elm packages elm-explorations / test / 1.2.2

elm test build passing

Write unit and fuzz tests for Elm code.

Quick Start

Here are three example tests:

```
suite : Test
suite =
    describe "The String module"
        [ describe "String.reverse" -- Nest as many descripti
            [ test "has no effect on a palindrome" <|
                \_ ->
                    let
                        palindrome =
                            "hannah"
                    in
                        Expect.equal palindrome (String.rever

            -- Expect.equal is designed to be used in pipelin
            , test "reverses a known string" <|
                \_ ->
                    "ABCDEFG"
                        |> String.reverse
                        |> Expect.equal "GFEDCBA"

            -- fuzz runs the test 100 times with randomly-gen
            , fuzz string "restores the original string if yo
                \randomlyGeneratedString ->
                    randomlyGeneratedString
                        |> String.reverse
                        |> String.reverse
                        |> Expect.equal randomlyGeneratedStri
            ]
        ]
```

This code uses a few common functions:

- `describe` to add a description string to a list of tests
- `test` to write a unit test
- `Expect` to determine if a test should pass or fail
- `fuzz` to run a function that produces a test several times with randomly-generated inputs

Check out a large real-world test suite for more.

Running tests locally

There are several ways you can run tests locally:

- from your terminal via `npm install -g elm-test`

README
About
Source

Modules

Search

Expect
Fuzz
Shrink
Test
Test.Html.Event
Test.Html.Query
Test.Html.Selector
Test.Runner
Test.Runner.Failure

- from your terminal via `npm install -g elm-test`
- from your browser

Here's how to set up and run your tests using the CLI test runner:

1. Run `npm install -g elm-test` if you haven't already.
2. `cd` into the project's root directory that has your `elm.json` .
3. Run `elm-test init` . It will create a `tests` directory inside this one, with some files in it.
4. Run `elm-test` .
5. Edit `tests/Example.elm` to introduce new tests.

Hint: If you have dependencies add them via `elm-test install authorName/dependencyName` .

Running tests on CI

Here are some examples of running tests on CI servers:

- `travis.yml`
- `appveyor.yml`

Not running tests

During development, you'll often want to focus on specific tests, silence failing tests, or jot down many ideas for tests that you can't implement all at once. We've got you covered with `skip` , `only` , and `todo` :

```
wipSuite : Test
wipSuite =
    describe "skip, only, and todo"
        [ only <|
            describe "Marking this test as `only` means no otl
                [ test "This test will be run" <|
                    \_ -> 1 + 1 |> Expect.equal 2
                , skip <| test "This test will be skipped, eve
                    \_ -> 2 + 3 |> Expect.equal 4
                ]
        , test "This test will be skipped because it has no `
            \_ -> "left" |> Expect.equal "right"
        , todo "Make sure all splines are reticulated"
        ]
```

If you run this example, or any suite that uses one of these three functions, it will result in an *incomplete* test run. No tests failed, but you also didn't run your entire suite, so we can't call it a success either. Incomplete test runs are reported to CI systems as indistinguishable from failed test runs, to safeguard against accidentally committing a gutted test suite!

Strategies for effective testing

1. Make impossible states unrepresentable so that you don't have to test that they can't occur.
2. When doing TDD, treat compiler errors as a red test. So feel free to write the test you wish you had even if it means calling functions that don't exist yet!

3. If your API is difficult for you to test, it will be difficult for someone else to use. You are your API's first client.
4. Prefer fuzz tests to unit tests, where possible. If you have a union type with a small number of values, list them all and map over the list with a unit test for each. Unit tests are also great for when you know the edge cases, and for regression tests.
5. If you're writing a library that wraps an existing standard or protocol, use examples from the specification or docs as unit tests.
6. For simple functions, it's okay to copy the implementation to the test; this is a useful regression check. But if the implementation isn't obviously right, try to write tests that don't duplicate the suspect logic. The great thing about fuzz tests is that you don't have to arrive at the exact same value as the code under test, just state something that will be true of that value.
7. Tests should be small and focused: call the code under test and set an expectation about the result. Setup code should be moved into reusable functions, or custom fuzzers. For example, a test to remove an element from a data structure should be given a nonempty data structure; it should not have to create one itself.
8. If you find yourself inspecting the fuzzed input and making different expectations based on it, split each code path into its own test with a fuzzer that makes only the right kind of values.
9. Consider using elm-verify-examples to extract examples in your docs into unit tests.
10. Not even your test modules can import unexposed functions, so test them only as the exposed interface uses them. Don't expose a function just to test it. Every exposed function should have tests. (If you practice TDD, this happens automatically!)
11. How do you know when to stop testing? This is an engineering tradeoff without a perfect answer. If you don't feel confident in the correctness of your code, write more tests. If you feel you are wasting time testing better spent writing your application, stop writing tests for now.

Application-specific techniques

There are a few extra ideas that apply to testing webapps and reusable view packages:

1. Avoid importing your `Main` module. Most of your code belongs in other modules, so import those instead.
2. Test your views using `Test.Html.Query`, `Test.Html.Selector`, and `Test.Html.Event`.
3. There is currently no Elm solution for integration or end-to-end testing. Use tools from outside the Elm ecosystem such as Capybara or Cypress.