

ERT – Emacs Lisp Regression Testing

Description	Keystroke	Function	Note
Using ERT		If you write Emacs Lisp code, writing unit test code normally helps increase code quality and maintainability.	For `ERT - Emacs Lisp`
o Help & Customization		ERT is a simple, yet powerful, environment to write test you can run from the command line and interactively inside Emacs with the ability to debug failing code.	
• Run tests interactively		• ERT is part of Emacs standard distribution since Emacs 24 .	
• Byte Compile & run tests		• ERT provides a command to run tests that have been written using the ERT macros.	
• *ert* buffer commands		• Tests are normally written inside separate .el files, with names using the same prefix as the file being tested and also often placed inside a test directory.	
o command line tests			
• Emacs Lisp test code facilities			
o ERT Reference		To run the test:	
• ERT Manuals	1. Load the Emacs Lisp file that defines the tests. You can also visit and byte compile the file (with PEL: <code><f12> c b</code>).		
• ERT Tools	• With PEL you can use the <code><f12> 1 f</code> or <code><f12> 1 v</code> key bindings to load-file and <code>pel-load-visited-file</code> commands.		
• Articles on ERT	• See `ERT - Emacs Lisp` for more info.		
• Other unit testing tools	2. Type <code>M-x ert RET t RET</code> .		
• Emacs Lisp Concepts	• That executes the ert command described below and provides the selector <code>t</code> which means running every test.		
• CI/CD Test on Github	• You can also use other regular expression that identify the names of the test functions to run:		
	• for example use <code>^foo-</code> to run all tests that have a name that begins with foo-.		
• Emacs Lisp language extension.		You can extend ERT testing capabilities with several external packages and libraries. PEL support the automatic installation and setup of the following ones. Just turn on the corresponding PEL user-option to install and activate them.	
• Mocking Libraries			
• Test coverage highlighting	 noflet (PEL uses my fork)	 pel-use-noflet	Locally override a function in the manner of <code>flet</code> but with access to the original function though the <code>this-fn</code> symbol. PEL uses my fork that integrates all fixes provided by many.
			 Unless you need <code>noflet this-fn</code> feature, you can use <code>cl-lib cl-letf</code> instead of <code>noflet</code> .
• ERT test runner	 el-mock	 pel-use-el-mock	A tiny Mock and Sti=ub framework for Emacs Lisp.
• Alternative test frameworks	 el-spy	 pel-use-el-spy	A small mocking framework for Emacs Lisp that also support spy and proxy.
	 mocker	 pel-use-mocker	A simple mocking framework for Emacs.
	 coverlay	 pel-use-coverlay	Test coverage overlay. Supports LCOV format (which Coveralls uses).
	 coverage	 pel-use-coverage	An older test coverage system, which depends on even older ert-expectations.
	 ert-expectations	 pel-use-ert-expectations	A nice wrapper around ert to create test simply. However the code is old and would need updates.
	 test-cover-mark	 pel-use-testcover-mark	Mark the whole line of code not completely tested. Extends what builtin <code>testcover</code> marks.
	 ert-runner	 pel-use-ert-runner	ERT runner to run ERT test in CI/CD systems. Often used with Cask to drive ERT tests.
	 buttercup	 pel-use-buttercup	Behavior-Driven Emacs Lisp Testing. An alternative to ERT that addresses several limitations of ERT.
Last updated on:	2026-01-15	<f12> ERT commands key bindings refer to keys typed inside an Emacs Lisp buffer. See more info in `ERT - Emacs Lisp` .	
Open this PDF file. See also: Help/Info	<code><f12> C-t <f1></code>	(<code>pel-help-pdf</code> &optional OPEN-WEB-PAGE)	Open the `ERT - ERT` local PDF. If the prefix argument (like <code>C-u</code> or <code>M--</code>) is used, then it opens the remote GitHub hosted raw PDF instead. If the <code>pel-flip-help-pdf-arg</code> user-option is set it's the other way around.
Customize PEL ERT support	<code><f12> <f2></code>	(<code>pel-customize-pel</code> &optional OTHER-WINDOW)	Customize PEL ERT support. • If OTHER-WINDOW is non-nil (use <code>C-u</code>), display in another window.
