

EUnit

Copyright © 2008-2023 Ericsson AB, All Rights Reserved
EUnit 2.4
May 9, 2023

Copyright © 2008-2023 Ericsson AB, All Rights Reserved
Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0 Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License. The Initial Developer of the Original Code is Ericsson AB. Ericsson AB, All Rights Reserved.

May 9, 2023

1 EUnit User's Guide

The **EUnit** application contains modules with support for unit testing.

1.1 EUnit - a Lightweight Unit Testing Framework for Erlang

EUnit is a unit testing framework for Erlang. It is very powerful and flexible, is easy to use, and has small syntactical overhead.

- *Unit testing*
- *Terminology*
- *Getting started*
- *EUnit macros*
- *EUnit test representation*

EUnit builds on ideas from the family of unit testing frameworks for Object Oriented languages that originated with JUnit by Beck and Gamma (and Beck's previous framework SUnit for Smalltalk). However, EUnit uses techniques more adapted to functional and concurrent programming, and is typically less verbose than its relatives.

Although EUnit uses many preprocessor macros, they have been designed to be as nonintrusive as possible, and should not cause conflicts with existing code. Adding EUnit tests to a module should thus not normally require changing existing code. Furthermore, tests that only exercise the exported functions of a module can always be placed in a completely separate module, avoiding any conflicts entirely.

1.1.1 Unit testing

Unit Testing is testing of individual program "units" in relative isolation. There is no particular size requirement: a unit can be a function, a module, a process, or even a whole application, but the most typical testing units are individual functions or modules. In order to test a unit, you specify a set of individual tests, set up the smallest necessary environment for being able to run those tests (often, you don't need to do any setup at all), you run the tests and collect the results, and finally you do any necessary cleanup so that the test can be run again later. A Unit Testing Framework tries to help you in each stage of this process, so that it is easy to write tests, easy to run them, and easy to see which tests failed (so you can fix the bugs).

Advantages of unit testing

Reduces the risks of changing the program

> Most programs will be modified during their lifetime: bugs will be fixed, features will be added, optimizations may become necessary, or the code will need to be refactored or cleaned up in other ways to make it easier to work with. But every change to a working program is a risk of introducing new bugs - or reintroducing bugs that had previously been fixed. Having a set of unit tests that you can run with very little effort makes it easy to know that the code still works as it should (this use is called **regression testing**; see *Terminology*). This goes a long way to reduce the resistance to changing and refactoring code.

Helps guide and speed up the development process

> By focusing on getting the code to pass the tests, the programmer can become more productive, not overspecify or get lost in premature optimizations, and create code that is correct from the very beginning (so-called **test-driven development**; see *Terminology*).

Helps separate interface from implementation

When writing tests, the programmer may discover dependencies (in order to get the tests to run) that ought not to be there, and which need to be abstracted away to get a cleaner design. This helps eliminate bad dependencies before they spread throughout the code.

Makes component integration easier

By testing in a bottom-up fashion, beginning with the smallest program units and creating a confidence in that they work as they should, it becomes easier to test that a higher-level component, consisting of several such units, also behaves according to specification (known as **integration testing**; see *Terminology*).

Is self-documenting

The tests can be read as documentation, typically showing both examples of correct and incorrect usage, along with the expected consequences.

1.1.2 Terminology

Unit testing

Testing that a program unit behaves as it is supposed to do (in itself), according to its specifications. Unit tests have an important function as regression tests, when the program later is modified for some reason, since they check that the program still behaves according to specification.

Regression testing

Running a set of tests after making changes to a program, to check that the program behaves as it did before the changes (except, of course, for any intentional changes in behaviour). Unit tests are important as regression tests, but regression testing can involve more than just unit testing, and may also test behaviour that might not be part of the normal specification (such as bug-for-bug-compatibility).

Integration testing

Testing that a number of individually developed program units (assumed to already have been separately unit tested) work together as expected. Depending on the system being developed, integration testing may be as simple as "just another level of unit testing", but might also involve other kinds of tests (compare **system testing**).

System testing

Testing that a complete system behaves according to its specification. Specifically, system testing should not require knowing any details about the implementation. It typically involves testing many different aspects of the system behaviour apart from the basic functionality, such as performance, usability, and reliability.

Test-driven development

A program development technique where you continuously write tests **before** you implement the code that is supposed to pass those tests. This can help you focus on solving the right problems, and not make a more complicated implementation than necessary, by letting the unit tests determine when a program is "done": if it fulfils its specifications, there is no need to keep adding functionality.

Mock object

Sometimes, testing some unit A (e.g., a function) requires that it collaborates somehow with some other unit B (perhaps being passed as an argument, or by reference) - but B has not been implemented yet. A "mock object" - an object which, for the purposes of testing A, looks and behaves like a real B - might then be used instead. (This is of course only useful if it would be significantly more work to implement a real B than to create a mock object.)

