User Manual for the cl-naive-tests library

Phil Marneweck

Tue Aug 16 17:39:51 CEST 2022

Contents

1 cl-naive-tests

A test framework that is not explicitly based on any of the mainstream popular testing frameworks for other languages. It has a very simple API, so the learning curve should be considerably less than for most other frameworks.

The reason for its simplicity is that it focuses on managing and reporting on tests. It does not try to introduce a testing syntax/language. Even in managing and reporting it tries to not be prescriptive and only implements the bare functional minimum.

It is also designed to work well with gitlab CI out of the box and still be simple! Have a look at .gitlab-ci.yml in the project root folder and for more on gitlab CI for list go to sbcl-docker-gitlab-ci to see how its possible.

1.1 Testsuites result plists

The testsuites results are collected into a plist with the following keys:

key	description	value or type
`:name`	The name of the testsuites	"suites"
`:disabled`	total number of disabled testcases	0
`:skipped`	total number of skipped testscases	0
`:errors`	total number of errors in testcases	integer
`:failures`	total number of failed testcases	integer
`:tests`	total number of testcases ran	integer
`:time`	total duration of the whole suites, in seconds	integer
`:suites`	a list of testsuite results.	plist

Currently there's no provision to skip or disable a testcase, so these counters are always 0.

The testsuite results are collected into a plist with the following keys:

key	description	value or type
`:identifier`	A symbol identifying the testsuite	symbol
`:tests`	number of testcases ran	integer
`:errors`	number of errors in testcases	integer
`:failures`	number of failed testcases	integer
`:disabled`	number of disabled testcases	0
`:skipped`	number of skipped testcases	0
`:package`	the name of the package that is tested	string
`:time`	duration of the suite, in seconds	integer
`:timestamp`	ISO-8601 timestamp of the start of the testsuite	string
`:testcases`	a list of testcase results	plist

The testcase results are collected into a plist with the following keys:

Table 1: Testcase result plist

key	description	value or type
`:identifier`	A symbol identifying the testsuite	symbol
`:info`	A human readable description of the testcase	string
`:equal`	The name of the function used to compare the actual results with the expected results	symbol
`:test-func`	The name of the function used to process the test results	symbol
`:test-data`	Data used by the `TEST-FUNC`	t
`:expected`	The expected result value	t
`:expression`	The source expression that is tested	t
`:actual`	The actual result value obtained by evaluating the `EXPRESSION`	t
`:error`	The error condition if the evaluation of `EXPRESSION` signaled one.	error
`:sysout`	A string collecting the **standard-output** during the testcase.	string
`:syserr`	A string collecting the **error-output** and **trace-output** during the testcase.	string
`:result`	Same as `:FAILURE-TYPE‘`	keyword
`:failure-type`	A keyword indicating the result of the testcase: `:SUCCESS` `:FAILURE` `:ERROR` or something else.	keyword

2 API

2.1 [variable] `*debug*`

If this generalized-boolean variable is set to true, then errors during the execution of a test will invoke the debugger. Otherwise, errors are merely accounted and the testsuite goes on.

2.2 [variable] `*verbose*`

When this generalized-boolean variable is set to true, the `:TEXT` result formatter reports also successful tests. Otherwise only the failed or error tests are reported.

2.3 [variable] `*suites-results*`

Test results of the last testsuites ran are stored here. This variable is reset for each `RUN`.

2.4 [variable] `*junit-no-properties*`

Set this variable to true to disable the **junit** formatter to issue the properties (which are local to the host). Used to produce reproducible test cases.

2.5 [macro] `testsuite (identifier &body body)`

Defines a `TESTSUITE`.

The `IDENTIFIER` is a symbol identifying the testsuite. (It is used as `BLOCK` name which can be used to cut short the testsuite by calling (return-from `/identifier/`) from the `BODY`).

The `BODY` is a list of lisp forms or `TESTCASE` forms.

The results of the `TESTCASE` are collected as results of the `TESTSUITE`.

2.6 [macro] `testcase (identifier &key test-func test-data (equal 'equal) expected actual info)`

The `TESTCASE` macro evaluates the `ACTUAL` expression and compare its result with the `EXPECTED` expression.

The comparison is done either with the `TEST-FUNC` if provided or with the `EQUAL` function. If the comparison returns true, the test is successful, otherwise it's a failure. Test runs the test func returning a plist with information about the result and test.

A plist containing the test info and results is returned. It should be used in the lambda that is registered with register-test.

IDENTIFIER must be a symbol.

