DevSpace **Elixir Study Group**
October 2014

Exploring ExUnit

@xavierdefrang
github.com/xavier

© elberling.dk

Disclaimer: This is not yet another talk about testing.

You already know you should do it.

ExUnit Internals

Today we focus on ExUnit internals

Why?

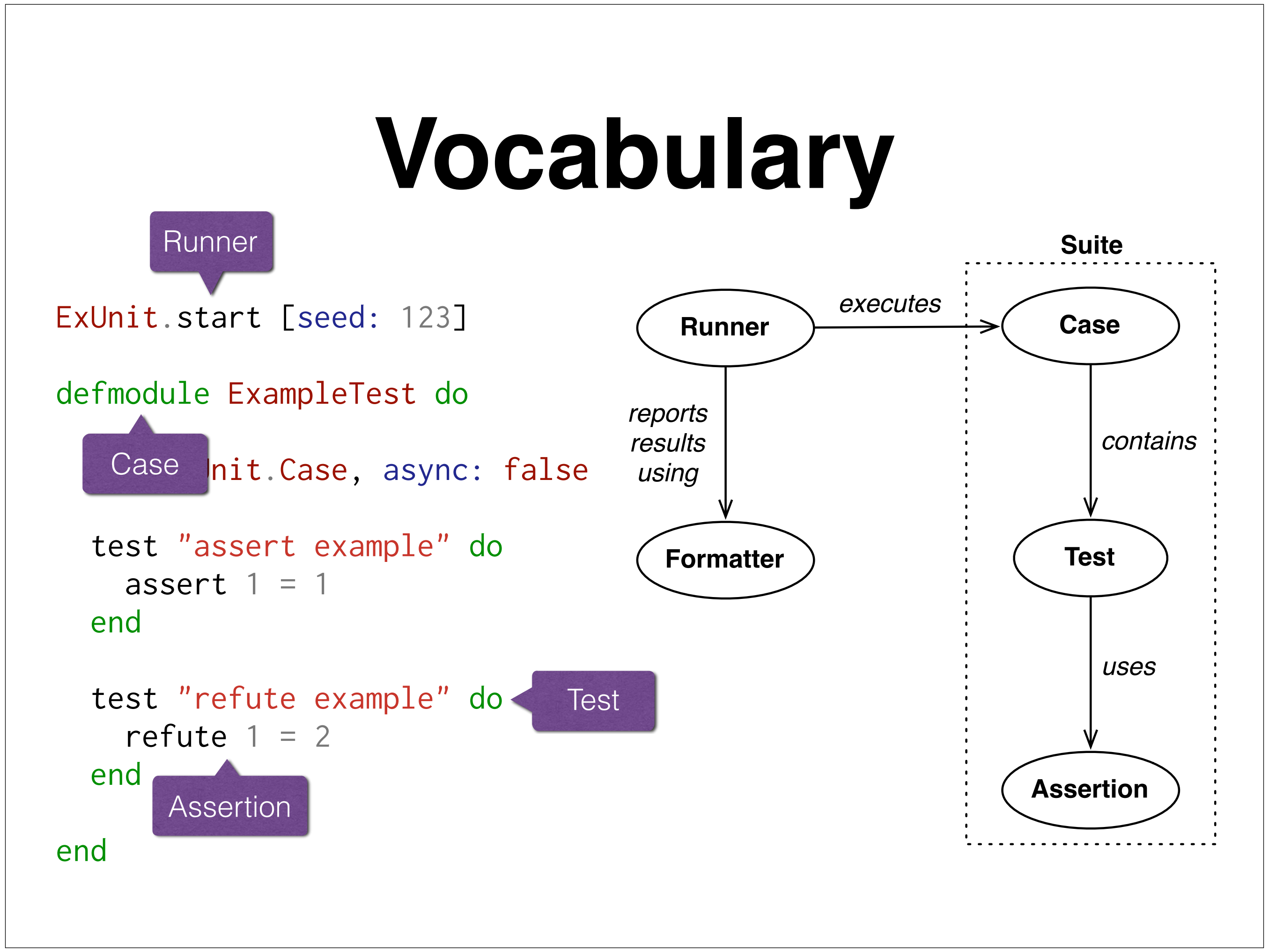
- Top of the class idiomatic Elixir code
- Application design patterns
- Advanced metaprogramming

Agenda

- Test runner architecture and code walkthrough
- Implementation of the testing DSL
- Awesome tricks

Example

```
ExUnit.start [seed: 123]

defmodule ExampleTest do

  use ExUnit.Case, async: false

  test ”assert example” do
    assert 1 = 1
  end

  test ”refute example” do
    refute 1 = 2
  end

end
```



Vocabulary

```
ExUnit.start [seed: 123]

defmodule ExampleTest do
       nit.Case, async: false

  test "assert example" do
    assert 1 = 1
  end

  test "refute example" do
    refute 1 = 2
  end

end
```

