were just passed around like values. Good News Everyone! There is a type of function just like that called a lambda or anonymous function. I can make one that works a lot like say-hi above by doing this:

(sod say-hi (fn s (print (str "Hello " $name))))

Paste that hideous, chthonic gibberish into the REPL, then try typing 'say-hi Dan' again and lo and behold, it works just the same. The reason for that is that we created a variable named say-hi in that snipped and actually assigned a lambda (a function) to it. The 'fn' keyword creates and evaluates to a lambda, with the first parameter being the argument list and the second the body. You can also use => instead of fn as the keyword if you like making sure people can't read your code.

Now if all you could do with lambdas is assign them to variables and call them just like functions they'd just be functions with extra steps and slightly less efficiency at runtime. Fortunately, there's so much more you can do, like passing them as parameters to other functions, or to keywords. For example there's a keyword apply in shiro which applies a particular command to everything in a list. You can use it like this:

(apply print '(1 2 3))

Don't worry we'll be talking that stuff to death a bit later on. For now it's good to understand that you can also use a lambda as the first parameters of apply (or any similar command):

(apply (fn s (print $s)) '(1 2 3))

Lambdas by themselves can even be commands, although this makes for some very ugly syntax. Here's a very obscure way to calculate 2+2 in shiro:

((=> (x y) (+ $x $y)) 2 2) ;note => and fn are interchangeable

If you're not confused by that then I must be doing a really good job describing Lisp syntax. Basically the first item of this list is a list that evaluates to a lambda, which is something shiro knows how to treat as a command. The next 2 parameters (2 and 2) are the parameters to the lambda.

Finally you've already seen that we sometimes make what look like JavaScript objects in shiro. They do a lot of what you'd expect a JavaScript object to do. You make them like this:

(sod o {name: 'Dan', age: 35, loc: 'OR' })

And then you can:

(do

(print (. $o name))

(.sod o name 'Steve')

(print (. $o name)))

Notice that the dots work a lot like dots in normal languages, they just use shiro syntax instead of the more traditional one. You can dereference down any number of layers with a single dot, so if you have objects containing objects containing objects you can get even to the innermost properties with a single list. Keywords like .sod in the example above (and the obvious counterpoints .def and .set) can be used to change and create new properties on objects. If you're not sure if a particular object has a particular property, you can use the .? command, which returns nil if it can't find any of the properties you ask for.

Now objects in shiro are just lists (everything in shiro is a list), but they have a special property wherein the values in the list have names. These are called pairs. You can make a pair using the pair keyword, which you can use as a backhanded way of adding things to objects if you don't want to use .sod for some reason. Check it out:

(do

(sod obj {name: 'dan'})

(print (.? $obj fakeProperty)) ; nil

(sod obj (concat $obj (pair fakeProperty "Its magic!")))

(print (.? $obj fakeProperty))) ; "It's magic!"

So we make an object with a single property (name), prove that there's no property named fakeProperty on that object, then we add a new pair to obj using concat (a keyword which concatenates lists) and pair to make a new named value. Then we prove that the new value is there. This bit is just here to help you understand a bit about objects... if you really want to add a property to an object that doesn't have it, this is how you do it:

(do

(sod obj {name: 'dan'})

(.sod obj fakeProperty "Its magic!")

(print (.? $obj fakeProperty)))

Fun with Lists

If you're still reading and understanding, you're probably starting to get shiro a little bit even if you don't have a background with this sort of syntax. Everything's a list, often a list of lists, and we just sort of evaluate them from the innermost lists to the outermost ones until we get a final result. Cool.

Since shiro is a programming language where everything is a list, there are a bajillion ways you can manipulate lists using different commands and functions. And since every list is technically also code, you can use these functions to dynamically build executable shiro and it's no different from the code you'd write to manipulate a list.

When you're making a list, remember the basic rule – the first thing in the list is the command unless the list is quoted (in which case this is still true, but the interpreter sneaks a 'quote' keyword in there for you). So when you're making a list, if you want a list that's purely data (like an array or linked-list type thing) then you want to make sure it's quoted, otherwise you might be accidentally building code that shiro will try to evaluate.