TEST-FUNC is the function that is run to determine the result of the test. If none is supplied, then the EQUAL function is used to compare the EXPECTED and the ACTUAL values. The TEST-FUNC returns a failure-type: T or :SUCCESS in case of success, NIL or :FAILURE in case of failure, or some other keyword if the test wasn't run successfully, but this shouldn't be counted as a failure. The TEST-FUNC is given two arguments: a plist containing :TEST-DATA :EXPECTED :ACTUAL :EXPRESSION, and the INFO. (:ACTUAL is the value of the :EXPRESSION that is tested; if an error is signaled, it's passed in :ERROR).

TEST-DATA is a convenient place to store data the test relies on, this can be used during the test and later in reporting on test results. You can put what ever you want to in it.

INFO is a string to be read by the human that is digging into the tests results, describing the test.

2.6.1 Example

```
(testsuite division
  (testcase (division non-zero-dividend)
    :equal '=
    :expected 3/2
    :actual (/ 3 2)
    :info \"Integer division by non-zero, giving a ratio.\")

  (testcase (division by-zero)
    :test-func (lambda (result info)
 (let ((actual (getf result :actual)))
   (cond ((eql (getf result :expected)
       :success)
  ((typep actual 'error)
   (setf (aref (getf result :test-data) 0)
 (list 'unexpected-error (type-of actual)))
   :failure)
  (t
   (setf (aref (getf result :test-data) 0)
 (list 'unexpected-result actual))
   :failure)))))
    :test-data (vector nil)
    :expected 'division-by-zero
    :actual (handler-case (/ 3 0)
      (:no-error (result) result)
      (division-by-zero () 'division-by-zero)
      (error (err) err))
    :info \"Integer division by zero, giving a DIVISION-BY-ZERO error.\"))
```

2.7 [function] run (&key (suites *test-suites*) keep-stats-p ((:debug *debug*) nil) (name "suites"))

SUITES is an equal hash-table mapping testsuite identifiers to functions taking the testsuite identifier as argument, and returning a testsuite result plist.

Runs all tests passed in or all tests registered.

Pass true to `KEEP-STATS-P` to calculate the statistics. Statistics can be calculated during a test run, but the default is to use `STATISTICS` after a test run to calculate stats.

Pass true to `DEBUG` so that upon error in a testcase, the debugger is invoked.

The name of the testsuites can be specified with the `NAME` parameter.

Returns:

- the testsuites result plist;
- the statistics.

2.8 [function] `report (&optional (suites-results *suites-results*))`

Reports on the pass or failure of the results set over all. This does not do any pretty printing because it needs to be machine readable. If you want pretty reporting look at `WRITE-RESULTS` or `FORMAT-RESULTS` or do your own.

The output format is:

```
Passed:       <integer>
Failed:       <integer>
```

If some tests have other statuses, then additionnal lines with the format:

```
<status>:     <integer>
```

are issued.

Returns:

- a boolean indicating whether all testcases were successful;
- the list of successful testcases;
- the list of failed testcases;
- the list of other testcases.

2.9 [function] `find-testcase (testcase-identifier suites-results &key test)`

Finds a testcase in the `SUITES-RESULTS`.

2.10 [function] `calc-stats (result &optional (stats (make-hash-table :test #'equalp)))`

Calculates stats. Stats are simple counts of tests, passed and failed per level. Stats are stored in a hashtable per identifier level, which makes it easy to get to in format-results if needed.

2.11 [function] `statistics (results)`

Can be used to calculate statistics post tests if **keep-stats-p** was nil.

2.12 [generic fucntion] `format-results (format results)`

format-results (format suites-results)

Formats the `SUITES-RESULTS` according to `FORMAT`. The default method just outputs the results using lisp format string.

format-results ((format (eql :junit)) suites-results)

Formats the `SUITES-RESULTS` as Junit XML; junits only allows 3 levels nl. suites, suite and testcase.

format-results ((format (eql :text)) suites-results)

Formats the `SUITES-RESULTS` as text; can be used to output them on then REPL. When `*VERBOSE*` is `NIL`, the successful testcases are not reported.

2.13 [function] `write-results (suites-results &key (stream *standard-output*) format)`

Writes the `SUITES-RESULTS` to the `STREAM`. Formats the results using `FORMAT-RESULT`.

2.14 [function] `save-results (suites-results &key (file "results.log") format)`

Saves the `SUITES-RESULTS` to the `FILE`. Formats the results using `FORMAT-RESULT`. This can be used to produce files that could be used by some thing like gitlab CI.