First things first: **vocabulary**

ExUnit.start will start the **runner** which executes a **test suite**

The **suite** is made of one or more test **cases**

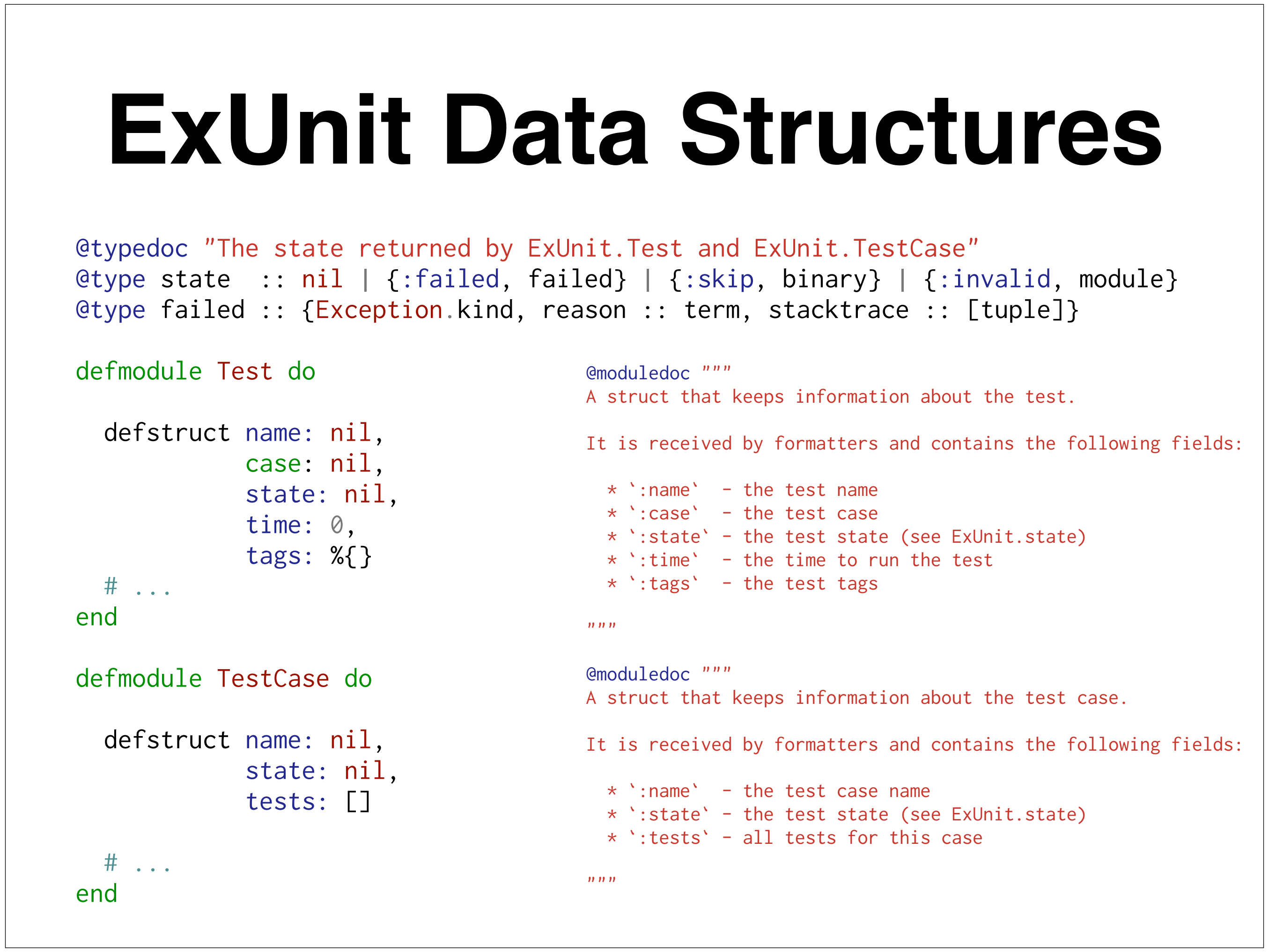
Test case are modules (ExampleTest here), they contain one or more tests

Test are named “blocks” of code containing some assertion

As the tests are executed, the runner provides feedback to the user using a **Formatter**.