Test case

A single, well-defined test, that somehow can be uniquely identified. When executed, the test case either **passes** or **fails**; the test report should identify exactly which test cases failed.

Test suite

A collection of test cases, generally with a specific, common target for testing, such as a single function, module, or subsystem. A test suite may also be recursively composed by smaller test suites.

1.1.3 Getting started

- *Including the EUnit header file*
- *Writing simple test functions*
- *Running EUnit*
- *Writing test generating functions*
- *An example*
- *Disabling testing*
- *Avoiding compile-time dependency on EUnit*

Including the EUnit header file

The simplest way to use EUnit in an Erlang module is to add the following line at the beginning of the module (after the `-module` declaration, but before any function definitions):

```
-include_lib("eunit/include/eunit.hrl").
```

This will have the following effect:

- Creates an exported function `test()` (unless testing is turned off, and the module does not already contain a test() function), that can be used to run all the unit tests defined in the module
- Causes all functions whose names match `..._test()` or `..._test_()` to be automatically exported from the module (unless testing is turned off, or the `EUNIT_NOAUTO` macro is defined)
- Makes all the preprocessor macros of EUnit available, to help writing tests

Note: For `-include_lib(...)` to work, the Erlang module search path **must** contain a directory whose name ends in `eunit/ebin` (pointing to the `ebin` subdirectory of the EUnit installation directory). If EUnit is installed as `lib/eunit` under your Erlang/OTP system directory, its `ebin` subdirectory will be automatically added to the search path when Erlang starts. Otherwise, you need to add the directory explicitly, by passing a `-pa` flag to the `erl` or `erlc` command. For example, a Makefile could contain the following action for compiling `.erl` files:

```
erlc -pa "path/to/eunit/ebin" $(ERL_COMPILE_FLAGS) -o$(EBIN) $<
```

or if you want Eunit to always be available when you run Erlang interactively, you can add a line like the following to your `$HOME/.erlang` file:

```
code:add_path("/path/to/eunit/ebin").
```

Writing simple test functions

The EUnit framework makes it extremely easy to write unit tests in Erlang. There are a few different ways of writing them, though, so we start with the simplest:

A function with a name ending in `..._test()` is recognized by EUnit as a simple test function - it takes no arguments, and its execution either succeeds (returning some arbitrary value that EUnit will throw away), or fails by throwing an exception of some kind (or by not terminating, in which case it will be aborted after a while).

An example of a simple test function could be the following:

```
reverse_test() -> lists:reverse([1,2,3]).
```

This just tests that the function `lists:reverse(List)` does not crash when `List` is `[1,2,3]`. It is not a great test, but many people write simple functions like this one to test the basic functionality of their code, and those tests can be used directly by EUnit, without changes, as long as their function names match.

Use exceptions to signal failure To write more interesting tests, we need to make them crash (throw an exception) when they don't get the result they expect. A simple way of doing this is to use pattern matching with `=`, as in the following examples:

```
reverse_nil_test() -> [] = lists:reverse([]).
reverse_one_test() -> [1] = lists:reverse([1]).
reverse_two_test() -> [2,1] = lists:reverse([1,2]).
```

If there was some bug in `lists:reverse/1` that made it return something other than `[2,1]` when it got `[1,2]` as input, then the last test above would throw a `badmatch` error. The first two (we assume they do not get a `badmatch`) would simply return `[]` and `[1]`, respectively, so both succeed. (Note that EUnit is not psychic: if you write a test that returns a value, even if it is the wrong value, EUnit will consider it a success. You must make sure that the test is written so that it causes a crash if the result is not what it should be.)

Using assert macros If you want to use Boolean operators for your tests, the `assert` macro comes in handy (see *EUnit macros* for details):

```
length_test() -> ?assert(length([1,2,3]) =:= 3).
```

The `?assert(Expression)` macro will evaluate `Expression`, and if that does not evaluate to `true`, it will throw an exception; otherwise it just returns `ok`. In the above example, the test will thus fail if the call to `length` does not return 3.

Running EUnit

If you have added the declaration `-include_lib("eunit/include/eunit.hrl")` to your module, as described above, you only need to compile the module, and run the automatically exported function `test()`. For example, if your module was named `m`, then calling `m:test()` will run EUnit on all the tests defined in the module. You do not need to write `-export` declarations for the test functions. This is all done by magic.

You can also use the function *eunit:test/1* to run arbitrary tests, for example to try out some more advanced test descriptors (see *EUnit test representation*). For example, running `eunit:test(m)` does the same thing as the auto-generated function `m:test()`, while `eunit:test({inparallel, m})` runs the same test cases but executes them all in parallel.

