Abstract

Reference documentation for the source-integrated unit testing engine for Rebol /Red language.

Slim unit testing engine

(v1.0.3)

Quick intro

Slut is the unit testing engine for use in Rebol source code projects. It’s an integral part of the slim library manager.

The tests are specified within the source, so they are very close in proximity to the source being reviewed.

This guarantees a higher degree of confidence in the test’s validity.

# Getting Slut

You can get slut and all dependencies here:

<https://github.com/moliad/slim/archive/master.zip>

<https://gitlab.com/coginov/semantic/libs/slim/repository/master/archive.zip>

[git@gitlab.com:coginov/semantic/libs/slim.git](mailto:git@gitlab.com:coginov/semantic/libs/slim.git)

# Embedding tests in your code

The idea is to add comments within your source code which will represent a unit test.

Because the tests are embedded directly within the source, your tests will more likely be maintained and be relevant to the process.

Here is an example:

count: func [numbers [block!]][

; test-group [count series block! utils-series.r ] [ ]

; [ 2 = count [1 2 3 4 5 3 4] [ 3 4 ] ]

; [ 0 = count [1 2 3 ] ]

; end-group

Val: 0

Foreach value numbers [val: val + number]

]

# Running tests

In order to launch your tests, simply load the slut.r library and run the following commands in sequence:

do %slim-libs/slim.r

slut: slim/open ‘slut 1.0.3

slut/extract script-path

slut/do-tests

do-tests() has a few useful refinements for viewing and controlling tests, mainly:

/ANY-OF […] Given a list of tags, only performs tests qualified with any one of those tags.

/ALL-OF […] Given a list of tags, only perform tests qualified by all those tags.

/VERBOSE Traces tests in console as they are performed.

Guide

# Retrieving slim and slut

The best way to use slut, is to do a git clone of the Slim repository and of all the slim packages you need.

You can get it here:

[git@gitlab.com:coginov/semantic/libs/slim.git](mailto:git@gitlab.com:coginov/semantic/libs/slim.git)

[git@github.com:moliad/slim.git](mailto:git@github.com:moliad/slim.git)

The repository on gitlab tends to be a bit more up to date.

Extract the files and you’re ready.

To load up slim just use the Rebol DO function on the filepath where you installed the slim.r file.

Ex:

Rebol [ ]

do %/c/slim-libs/slim.r

# General Syntax

There are two ways to embed slut instructions within your source code. We will define the actual test commands later, but its good to know that all of them are support in two distinct notations.

## Comment mode

The comment mode test specification is a simple notation where you prefix ALL lines of your test specification (single, group, or otherwise) with simple Rebol comment markers. The system can thus be hidden completely from the Rebol Interpreter and live anywhere in the source.

A simple in-line comment makes it very easy to put your tests very close to the code they are supposed to verify.

You can even put your comments within data, anywhere a comment is allowed in Rebol. This is useful to put tests in places which may impossible in block mode, which requires block/data coherence within the source.

Here is an example of a single test case:

; test [ simple app.r ] [ 1 = first [ 1 2 ] ]

## Multi-line tests

By default, simple tests are a one-line affaire, but you may need to have more code or many tests, so you should use the multi-line block test syntax. You simply wrap your test in a <[ ]> pair like so:

; test [ multi app.r ] <[

; i: info? %file

; all [

; i/date > (now – 2) ; max two days old

; i/type = ‘file

; true

; ]

; ]>

Don’t forget to include all the text within rebol comments 😊 .

## Block mode

(future specification, not yet implemented)

The block mode is useful to enter tests with simpler syntax, since you do not need to prefix each line with a comment marker.

The downside is that it must maintain block and source code coherency and it usually cannot be embedded within data and many intricate parts of source code like function argument specs.

Block mode specified tests also cannot be in-lined within most constructs like ANY / ALL decision trees.

Here is an example of the same test above, using block mode syntax:

#TEST [ [ simple app.r ] [ 1 = first [ 1 2 ] ] ]

Note that the WHOLE test is wrapped within a single outer block marker. This is to ensure the content is considered as a single data element within the source. It also makes it much easier for the slut source parser to make sure the complete test is loaded in a single instruction with no chance for ambiguity in specification.