There are lots of ways to slice and dice lists to your needs (and I do mean 'lots'). You can get the keyword of a list (the first thing in it) with the 'kw' command, and you can get the rest of the list with the params keyword. Here are some examples of those two and others simple ways to get stuff out of lists:

(do

(print (kw '(1 2 3))) ; 1

(print (params '(1 2 3))) ; 2 3

(print (nth 2 '(1 2 3))) ; 2

(print (range 2 2 '(1 2 3 4)))) ; 2 3

But really you very rarely want to slice lists up this way, and when you do you're either doing something very boring, or very interesting like making dynamic code at runtime. It's a lot more interesting to do things to stuff in lists. A lot of the time you use a for loop or a foreach loop in your programming language of choice to iterate through a list; you do that same stuff in shiro, but of course it's different. Let's take a tour...

(do

(sod stuff '(1 12.5 'Dan' ))

(print (filter num? $stuff))) ; (1 12.5)

asdasd

; You can do some pretty cool things with quoted lists, like:

(eval (concat '(print 0) $x)) ; Concatenating a keyword (and even other parameters) and evaluating the resulting list

(eval (skw print $x)) ; skw sets the 'keyword' (the first item) in the list, which can then be evaluated

(sod x '(1 2 "dan was here" "hello world"))

(print (filter str? $x)) ; Pull out the strings. The first parameter to filter can be any keyword or function.

(defn say-hi (name) (print (str "Hello " $name))) ; By the way this is how you declare functions

(map say-hi '("Dan" "Dhiraj" "Dave")) ; map evaluates the first parameter individually against everything in the list in the second parameter and returns the result list

; of that evaluation

;the line above is equivalent to:

(say-hi "Dan")

(say-hi "Dhiraj")

(say-hi "Dave")

;You can also map to keywords, like this:

(map print '(1 2 3))

)

; There are lots of ways to slice-and-dice lists:

(kw '(1 2 3)) ; 1

(params '(1 2 3)) ; 2 3

(nth 2 '(1 2 3)) ; 2

(range 2 2 '(1 2 3 4)) ; 2 3

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Remember that 'str?' thing from the filter example above? Let's look at some of the other ones:

(sod x '(1 2 3))

(sod y 2)

(sod z {name: "dan", age: 35}) ; More on this later, don't panic!

(sod s "Hello nurse")

(list? $x) (list? $z) ; T

(list? $y) (list? $s) ; F

(obj? $z) ; T

(obj? $x) (obj? $y) (obj? $s) ; F

(num? $y) ; T

(num? $x) (num? $z) (num? $s) ; F

(def? x) (def? s) ; T

(def? bob) ; F

(sod s (json $x)) ; json returns the inline-object in question, JSON-serialized.

(sod obj (dejson $s)) ; duh

(print (. $obj name)) ; obj is now the same as x, by way of JSON

(filter (=> (n) (> $n 5)) '(1 10 7 3 -4 154))

; Notice the difference between:

(apply (=> (x) (+ $x 1)) '(1 2 3 4)) ; results in '(1 2 3 4)

(map (=> (x) (+ $x 1)) '(1 2 3 4)) ; results in `(2 3 4 5)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Sometimes you have a thought in the shower like, "lisp would make a way better node.js than js"

; and sometimes you actually go and do it. But before we get into that, let's talk about

; the world's simplest telnet chat server:

(telnet 4676 (sendAll (str $id " says '" $input "'")))

; Shiro has a few keywords (like 'telnet' above) that go into network server mode. This has a few properties:

; a) The interpreter's main thread (the one that executes your Shiro) begins blocking. A multi-threaded network

; server component will begin listening, and as events occur which evaluate Shiro they will be evaluated by

; the network server's threads. Don't worry, your Shiro is always thread-safe.

;

; b) The main thread doesn't go away -- all your code and variables are still there, and if the network-server

; ever executes a 'stop' keyword it will come right back. You can even return something from the network thread

; to the main thread by passing it as a parameter to stop. Here's a telnet server that can be stopped:

(telnet 4676

(if (= $input "quit")

(do (print "quitting")(stop $input))

(print $input)))

; If you telnet into this server and type anything it will print out in the Shiro window. If you send it 'quit'

; the Shiro server will stop listening and return "quit" to the main thread.

;

; c) A series of local variables (ie: 'let' scoped variables) will be created for Shiro evaluated in the server's

; context. For telnet, these are id (a guid-as-string uniquely identifying the socket which triggered the

; evaluation) and input (the full line-command sent to the server).