Putting tests in separate modules

If you want to separate your test code from your normal code (at least for testing the exported functions), you can simply write the test functions in a module named `m_tests` (note: not `m_test`), if your module is named `m`. Then, whenever you ask EUnit to test the module `m`, it will also look for the module `m_tests` and run those tests as well. See `ModuleName` in the section *Primitives* for details.

EUnit captures standard output

If your test code writes to the standard output, you may be surprised to see that the text does not appear on the console when the tests are running. This is because EUnit captures all standard output from test functions (this also includes setup and cleanup functions, but not generator functions), so that it can be included in the test report if errors occur. To bypass EUnit and print text directly to the console while testing, you can write to the `user` output stream, as in `io:format(user, "~w", [Term])`. The recommended way of doing this is to use the EUnit *Debugging macros*, which make it much simpler.

Writing test generating functions

A drawback of simple test functions is that you must write a separate function (with a separate name) for each test case. A more compact way of writing tests (and much more flexible, as we shall see), is to write functions that **return** tests, instead of **being** tests.

A function with a name ending in `..._test_()` (note the final underscore) is recognized by EUnit as a **test generator** function. Test generators return a **representation** of a **set of tests** to be executed by EUnit.

Representing a test as data The most basic representation of a test is a single fun-expression that takes no arguments. For example, the following test generator:

```
basic_test_() ->
    fun () -> ?assert(1 + 1 =:= 2) end.
```

will have the same effect as the following simple test:

```
simple_test() ->
    ?assert(1 + 1 =:= 2).
```

(in fact, EUnit will handle all simple tests just like it handles fun-expressions: it will put them in a list, and run them one by one).

Using macros to write tests To make tests more compact and readable, as well as automatically add information about the line number in the source code where a test occurred (and reduce the number of characters you have to type), you can use the `_test` macro (note the initial underscore character), like this:

```
basic_test_() ->
    ?_test(?assert(1 + 1 =:= 2)).
```

The `_test` macro takes any expression (the "body") as argument, and places it within a fun-expression (along with some extra information). The body can be any kind of test expression, just like the body of a simple test function.

Underscore-prefixed macros create test objects But this example can be made even shorter! Most test macros, such as the family of `assert` macros, have a corresponding form with an initial underscore character, which automatically adds a `?_test(...)` wrapper. The above example can then simply be written:

```
basic_test_() ->
    ?_assert(1 + 1 =:= 2).
```

which has exactly the same meaning (note the `_assert` instead of `assert`). You can think of the initial underscore as signalling **test object**.

An example

Sometimes, an example says more than a thousand words. The following small Erlang module shows how EUnit can be used in practice.

```
  -module(fib).
  -export([fib/1]).
  -include_lib("eunit/include/eunit.hrl").

  fib(0) -> 1;
  fib(1) -> 1;
  fib(N) when N > 1 -> fib(N-1) + fib(N-2).

  fib_test_() ->
      [?_assert(fib(0) =:= 1),
?_assert(fib(1) =:= 1),
?_assert(fib(2) =:= 2),
?_assert(fib(3) =:= 3),
?_assert(fib(4) =:= 5),
?_assert(fib(5) =:= 8),
?_assertException(error, function_clause, fib(-1)),
?_assert(fib(31) =:= 2178309)
      ].
```

(Author's note: When I first wrote this example, I happened to write a `*` instead of `+` in the `fib` function. Of course, this showed up immediately when I ran the tests.)

See *EUnit test representation* for a full list of all the ways you can specify test sets in EUnit.

Disabling testing

Testing can be turned off by defining the `NOTEST` macro when compiling, for example as an option to `erlc`, as in:

```
  erlc -DNOTEST my_module.erl
```

or by adding a macro definition to the code, **before the EUnit header file is included**:

```
  -define(NOTEST, 1).
```

(the value is not important, but should typically be 1 or `true`). Note that unless the `EUNIT_NOAUTO` macro is defined, disabling testing will also automatically strip all test functions from the code, except for any that are explicitly declared as exported.

For instance, to use EUnit in your application, but with testing turned off by default, put the following lines in a header file:

```
  -define(NOTEST, true).
  -include_lib("eunit/include/eunit.hrl").
```

and then make sure that every module of your application includes that header file. This means that you have a only a single place to modify in order to change the default setting for testing. To override the `NOTEST` setting without modifying the code, you can define `TEST` in a compiler option, like this:

```
  erlc -DTEST my_module.erl
```

See *Compilation control macros* for details about these macros.

Avoiding compile-time dependency on EUnit

If you are distributing the source code for your application for other people to compile and run, you probably want to ensure that the code compiles even if EUnit is not available. Like the example in the previous section, you can put the following lines in a common header file:

```
  -ifdef(TEST).
  -include_lib("eunit/include/eunit.hrl").
  -endif.
```

and, of course, also make sure that you place all test code that uses EUnit macros within `-ifdef(TEST)` or `-ifdef(EUNIT)` sections.