Note that adding issue! And block! data within source code doesn’t bother Rebol, because it will simply skip over it, as long as it isn’t imbedded between a function (word) and its arguments. Obviously putting an extra issue! and block in any dataset will also break your source, so use the comment mode syntax instead.

# Direct slim integration

An extremely powerful feature of slut is that it’s tightly integrated with the slim library management engine.

It can contextualise your tests within a slim library automatically, because the simple act of using slut means you have a slim environment setup. There is a simple comment marker to use at the beginning and end of any slim library which will allow slut to load that library just before starting its evaluation.

Doing so will then force a load of all dependencies of this library. This frees you from having to deal with this very complex situation.

# Setting Up an Init

In many cases, you need to setup some general initialisation before running some unit tests.

The reasons vary wildly, but you may need to reset some files, initialize a server, run some data generating code, whatever, the slut init system is where you do this.

You simply add one or more init commands to the source being scanned and these will be accumulated as one big task.

Before the system performs any unit test, it will run ALL of the init code Exactly Once.

It cannot, ever, run it twice, even if you run your tests over and over or separate them in different sets of tagged stages.

The only way an init can be run again, is if you scan a file (possibly the same one). In this case, the init is accumulated again and will be run again before any new test is performed.

Here is an example of an init block:

; test-init [main-buffer: make string! 10’000 ]

It is important to understand that the init block can be separated in many chunks throughout your source. You do not have to accumulate it all at the beginning of your script. This means you can just specify the initialization you need for a subset of your code in different sections of you code, making it much more local to the problem.

This improves contextualisation and facilitates maintenance of your tests when the source changes.

For example, some init may not be needed anymore. This will be hard to remember and cleanup if the init block is 2000 lines above in some totally unrelated piece of code.

# Adding A Test

We’ve seen a few examples of tests so far, but here we’ll start explaining how they function.

# Test group

# Preambles

Tests can reuse a block of code which is re-executed just before each test is run, it can easily be used to do “per test” initialisation

test-preamble 'pr-str [ test-string: copy "TADAM" ]

This preamble will reset the test-string at EACH test, because we are copying a new instance of it.

To use a preamble, add an extra block before your tests with the name of one or more preamble labels defined prior to the test.

Test [ mytest ][ pr-str ] [ #”T” = take test-string ]

Note that the preamble block CANNOT be empty. If you specify an empty block it will assume it is the test itself, and the test will be skipped and will NOT raise an error.

# Testing for valid error results

Sometimes we must enforce that a function will return a specific error type, it this case use the single test format and add an issue! between the ‘test keyword and the beginning of the test spec, ex:

; test #400 [ err-test ] [ 0 / 0 ]

; test #000 [ err-test ] [ 0 / 0 ]

The number can be the exact error code from the related error, or it can be #000 if you want to support ANY error.

You can ALSO (optionally) add an extra lit-word to support a more specific error which is the ‘id property of the error in question, ex:

; test #400 ‘zero-divide [ err-test ] [ 0 / 0 ]

The above tests will return OK, since you are EXPECTING an error to be generated.

# Running tests

To run tests, simply use the do-tests function, it will launch all extracted tests so far, and return a compilation of all test results.

>> results: slut/do-tests

>> probe results

make object! [

errors?: 0

failed?: 0

succeeded?: 2

total: 2

labels: none

report: [

-------------------------

- %/S/dev/tests/test-slut-data.r

-------------------------

2 "#1" [mytest] none OK

3 "#2" [err-test] none OK

]

]

# Choosing tests to run using filters

The do-tests function has two filters which allow you to chose which tests to run.

**/all-of:** can be used to select unit tests which have multiple tags, ex:

slut/do-tests/all-of [string limits]

Here, we will only run tests which are tagged with both ‘string and ‘limits .

**/any-of :** can be used to select unit test which match any given tag, in no particular order, ex:

slut/do-tests/any-of [string limits]

Here, we will run all tests which are tagged with either ‘string or ‘limits .

# Generating useful results in tests.

Reference

# Comment format

## Inline slut comment format

# Init

# Test

# Test-group

Interactive testing in the REBOL command-prompt

extract

reload

retest

TO DO

# In-line Block format

# Store test results to disk

# JUNIT format result

# Diff module