;

; d) Several different keywords will become available for use, depending on the type of the server. In this telnet

; example, they are send, sendTo and sendAll.

; Telnet can also take an optional third parameter which is a list that will be evaluated whenever someone connects,

; this has an id let-scoped variable, but not an input:

(telnet 4676

(send $input) ; echo whatever is sent back to the client

(send "Hello and welcome!"))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Okay, we're finally ready for the fun stuff! Let's make some web services:

(http 8676

(print (. $request url))

; This is pretty lame. It listens on port 8676, negotiates an HTTP connection, prints out

; the request url and then returns the request url back to the client as whatever the default

; content-type is. It's worth noting that there are no $id and $input like in a telnet server,

; instead there is a $request variable, which is an inline object that has the HTTP request properties on it

; Today's web services are normally returning json, so let's do that:

(http 8676

(content "application/json"

(json {name: "Dan Larsen", age: 35})))

; content is a keyword which can only be executed in an http-server context (like telnet's 'send' variants)

; it sets the content-type header of the returned value. We can use the json keyword that you already know

; to turn one of our inline-objects into a json string.

; So normally websites and webservices have different endpoints on them, they don't respond to all connections

; the same way. We call this routing the request, and like most of this stuff so far, it's pretty easy to do in Shiro:

(http 8676 (route

"getJson" (content "application/json" (json {name: "Dan Larsen", age: 35}))

"quit" (stop)

"default" (status 404 "Endpoint not Found")))

; Check out:

; http://localhost:8676/getJson

; http://localhost:8676/pageDoesntExist

; http://localhost:8676/quit

; Alright let's put it all together and build a mocked web service which can store and retrieve data:

(do

(sod data {dan: {name: "Dan", age: 35}, dhiraj: {name: "Dhiraj", age: 28}})

(defn getOrDefault (key)

(if (nil? (.? $data $key))

(status 404 "Id not found")

(. $data $key)))

(http 8676 (route

"get" (content "application/json" (json (getOrDefault (. $request args id))))

"store"

(do

(set data (concat $data (pair (. $request args id) (dejson (. $request args json)))))

(status 200 "Saved"))

"quit" (stop $data))))

; Check out:

; http://localhost:8676/get?id=dhiraj

; http://localhost:8676/get?id=1 Not found

; http://localhost:8676/store?id=1&json={name:"Steve",age:21}

; http://localhost:8676/get?id=1

; Or we could just stand up a REST service. Note that route can also take a lambda which takes a

; single parameter and returns a boolean, which is useful for routing rest requests.

(do

(sod data '({id: 1, name: "Dan", age: 35} {id: 2, name: "Dhiraj", age: 28}))

(http 8676 (route

(=> s (contains $s "folks")) (rest $data id)

"quit" (stop $data))))

; You can now GET/POST/PUT/DELETE on

; http://localhost:8676/folks

; like (GET):

; http://localhost:8676/folks/1

console app built-ins (includes compiled):

cls / input

examples:""

(print (str 'hello ' (do (pnb 'Enter your name: ') (input))))

keywords:

json/jsonv ;The latter evaluates the object before JSONifying it, the former just JSONifies it

dejson

pair

print

printnb / pnb

quote / '(

string / str

def

set

sod (SetOrDef)

eval

skw (Set KeyWord)

concat

v / $<name>

. / .? ;latter returns nil if not found, former throws

+ - \* /

= ! !=

> < <= >=

list?

obj?

num?

str?

def? ;takes a name, not a $name

fn?

nil?

let

nop / qnop

defn

filter

map ;map does the process and returns the result-list (so if the lambda changes the value in the list, map returns the new value)

apply ;works like map but returns the original, unmunged list

kw / params

nth / range

while ; (while (condition) (action))

contains

upper / lower

split

=>/fn ; (=> (args) (body))

.s[et]

.d[ef]

.sod

telnet <port> <command handler> [<connect handler>] ;within command handler, 'input' contains the value of the last telnet command, and 'id' contains a guid-string of the connection id

send / sendTo <id> <msg> / sendAll ;only available for telnet connections

stop [<result>]

http <port> <handler>

content "<content-type>" <return list>

route {"routeContains"|<lambda>} <return list>...

status <#> <return list>

rest <collection> <id field name>

http request:

body

url

args

method

Alphabetical Index