1.1.4 EUnit macros

Although all the functionality of EUnit is available even without the use of preprocessor macros, the EUnit header file defines a number of such macros in order to make it as easy as possible to write unit tests as compactly as possible and without getting too many details in the way.

Except where explicitly stated, using EUnit macros will never introduce run-time dependencies on the EUnit library code, regardless of whether your code is compiled with testing enabled or disabled.

- *Basic macros*
- *Compilation control macros*
- *Utility macros*
- *Assert macros*
- *Macros for running external commands*
- *Debugging macros*

Basic macros

`_test(Expr)`

Turns `Expr` into a "test object", by wrapping it in a fun-expression and a source line number. Technically, this is the same as `{?LINE, fun () -> (Expr) end}`.

Compilation control macros

`EUNIT`

This macro is always defined to `true` whenever EUnit is enabled at compile time. This is typically used to place testing code within conditional compilation, as in:

```
-ifdef(EUNIT).
    % test code here
    ...
-endif.
```

e.g., to ensure that the code can be compiled without including the EUnit header file, when testing is disabled. See also the macros `TEST` and `NOTEST`.

`EUNIT_NOAUTO`

If this macro is defined, the automatic exporting or stripping of test functions will be disabled.

`TEST`

This macro is always defined (to `true`, unless previously defined by the user to have another value) whenever EUnit is enabled at compile time. This can be used to place testing code within conditional compilation; see also the macros `NOTEST` and `EUNIT`.

For testing code that is strictly dependent on EUnit, it may be preferable to use the `EUNIT` macro for this purpose, while for code that uses more generic testing conventions, using the `TEST` macro may be preferred.

The `TEST` macro can also be used to override the `NOTEST` macro. If `TEST` is defined **before** the EUnit header file is included (even if `NOTEST` is also defined), then the code will be compiled with EUnit enabled.

`NOTEST`

This macro is always defined (to `true`, unless previously defined by the user to have another value) whenever EUnit is **disabled** at compile time. (Compare the `TEST` macro.)

This macro can also be used for conditional compilation, but is more typically used to disable testing: If `NOTEST` is defined **before** the EUnit header file is included, and `TEST` is **not** defined, then the code will be compiled with EUnit disabled. See also *Disabling testing*.

`NOASSERT`

If this macro is defined, the assert macros will have no effect, when testing is also disabled. See *Assert macros*. When testing is enabled, the assert macros are always enabled automatically and cannot be disabled.

`ASSERT`

If this macro is defined, it overrides the NOASSERT macro, forcing the assert macros to always be enabled regardless of other settings.

`NODEBUG`

If this macro is defined, the debugging macros will have no effect. See *Debugging macros*. `NODEBUG` also implies `NOASSERT`, unless testing is enabled.

`DEBUG`

If this macro is defined, it overrides the NODEBUG macro, forcing the debugging macros to be enabled.

Utility macros

The following macros can make tests more compact and readable:

`LET(Var,Arg,Expr)`

Creates a local binding `Var = Arg` in `Expr`. (This is the same as `(fun(Var)->(Expr)end)(Arg)`.) Note that the binding is not exported outside of `Expr`, and that within `Expr`, this binding of `Var` will shadow any binding of `Var` in the surrounding scope.

`IF(Cond,TrueCase,FalseCase)`

Evaluates `TrueCase` if `Cond` evaluates to `true`, or otherwise evaluates `FalseCase` if `Cond` evaluates to `false`. (This is the same as `(case (Cond) of true->(TrueCase); false->(FalseCase) end)`.) Note that it is an error if `Cond` does not yield a boolean value.

Assert macros

(Note that these macros also have corresponding forms which start with an "_" (underscore) character, as in `?_assert(BoolExpr)`, that create a "test object" instead of performing the test immediately. This is equivalent to writing `?_test(assert(BoolExpr))`, etc.)

If the macro `NOASSERT` is defined before the EUnit header file is included, these macros have no effect when testing is also disabled; see *Compilation control macros* for details.

`assert(BoolExpr)`

Evaluates the expression `BoolExpr`, if testing is enabled. Unless the result is `true`, an informative exception will be generated. If there is no exception, the result of the macro expression is the atom `ok`, and the value of `BoolExpr` is discarded. If testing is disabled, the macro will not generate any code except the atom `ok`, and `BoolExpr` will not be evaluated.

Typical usage:

```
?assert(f(X, Y) =:= [])
```

The `assert` macro can be used anywhere in a program, not just in unit tests, to check pre/postconditions and invariants. For example:

```
some_recursive_function(X, Y, Z) ->
    ?assert(X + Y > Z),
    ...
```

`assertNot(BoolExpr)`

Equivalent to `assert(not (BoolExpr))`.