Customize Emacs ERT support	<code><f12> <f3></code>	(<code>pel-customize-library</code> &optional OTHER-WINDOW)	Customize Emacs ERT support: overlay, testcover, testcover-mark-line • If OTHER-WINDOW is non-nil (use <code>C-u</code>), display in another window.
Byte Compile and run all Emacs Lisp tests	<code><f12> C-t C-t</code>	(<code>pel-run-ert</code>)	Byte compile and run ERT test on current `ERT - Emacs Lisp` buffer. • Prompts if the buffer needs to be saved first.
Run test interactively	<code><f12> C-t C-r</code>	(<code>ert SELECTOR &optional OUTPUT-BUFFER-NAME MESSAGE-FN</code>)	Run the tests specified by SELECTOR and display the results in a buffer. • SELECTOR works as described in 'ert-select-tests'. (Use <code>t</code> to run all tests, or name the test to execute.)
Extra 2 args, only available in Emacs < 29			• By default, the results are stored inside the *ert* buffer, opened in ERT-Results mode.  ert does not save and byte compile the buffer before attempting to run the test. It must be done before. With PEL, use <code>pel-run-test</code> . See below.
ert buffer commands		In the *ert* buffer, the result of each test appears as a coloured character button:	The OUTPUT-BUFFER-NAME and MESSAGE-FN optional arguments were available until Emacs 29 . Starting at Emacs 29 they are no longer available. OUTPUT-BUFFER-NAME and MESSAGE-FN should normally be nil; they are used for automated self-tests and specify which buffer to use and how to display message.
(Available from the *ert* buffer opened once art runs a test.)		• A period . if the test passed. Coloured in green. • A red upper case F if the test failed. • A green lower case f if the test failed, but was expected to fail (the <code>ert-deftest</code> has a :expected-result :failed). Used to identify bugs not yet fixed. • A red upper case P indicating a test passed but was expected to fail. • While at a button you can issue other commands. The following single key commands are available when point is inside the buffer's window:	
Move to test	TAB	(<code>forward-button N &optional WRAP DISPLAY-MESSAGE</code>)	Move point to the next test result button.
Re-run test	r	(<code>ert-results-rerun-test-at-point</code>)	Re-run the same test
Jump to test source	.	(<code>ert-results-find-test-at-point-other-window</code>)	Jump to the source code of the test (in another window)
List should forms	l	(<code>ert-results-pop-to-should-forms-for-test-at-point</code>)	Shows the list of all <code>should</code> forms executed during the test before it failed
View backtrace	b	(<code>ert-results-pop-to-backtrace-for-test-at-point</code>)	View the backtrace for the failed test at point
Re-run test with debugging	d	(<code>ert-results-rerun-test-at-point-debugging-errors</code>)	Re-run the same test with debugging enabled
Show messages	m	(<code>ert-results-pop-to-messages-for-test-at-point</code>)	Show what messages were printed before the test failed
Toggle condition printing	L	(<code>ert-results-toggle-printer-limits-for-test-at-point</code>)	Toggle how much of the condition to print for the test at point.
Delete obsolete tests	D	(<code>ert-delete-test TEST-NAME</code>)	Delete obsolete tests (test whose code might have changed)
Re-run all tests	R	(<code>ert-results-rerun-all-tests</code>)	Re-run all tests, using the same selector
Move to the next test result	n	(<code>ert-results-next-test</code>)	Move to the next test results.
Move to previous test result	p	(<code>ert-results-previous-test</code>)	Move to the previous test results.
Jump between summary and result	j	(<code>ert-results-jump-between-summary-and-result</code>)	Jump between test result and summary.
			By positioning point on the test result character (., F or f) and then typing j point will move to the test summary (and will create one if the test passed). From the summary you can easily press RET to move point to the source code of the test (in another window).
Show help for test	h	(<code>ert-results-describe-test-at-point</code>)	Get help for the test corresponding to the test result character (or test summary) at point.