ExUnit Features

- Test cases can have a `setup` function to populate a **context** passed down to each test
- Test cases can register “on exit” callbacks to perform some clean up tasks
- **Test cases** can be run in parallel or sequentially
- **Tests** are always run sequentially
- Both tests and test cases are run in **random order**



ExUnit Data Structures

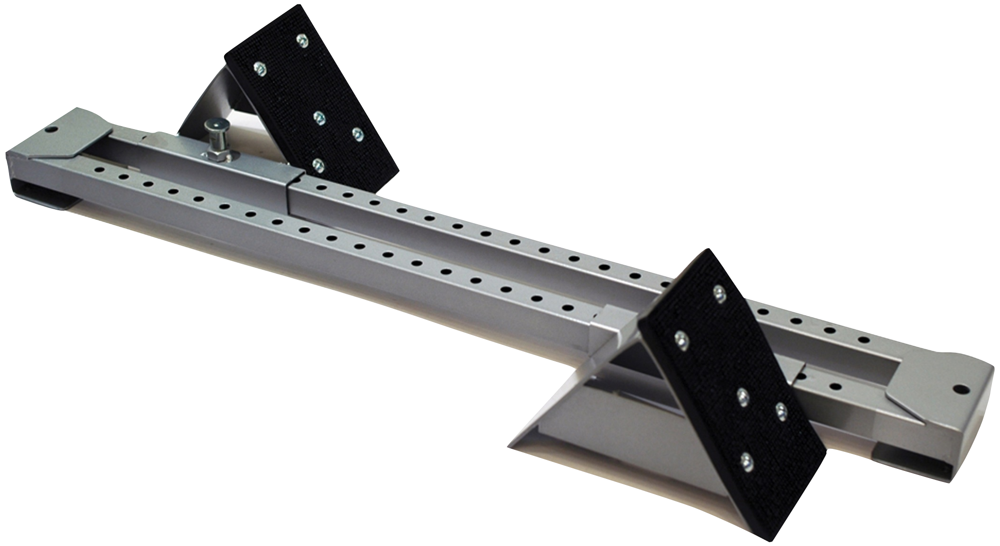
```
@typedoc "The state returned by ExUnit.Test and ExUnit.TestCase"
@type state  :: nil | {:failed, failed} | {:skip, binary} | {:invalid, module}
@type failed :: {Exception.kind, reason :: term, stacktrace :: [tuple]}

defmodule Test do

  defstruct name: nil,
            case: nil,
            state: nil,
            time: 0,
            tags: %{}
  # ...
end

defmodule TestCase do

  defstruct name: nil,
            state: nil,
            tests: []

  # ...
end
```

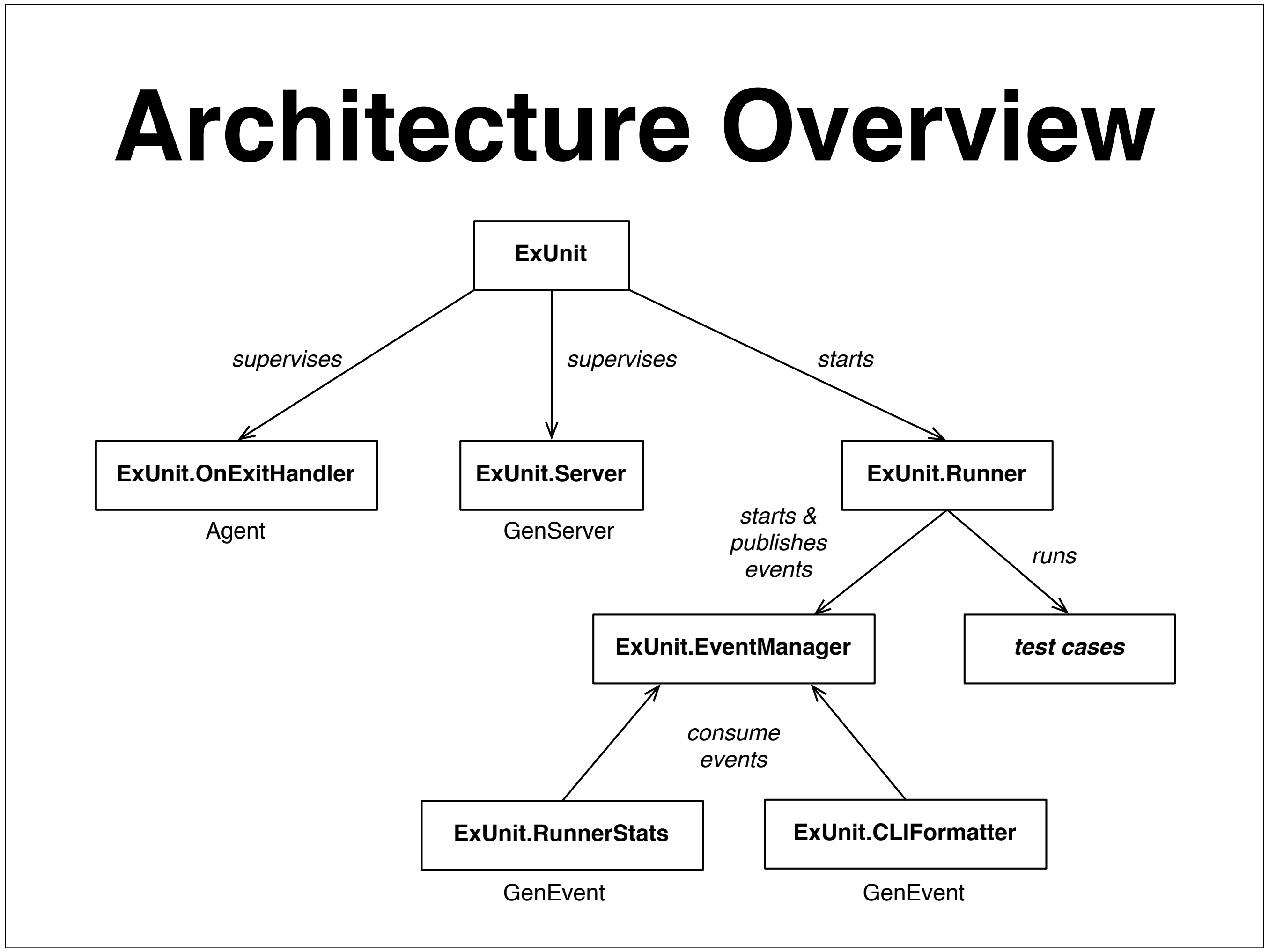
```
@moduledoc """
A struct that keeps information about the test.

It is received by formatters and contains the following fields:

  * `:name`  - the test name
  * `:case`  - the test case
  * `:state` - the test state (see ExUnit.state)
  * `:time`  - the time to run the test
  * `:tags`  - the test tags

"""

@moduledoc """
A struct that keeps information about the test case.

It is received by formatters and contains the following fields:

  * `:name`  - the test case name
  * `:state` - the test state (see ExUnit.state)
  * `:tests` - all tests for this case

"""
```