`assertMatch(GuardedPattern, Expr)`

Evaluates `Expr` and matches the result against `GuardedPattern`, if testing is enabled. If the match fails, an informative exception will be generated; see the `assert` macro for further details. `GuardedPattern` can be anything that you can write on the left hand side of the `->` symbol in a case-clause, except that it cannot contain comma-separated guard tests.

The main reason for using `assertMatch` also for simple matches, instead of matching with `=`, is that it produces more detailed error messages.

Examples:

```
?assertMatch({found, {fred, _}}, lookup(bloggs, Table))
```

```
?assertMatch([X|_] when X > 0, binary_to_list(B))
```

`assertNotMatch(GuardedPattern, Expr)`

The inverse case of assertMatch, for convenience.

`assertEqual(Expect, Expr)`

Evaluates the expressions `Expect` and `Expr` and compares the results for equality, if testing is enabled. If the values are not equal, an informative exception will be generated; see the `assert` macro for further details.

`assertEqual` is more suitable than `assertMatch` when the left-hand side is a computed value rather than a simple pattern, and gives more details than `?assert(Expect =:= Expr)`.

Examples:

```
?assertEqual("b" ++ "a", lists:reverse("ab"))
```

```
?assertEqual(foo(X), bar(Y))
```

`assertNotEqual(Unexpected, Expr)`

The inverse case of assertEqual, for convenience.

`assertException(ClassPattern, TermPattern, Expr)`
`assertError(TermPattern, Expr)`
`assertExit(TermPattern, Expr)`
`assertThrow(TermPattern, Expr)`

Evaluates `Expr`, catching any exception and testing that it matches the expected `ClassPattern:TermPattern`. If the match fails, or if no exception is thrown by `Expr`, an informative exception will be generated; see the `assert` macro for further details. The `assertError`, `assertExit`, and `assertThrow` macros, are equivalent to using `assertException` with a `ClassPattern` of `error`, `exit`, or `throw`, respectively.

Examples:

```
?assertError(badarith, X/0)
```

```
?assertExit(normal, exit(normal))
```

```
?assertException(throw, {not_found,_}, throw({not_found,42}))
```

Macros for running external commands

Keep in mind that external commands are highly dependent on the operating system. You can use the standard library function `os:type()` in test generator functions, to produce different sets of tests depending on the current operating system.

Note: these macros introduce a run-time dependency on the EUnit library code, if compiled with testing enabled.

`assertCmd(CommandString)`

Runs `CommandString` as an external command, if testing is enabled. Unless the returned status value is 0, an informative exception will be generated. If there is no exception, the result of the macro expression is the atom `ok`. If testing is disabled, the macro will not generate any code except the atom `ok`, and the command will not be executed.

Typical usage:

```
?assertCmd("mkdir foo")
```

`assertCmdStatus(N, CommandString)`

Like the `assertCmd(CommandString)` macro, but generates an exception unless the returned status value is `N`.

`assertCmdOutput(Text, CommandString)`

Runs `CommandString` as an external command, if testing is enabled. Unless the output produced by the command exactly matches the specified string `Text`, an informative exception will be generated. (Note that the output is normalized to use a single LF character as line break on all platforms.) If there is no exception, the result of the macro expression is the atom `ok`. If testing is disabled, the macro will not generate any code except the atom `ok`, and the command will not be executed.

`cmd(CommandString)`

Runs `CommandString` as an external command. Unless the returned status value is 0 (indicating success), an informative exception will be generated; otherwise, the result of the macro expression is the output produced by the command, as a flat string. The output is normalized to use a single LF character as line break on all platforms.

This macro is useful in the setup and cleanup sections of fixtures, e.g., for creating and deleting files or perform similar operating system specific tasks, to make sure that the test system is informed of any failures.

A Unix-specific example:

```
{setup,
 fun () -> ?cmd("mktemp") end,
 fun (FileName) -> ?cmd("rm " ++ FileName) end,
 ...}
```

Debugging macros

To help with debugging, EUnit defines several useful macros for printing messages directly to the console (rather than to the standard output). Furthermore, these macros all use the same basic format, which includes the file and line number where they occur, making it possible in some development environments (e.g., when running Erlang in an Emacs buffer) to simply click on the message and jump directly to the corresponding line in the code.

If the macro `NODEBUG` is defined before the EUnit header file is included, these macros have no effect; see *Compilation control macros* for details.

`debugHere`

Just prints a marker showing the current file and line number. Note that this is an argument-less macro. The result is always `ok`.

`debugMsg(Text)`

Outputs the message `Text` (which can be a plain string, an IO-list, or just an atom). The result is always `ok`.