Describe available commands	?	(<code>describe-mode &optional BUFFER</code>)	Describe mode (and these commands)
Quit - close window	q	(<code>quit-window &optional KILL WINDOW</code>)	Quit window - dismiss the *ert* popup buffer window.

Description	Keystroke	Function	Note
Run Tests in Batch Mode		To execute tests from the command line, use Emacs in batch mode to load the specific tests and run them using one of the following ERT functions. • For example: <code>emacs -batch -l ert -l my-tests.el -f ert-run-tests-batch-and-exit</code>	
Run batch test	(<code>ert-run-tests-batch</code> &optional SELECTOR)	• Run the tests specified by SELECTOR, printing results to the terminal. • SELECTOR works as described in ‘ert-select-tests’, except if SELECTOR is nil, in which case all tests rather than none will be run; this makes the command line “ <code>emacs -batch -l my-tests.el -f ert-run-tests-batch-and-exit</code> ” useful.	Returns the stats object.
Run batch tests and exit	(<code>ert-run-tests-batch-and-exit</code> &optional SELECTOR)	Like ‘ <code>ert-run-tests-batch</code> ’, but exits Emacs when done. • The exit status will be 0 if all test results were as expected, 1 on unexpected results, or 2 if the tool detected an error outside of the tests (e.g. invalid SELECTOR or bug in the code that runs the tests).	
Emacs Lisp Test Code		Test code is written using forms that use the <code>ert-deftest</code> macro. See the following macro descriptions	
Define a test	(<code>ert-deftest</code> NAME () [DOCSTRING] [:expected-result RESULT-TYPE] [:tags '(TAG...)] BODY...)	Define NAME (a symbol) as a test. • BODY is evaluated as a ‘progn’ when the test is run. It should signal a condition on failure or just return if the test passes. • ‘ <code>should</code> ’, ‘ <code>should-not</code> ’, ‘ <code>should-error</code> ’ and ‘ <code>skip-unless</code> ’ are useful for assertions in BODY. • Use ‘ <code>ert</code> ’ to run tests interactively. • Tests that are expected to fail can be marked as such using :expected-result. See ‘ <code>ert-test-result-type-p</code> ’ for a description of valid values for RESULT-TYPE.	
The <code>should</code> macro	(<code>should</code> FORM)	Evaluate FORM. If it returns nil, abort the current test as failed. • Returns the value of FORM.	
The <code>should-not</code> macro	(<code>should-not</code> FORM)	Evaluate FORM. If it returns non-nil, abort the current test as failed. • Returns nil.	
The <code>should-error</code> macro	(<code>should-error</code> FORM &rest KEYS &key TYPE EXCLUDE-SUBTYPES)	Evaluate FORM and check that it signals an error. • The error signaled needs to match TYPE. TYPE should be a list of condition names. (It can also be a non-nil symbol, which is equivalent to a singleton list containing that symbol.) If EXCLUDE-SUBTYPES is nil, the error matches TYPE if one of its condition names is an element of TYPE. If EXCLUDE-SUBTYPES is non-nil, the error matches TYPE if it is an element of TYPE. • If the error matches, returns (ERROR-SYMBOL . DATA) from the error. If not, or if no error was signaled, abort the test as failed.	
The <code>skip-unless</code> macro	(<code>skip-unless</code> CONDITION)	Skip the current <code>ert-deftest</code> defined test unless CONDITION is non-nil. • Normally used to skip tests when the environment does not provide the necessary conditions for a valid test.	
The <code>:expected-result</code> tag	:expected-result • :failed • :passed	Tag tests of lesser importances that are expected to fail potentially under some conditions. It can also be used to silence the report of a bug while leaving in the test. • See the documentation .	

Emacs Lisp Testing – References

Topic & Link	Description
ERT Manuals	
ERT : Emacs Lisp Regression Testing	ERT Manual, part of Emacs.