The data model behind ExUnit is straightforward.

Note the lack of “succeeded” state, we’ll only keep tracks of the things that went wrong

Starting & Running





Architecture Overview

ExUnit

supervises

supervises

starts

ExUnit.OnExitHandler

Agent

ExUnit.Server

GenServer

ExUnit.Runner

starts & publishes events

runs

ExUnit.EventManager

test cases

consume events

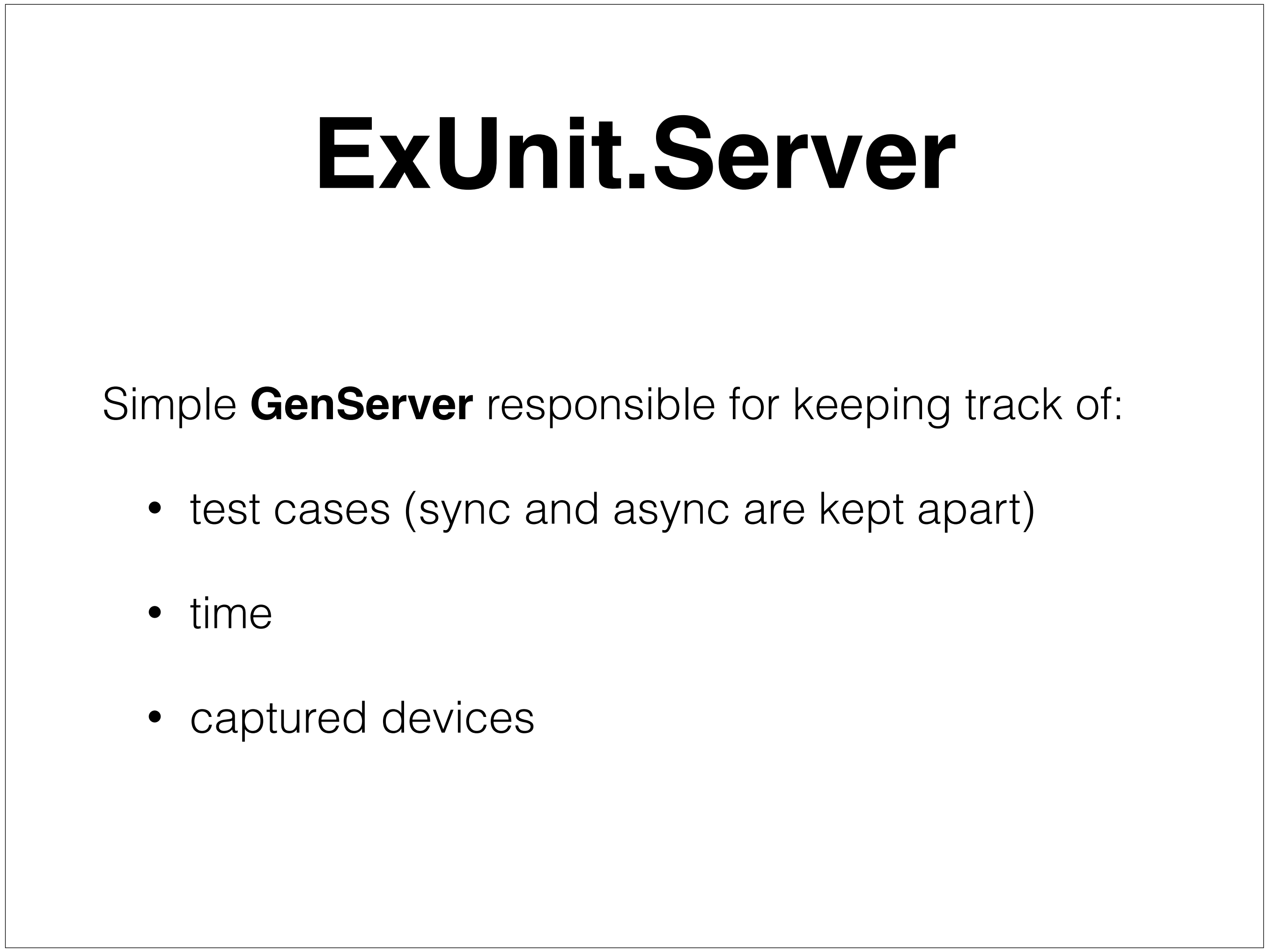
ExUnit.RunnerStats

GenEvent

ExUnit.CLIFormatter

GenEvent

Unsurprisingly, ExUnit is an OTP application.



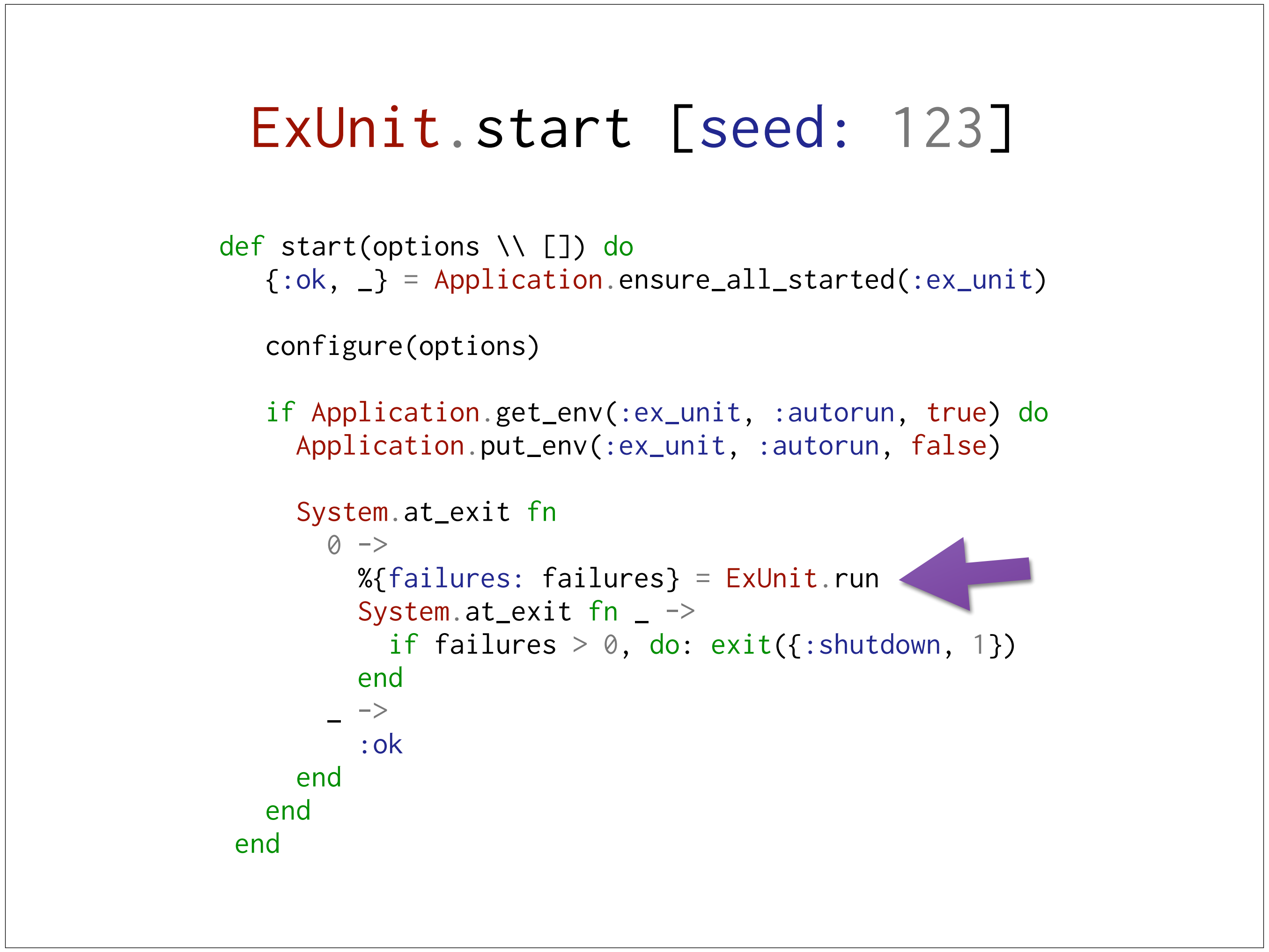
ExUnit.Server

Simple **GenServer** responsible for keeping track of:

- test cases (sync and async are kept apart)
- time
- captured devices

All test case modules get registered with this GenServer

Let’s Peel This Onion



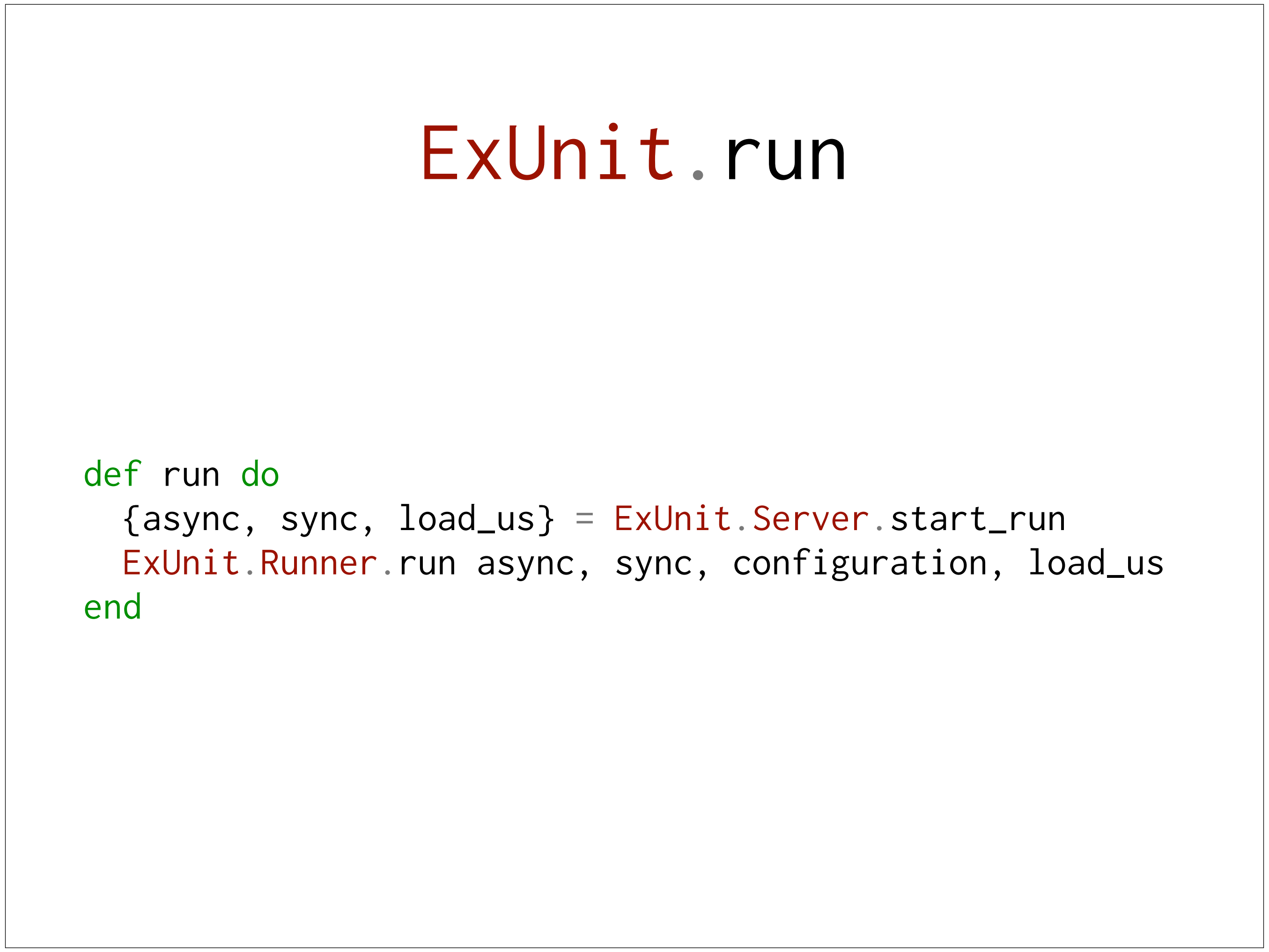
ExUnit.start [seed: 123]

```
def start(options \\ []) do
   {:ok, _} = Application.ensure_all_started(:ex_unit)

   configure(options)

   if Application.get_env(:ex_unit, :autorun, true) do
     Application.put_env(:ex_unit, :autorun, false)

     System.at_exit fn
       0 ->
         %{failures: failures} = ExUnit.run
         System.at_exit fn _ ->
           if failures > 0, do: exit({:shutdown, 1})
         end
       _ ->
         :ok
     end
   end
 end
```

configure injects all the given options in the app environment (= env variables for OTP apps)