`debugFmt(FmtString, Args)`

This formats the text like `io:format(FmtString, Args)` and outputs it like `debugMsg`. The result is always `ok`.

`debugVal(Expr)`

Prints both the source code for `Expr` and its current value. E.g., `?debugVal(f(X))` might be displayed as "`f(X) = 42`". (Large terms are truncated to the depth given by the macro `EUNIT_DEBUG_VAL_DEPTH`, which defaults to 15 but can be overridden by the user.) The result is always the value of `Expr`, so this macro can be wrapped around any expression to display its value when the code is compiled with debugging enabled.

`debugVal(Expr, Depth)`

Like `debugVal(Expr)`, but prints terms truncated to the given depth.

`debugTime(Text,Expr)`

Prints `Text` and the wall clock time for evaluation of `Expr`. The result is always the value of `Expr`, so this macro can be wrapped around any expression to show its run time when the code is compiled with debugging enabled. For example, `List1 = ?debugTime("sorting", lists:sort(List))` might show as "`sorting: 0.015 s`".

1.1.5 EUnit test representation

The way EUnit represents tests and test sets as data is flexible, powerful, and concise. This section describes the representation in detail.

- *Simple test objects*
- *Test sets and deep lists*
- *Titles*
- *Primitives*
- *Control*
- *Fixtures*
- *Lazy generators*

Simple test objects

A **simple test object** is one of the following:

- A nullary functional value (i.e., a fun that takes zero arguments). Examples:

```
fun () -> ... end
```

```
fun some_function/0
```

```
fun some_module:some_function/0
```

- A tuple `{test, ModuleName, FunctionName}`, where `ModuleName` and `FunctionName` are atoms, referring to the function `ModuleName:FunctionName/0`
- (Obsolete) A pair of atoms `{ModuleName, FunctionName}`, equivalent to `{test, ModuleName, FunctionName}` if nothing else matches first. This might be removed in a future version.
- A pair `{LineNumber, SimpleTest}`, where `LineNumber` is a nonnegative integer and `SimpleTest` is another simple test object. `LineNumber` should indicate the source line of the test. Pairs like this are usually only created via `?_test(...)` macros; see *Basic macros*.

In brief, a simple test object consists of a single function that takes no arguments (possibly annotated with some additional metadata, i.e., a line number). Evaluation of the function either **succeeds**, by returning some value (which is ignored), or **fails**, by throwing an exception.

Test sets and deep lists

A test set can be easily created by placing a sequence of test objects in a list. If `T_1`, ..., `T_N` are individual test objects, then `[T_1, ..., T_N]` is a test set consisting of those objects (in that order).

Test sets can be joined in the same way: if `S_1`, ..., `S_K` are test sets, then `[S_1, ..., S_K]` is also a test set, where the tests of `S_i` are ordered before those of `S_(i+1)`, for each subset `S_i`.

Thus, the main representation of test sets is **deep lists**, and a simple test object can be viewed as a test set containing only a single test; there is no difference between `T` and `[T]`.

A module can also be used to represent a test set; see `ModuleName` under *Primitives* below.

Titles

Any test or test set `T` can be annotated with a title, by wrapping it in a pair `{Title, T}`, where `Title` is a string. For convenience, any test which is normally represented using a tuple can simply be given a title string as the first element, i.e., writing `{"The Title", ...}` instead of adding an extra tuple wrapper as in `{"The Title", {...}}`.

Primitives

The following are primitives, which do not contain other test sets as arguments:

`ModuleName::atom()`

A single atom represents a module name, and is equivalent to `{module, ModuleName}`. This is often used as in the call `eunit:test(some_module)`.

`{module, ModuleName::atom()}`

This composes a test set from the exported test functions of the named module, i.e., those functions with arity zero whose names end with `_test` or `_test_`. Basically, the `..._test()` functions become simple tests, while the `..._test_()` functions become generators.

In addition, EUnit will also look for another module whose name is `ModuleName` plus the suffix `_tests`, and if it exists, all the tests from that module will also be added. (If `ModuleName` already contains the suffix `_tests`, this is not done.) E.g., the specification `{module, mymodule}` will run all tests in the modules `mymodule` and `mymodule_tests`. Typically, the `_tests` module should only contain test cases that use the public interface of the main module (and no other code).

`{application, AppName::atom(), Info::list()}`

This is a normal Erlang/OTP application descriptor, as found in an `.app` file. The resulting test set consists of the modules listed in the `modules` entry in `Info`.

`{application, AppName::atom()}`

This creates a test set from all the modules belonging to the specified application, by consulting the application's `.app` file (see `{file, FileName}`), or if no such file exists, by testing all object files in the application's `ebin`-directory (see `{dir, Path}`); if that does not exist, the `code:lib_dir(AppName)` directory is used.