Test Coverage – Emacs Lisp	Test coverage section of the GNU Emacs Lisp manual
ERT Tools Repositories and Files	
pel-ert.el	pel-ert.el defines a set of equality predicates that accept extra arguments for the sole purpose of having them shown in the ERT report of a failed test. To see a set of test environment variables in the test report just pass them as extra arguments to these equality predicates.
testcover.el source	Source of the test coverage support
overseer.el @ GitHub	
ert-runner @ GitHub	
Emacs Lisp Mock @ EmacsWiki	The original location of that mock library that can be used with ert.
El mock @ GitHub	The new location for el-mock.el
emacs-noflet @ GitHub	External package that provides the noflet macro which can be used to re-define functions locally. Can be used in ERT testing. I created my own fork of this and integrated most long standing pull requests. See pierre-rouleau / emacs-noflet .
Articles/Blogs on ERT	Interesting articles to read before writing Emacs Lisp ERT testing code.
Quick intro to ert testing	A quick overview of ERT based Emacs Lisp testing.
Elisp Unit Testing with ERT	Quick overview of ERT in an August 2012 blog written by Chris Wellons. • Since then the cl.el library was replaced by the cl-lib.el and the flet function was deprecated to cl-flet, but aside from these small items the description is still valid.
flet, cl-flet, cl-letf and noflet	<ul style="list-style-type: none"> • Make Flet Great Again : Another great post from Chris Wellons. <ul style="list-style-type: none"> • Describes the new cl-flet and cl-letf, and how to use cl-letf to create Ert test that modify the behaviour of called functions. • Understanding letf and how it replaces flet, by Arthur Malabarba <ul style="list-style-type: none"> • Describes cl-letf and how it can replace the old flet.
ERT: Emacs Lisp Regression Testing	A nice description of the ERT features and techniques given at the NYC meetup. Describes: <ul style="list-style-type: none"> • How to write a simple test, how to run the test and use the *ert* buffer commands. • How to test expected errors with should-error
Continuous integration and code coverage for Emacs packages with Travis and Coveralls	An overview of automated Emacs Lisp unit testing with coverage measurement from Sacha Chua blog.
Reddit discussion: Testing in Emacs	Interesting discussion on testing under Emacs. Leads to several interesting projects: <ul style="list-style-type: none"> • test-cockpit for testing code written in several programming languages under Emacs <ul style="list-style-type: none"> • dape : Debug Adapter Protocol for Emacs • stan-mode which uses buttercup for testing.
Other Unit Testing Support	Other packages and libraries for testing are not listed at the top of this page and currently not explicitly supported by PEL. Some are listed here.
• Unit Testing on EmacsWiki	Provides a list of alternatives to ERT with other testing-related information.
• emacs-test-simple @ GitHub	An Alternative to ERT developed by debugger expert R.Bernstein (see also here).
• undercover.el	A test coverage library for Emacs. Does not support byte-compilation nor circular objects. Meant to be used with Cask , Eask or Eldev .
Shell based unit test tools	
cram	A simple test system to test shell commands. cram is a command line tool that takes cram scripts stored in .t files. <ul style="list-style-type: none"> • PEL supports the cram-mode when pel-use-cram-mode is turned on.
Emacs Lisp Concepts	Several Emacs Lisp concepts are useful when writing ERT test and mock ups.
Scoping Rules for Variable Bindings	When writing Emacs Lisp code and test with mockup in particular its important to fully understand the concepts of dynamic and lexical binding in Emacs Lisp. ⚠ Be aware that starting with version 27 of Emacs Lisp lexical binding is the default while dynamic binding was the default in previous versions.
Github Continuous Integration	
• With GitHub Actions	<ul style="list-style-type: none"> • Workflow syntax for GitHub Actions • GitHub Actions Runner Images : select the os environment • Setup Emacs for Github Actions : select the Emacs version
Running ERT tests suite with Cask and ert-runner	<ul style="list-style-type: none"> • To run all tests: <code>cask exec ert-runner</code> • To run a specific test: <code>cask exec ert-runner -p "name-of-specific-test"</code> • To run tests that are tagged fast or important: <code>cask exec ert-runner -t fast,important</code> • To run tests not tagged 'network': <code>cask exec ert-runner -t '! network'</code> • Combining options: <code>cask exec ert-runner test/moda-test.el -p "specific-functionality"</code>