ExUnit.**start** can be called before all the tests are defined, the tests are actually **run** once everything is ready, so right before exit

if there were any failures, make sure the program ends with an error exit status



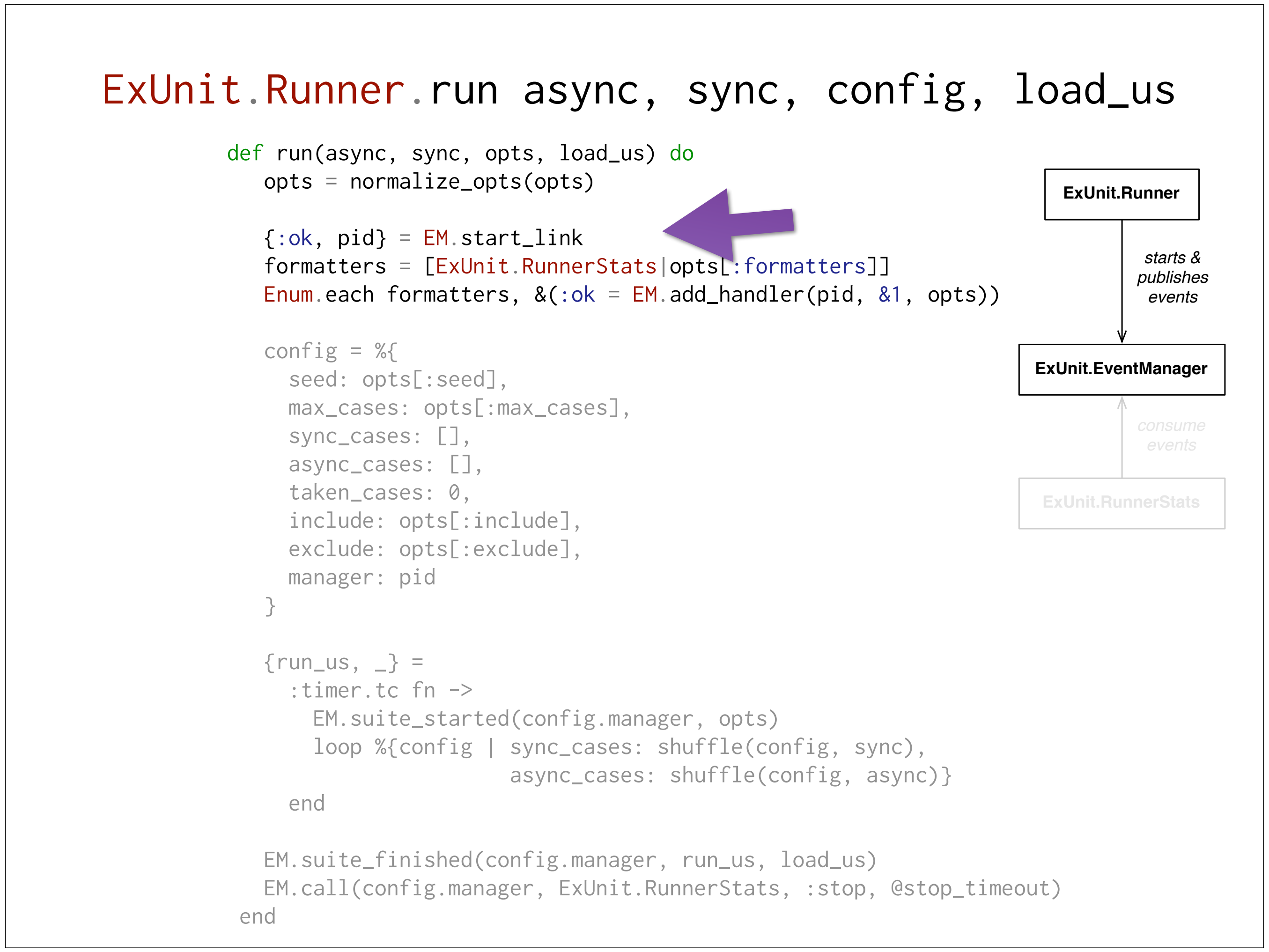
ExUnit.run

```elixir
def run do
  {async, sync, load_us} = ExUnit.Server.start_run
  ExUnit.Runner.run async, sync, configuration, load_us
end
```