`Path::string()`

A single string represents the path of a file or directory, and is equivalent to `{file, Path}`, or `{dir, Path}`, respectively, depending on what `Path` refers to in the file system.

`{file, FileName::string()}`

If `FileName` has a suffix that indicates an object file (`.beam`), EUnit will try to reload the module from the specified file and test it. Otherwise, the file is assumed to be a text file containing test specifications, which will be read using the standard library function `file:path_consult/2`.

Unless the file name is absolute, the file is first searched for relative to the current directory, and then using the normal search path (`code:get_path()`). This means that the names of typical "app" files can be used directly, without a path, e.g., `"mnesia.app"`.

`{dir, Path::string()}`

This tests all object files in the specified directory, as if they had been individually specified using `{file, FileName}`.

`{generator, GenFun::(() -> Tests)}`

The generator function `GenFun` is called to produce a test set.

`{generator, ModuleName::atom(), FunctionName::atom()}`

The function `ModuleName:FunctionName()` is called to produce a test set.

`{with, X::any(), [AbstractTestFun::((any()) -> any())]}`

Distributes the value `X` over the unary functions in the list, turning them into nullary test functions. An `AbstractTestFun` is like an ordinary test fun, but takes one argument instead of zero - it's basically missing some information before it can be a proper test. In practice, `{with, X, [F_1, ..., F_N]}` is equivalent to `[fun () -> F_1(X) end, ..., fun () -> F_N(X) end]`. This is particularly useful if your abstract test functions are already implemented as proper functions: `{with, FD, [fun filetest_a/1, fun filetest_b/1, fun filetest_c/1]}` is equivalent to `[fun () -> filetest_a(FD) end, fun () -> filetest_b(FD) end, fun () -> filetest_c(FD) end]`, but much more compact. See also *Fixtures*, below.

Control

The following representations control how and where tests are executed:

`{spawn, Tests}`

Runs the specified tests in a separate subprocess, while the current test process waits for it to finish. This is useful for tests that need a fresh, isolated process state. (Note that EUnit always starts at least one such a subprocess automatically; tests are never executed by the caller's own process.)

`{spawn, Node::atom(), Tests}`

Like `{spawn, Tests}`, but runs the specified tests on the given Erlang node.

`{timeout, Time::number(), Tests}`

Runs the specified tests under the given timeout. Time is in seconds; e.g., 60 means one minute and 0.1 means 1/10th of a second. If the timeout is exceeded, the unfinished tests will be forced to terminate. Note that if a timeout is set around a fixture, it includes the time for setup and cleanup, and if the timeout is triggered, the entire fixture is abruptly terminated (without running the cleanup). The default timeout for an individual test is 5 seconds.

`{inorder, Tests}`

Runs the specified tests in strict order. Also see `{inparallel, Tests}`. By default, tests are neither marked as `inorder` or `inparallel`, but may be executed as the test framework chooses.

`{inparallel, Tests}`

Runs the specified tests in parallel (if possible). Also see `{inorder, Tests}`.

`{inparallel, N::integer(), Tests}`

Like `{inparallel, Tests}`, but running no more than `N` subtests simultaneously.

Fixtures

A "fixture" is some state that is necessary for a particular set of tests to run. EUnit's support for fixtures makes it easy to set up such state locally for a test set, and automatically tear it down again when the test set is finished, regardless of the outcome (success, failures, timeouts, etc.).

To make the descriptions simpler, we first list some definitions:

`Setup`	`() -> (R::any())`
`SetupX`	`(X::any()) -> (R::any())`
`Cleanup`	`(R::any()) -> any()`
`CleanupX`	`(X::any(), R::any()) -> any()`
`Instantiator`	`((R::any()) -> Tests) | {with, [AbstractTestFun::((any()) -> any())]}`
`Where`	`local | spawn | {spawn, Node::atom()}`

Table 1.1:

(these are explained in more detail further below.)

The following representations specify fixture handling for test sets:

```
{setup, Setup, Tests | Instantiator}
{setup, Setup, Cleanup, Tests | Instantiator}
{setup, Where, Setup, Tests | Instantiator}
{setup, Where, Setup, Cleanup, Tests | Instantiator}
```

`setup` sets up a single fixture for running all of the specified tests, with optional teardown afterwards. The arguments are described in detail below.

```
{node, Node::atom(), Tests | Instantiator}
{node, Node::atom(), Args::string(), Tests | Instantiator}
```

`node` is like `setup`, but with a built-in behaviour: it starts a slave node for the duration of the tests. The atom `Node` should have the format `nodename@full.machine.name`, and `Args` are the optional arguments to the new node; see `slave:start_link/3` for details.

```
{foreach, Where, Setup, Cleanup, [Tests | Instantiator]}
{foreach, Setup, Cleanup, [Tests | Instantiator]}
{foreach, Where, Setup, [Tests | Instantiator]}
{foreach, Setup, [Tests | Instantiator]}
```

`foreach` is used to set up a fixture and optionally tear it down afterwards, repeated for each single one of the specified test sets.

```
{foreachx, Where, SetupX, CleanupX, Pairs::[{X::any(), ((X::any(), R::any())
-> Tests)}]}
{foreachx, SetupX, CleanupX, Pairs}
{foreachx, Where, SetupX, Pairs}
{foreachx, SetupX, Pairs}
```

`foreachx` is like `foreach`, but uses a list of pairs, each containing an extra argument `X` and an extended instantiator function.

