Episode 12: ExUnit

Starting ExUnit

- ExUnit needs to run and manage multiple BEAM processes to run tests, so they must be started.
- In a Mix project, ExUnit is automatically started. See `test/test_helper.exs`.

```
# test/test_helper.exs
ExUnit.start
```

Writing ExUnit tests

For an assertion to pass, the value must

1. Successfullly match if it uses =
2. Return a truthy value

```
# in test/math_test.exs

defmodule MyApp.MathTest do
  use ExUnit.Case # immports features from the ExUnit Case module

  # Sharing variables between tests.
  setup do
    variable = "some value"
    {:ok, variable: variable} # This keyword list becomes a map in the test
↪   block!
  end

  test "some test", %{variable: variable} do
    # Use variable here...
  end

  # An example of a macro.
  test ".add sums two numbers" do
    assert Math.add(1, 2) == 3 # assertion macro
  end

  # Overriding the error message when a test fails.
  test "customer created" do
```

```
    refute customer == nil, "Customer was not created in the database"
  end

  # Custom assertions to clarify and reduce duplication in test code.
  def assert_customer_created(data) do
    assert data != nil
    assert data.customer != nil
  end
end
```

Assertions

There are multiple types of assertions you can use, available from ExUnit.Assertions.

```
assert
assert_raise    # Assert that a code block raises an exceptionn
assert_in_delta # Assert that two things differ in a specific way
assert_receive  # Assert that a process message was received
```

There are corresponding `refute` functions for most of these.

Sharing code between tests using a common `ExUnit.CaseTemplate`

TO DO: *"To cover this in full, we need to get deeper into meta-programming and macros than I want to do at this point, so we will save it for a future episode."*

Running tests asynchronously

- If tests do not rely on shared state—such as a database—you can safely run them asynchronously.
- Will run each test in a separate BEAM process. Can make them take less time to execute.

```
defmodule MyApp.MathTest do
  use ExUnit.Case, async: true

  # ...
end
```

Tagging

Tags can be useful for selectively running tests.

```
# Tag all tests in a module as slow.
@moduletag :slow

# Tag a single test.
@tag :slow
test "a really, really slow test" do
  # ...
end

# Temporarily skip a test.
@tag :skip
@tag skip: "reason"

# Automatically tagged with @tag :not_implemented.
test ".add sums two numbers"
test ".div divides two numbers"
test ".mul multiplies two numbers"
```

There is a lot of config available for `ExUnit.configure`. To exclude tests flagged with `:slow`, run:

```
ExUnit.start
ExUnit.configure exclude: [:slow]
```

Doctests

Benefits

- Your documentation can be relied on.
- If your code changes, doctests will remind you to update your docs.

When to use doctests

- Whenever possible, use doctests.

- Use a normal test if there is (1) a lot of setup required and/or (2) a doctest would require many lines
  - *However, may indicate that the function should (a) be private or (b) be simplified*

How to write doctests

```elixir
@doc """
Adds two numbers together and returns the sum.

## Examples

    iex> Math.add(2, 3)
    5

    iex> sum = Math.add(5, 5)
    ...> Math.add(sum, 5)
    15
"""
def add(a, b), do: a + b
```

How to turn doctests into tests

Use the `doctest` macro in your test file as shown:

```elixir
defmodule MyApp.MathTest do
  use ExUnit.Case

  doctest MyApp.Math
end
```

Running ExUnit tests

```
# In mix projects
mix test

# In other projects
```

```
elixir my_test.exs

# Exclude tests tagged with :slow
mix test --exclude slow

# Include tests tagged with :slow
mix test --include slow

# Run only tests tagged with :slow
mix test --only slow
```

Final thoughts

1. Write tests on every public function. > A public function can be used by anyone using your project! Make sure it is tested and works as intended.
2. If you don't want to test a function, make it private or set `@doc false`.
3. Prefer doctests to regular tests. > Helps you write and maintain documentation.