A `Setup` function is executed just before any of the specified tests are run, and a `Cleanup` function is executed when no more of the specified tests will be run, regardless of the reason. A `Setup` function takes no argument, and returns some value which will be passed as it is to the `Cleanup` function. A `Cleanup` function should do whatever necessary and return some arbitrary value, such as the atom `ok`. (`SetupX` and `CleanupX` functions are similar, but receive one additional argument: some value `X`, which depends on the context.) When no `Cleanup` function is specified, a dummy function is used which has no effect.

An `Instantiator` function receives the same value as the `Cleanup` function, i.e., the value returned by the `Setup` function. It should then behave much like a generator (see *Primitives*), and return a test set whose tests have been **instantiated** with the given value. A special case is the syntax `{with, [AbstractTestFun]}` which represents an instantiator function that distributes the value over a list of unary functions; see *Primitives*: `{with, X, [...]}` for more details.

A `Where` term controls how the specified tests are executed. The default is `spawn`, which means that the current process handles the setup and teardown, while the tests are executed in a subprocess. `{spawn, Node}` is like `spawn`, but runs the subprocess on the specified node. `local` means that the current process will handle both setup/teardown and running the tests - the drawback is that if a test times out so that the process is killed, the **cleanup will not be performed**; hence, avoid this for persistent fixtures such as file operations. In general, `local` should only be used when:

- the setup/teardown needs to be executed by the process that will run the tests;
- no further teardown needs to be done if the process is killed (i.e., no state outside the process was affected by the setup)

Lazy generators

Sometimes, it can be convenient not to produce the whole set of test descriptions before the testing begins; for example, if you want to generate a huge amount of tests that would take up too much space to keep in memory all at once.

It is fairly easy to write a generator which, each time it is called, either produces an empty list if it is done, or otherwise produces a list containing a single test case plus a new generator which will produce the rest of the tests. This demonstrates the basic pattern:

```
lazy_test_() ->
    lazy_gen(10000).

lazy_gen(N) ->
    {generator,
     fun () ->
         if N > 0 ->
                [?_test(...)
                 | lazy_gen(N-1)];
            true ->
                []
         end
     end}.
```

When EUnit traverses the test representation in order to run the tests, the new generator will not be called to produce the next test until the previous test has been executed.

Note that it is easiest to write this kind of recursive generator using a help function, like the `lazy_gen/1` function above. It can also be written using a recursive fun, if you prefer to not clutter your function namespace and are comfortable with writing that kind of code.

2 Reference Manual

The **EUnit** application contains modules with support for unit testing.

eunit

Erlang module

This module is the main EUnit user interface.

Exports

`start() -> term()`

Starts the EUnit server. Normally, you don't need to call this function; it is started automatically.

`stop() -> term()`

Stops the EUnit server. Normally, you don't need to call this function.

`test(Tests) -> term()`

Equivalent to *test(Tests, [])*.

`test(Tests::term(), Options::[term()]) -> ok | {error, term()}`

Runs a set of tests. The format of `Tests` is described in the section *EUnit test representation* of the overview.

Example:

```
eunit:test(fred)
```

runs all tests in the module `fred` and also any tests in the module `fred_tests`, if that module exists.

Options:

`verbose`

Displays more details about the running tests.

Options in the environment variable EUNIT are also included last in the option list, i.e., have lower precedence than those in `Options`.

See also: *test/1*.

eunit_surefire

Erlang module

Surefire reports for EUnit (Format used by Maven and Atlassian Bamboo for example to integrate test results). Based on initial code from Paul Guyot.

Example: Generate XML result file in the current directory:

```
    eunit:test([fib, eunit_examples],
               [{report,{eunit_surefire,[{dir,"."}]}}]).
```

Exports

```
handle_begin(Kind, Data, St) -> term()
```

```
handle_cancel(X1, Data, St) -> term()
```

```
handle_end(X1, Data, St) -> term()
```

```
init(Options) -> term()
```

```
start() -> term()
```

```
start(Options) -> term()
```

```
terminate(X1, St) -> term()
```

See also

eunit