start_run returns the **start time** and the **test cases** to execute

async and sync test cases are kept in separate collections

At this point, **configuration** is a copy of the app environment



ExUnit.Runner.run async, sync, config, load_us

```
def run(async, sync, opts, load_us) do
   opts = normalize_opts(opts)

   {:ok, pid} = EM.start_link
   formatters = [ExUnit.RunnerStats|opts[:formatters]]
   Enum.each formatters, &(:ok = EM.add_handler(pid, &1, opts))

   config = %{
     seed: opts[:seed],
     max_cases: opts[:max_cases],
     sync_cases: [],
     async_cases: [],
     taken_cases: 0,
     include: opts[:include],
     exclude: opts[:exclude],
     manager: pid
   }

   {run_us, _} =
     :timer.tc fn ->
       EM.suite_started(config.manager, opts)
       loop %{config | sync_cases: shuffle(config, sync),
                       async_cases: shuffle(config, async)}
     end

   EM.suite_finished(config.manager, run_us, load_us)
   EM.call(config.manager, ExUnit.RunnerStats, :stop, @stop_timeout)
 end
```

ExUnit.Runner

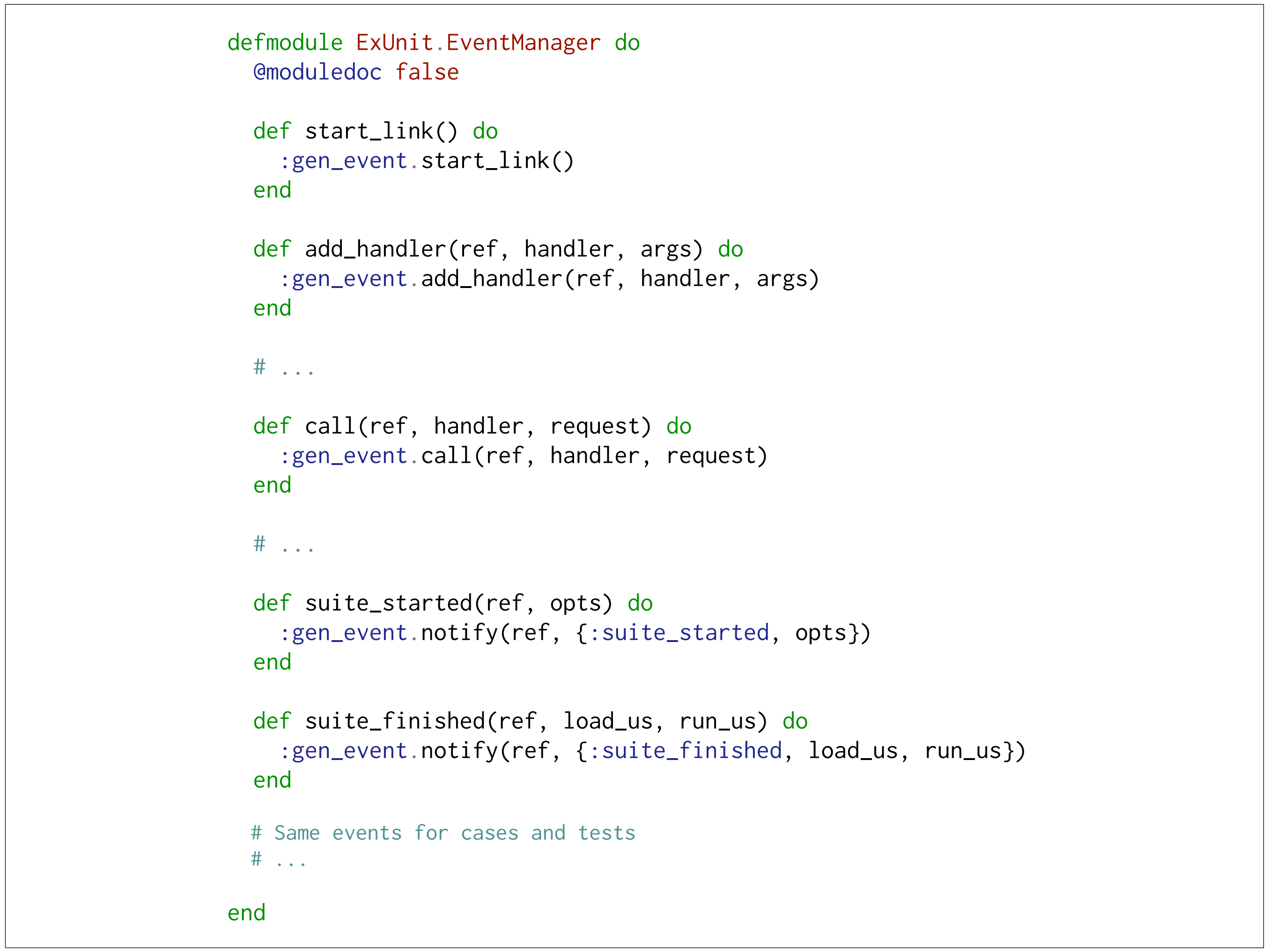
starts & publishes events

ExUnit.EventManager

consume events

ExUnit.RunnerStats

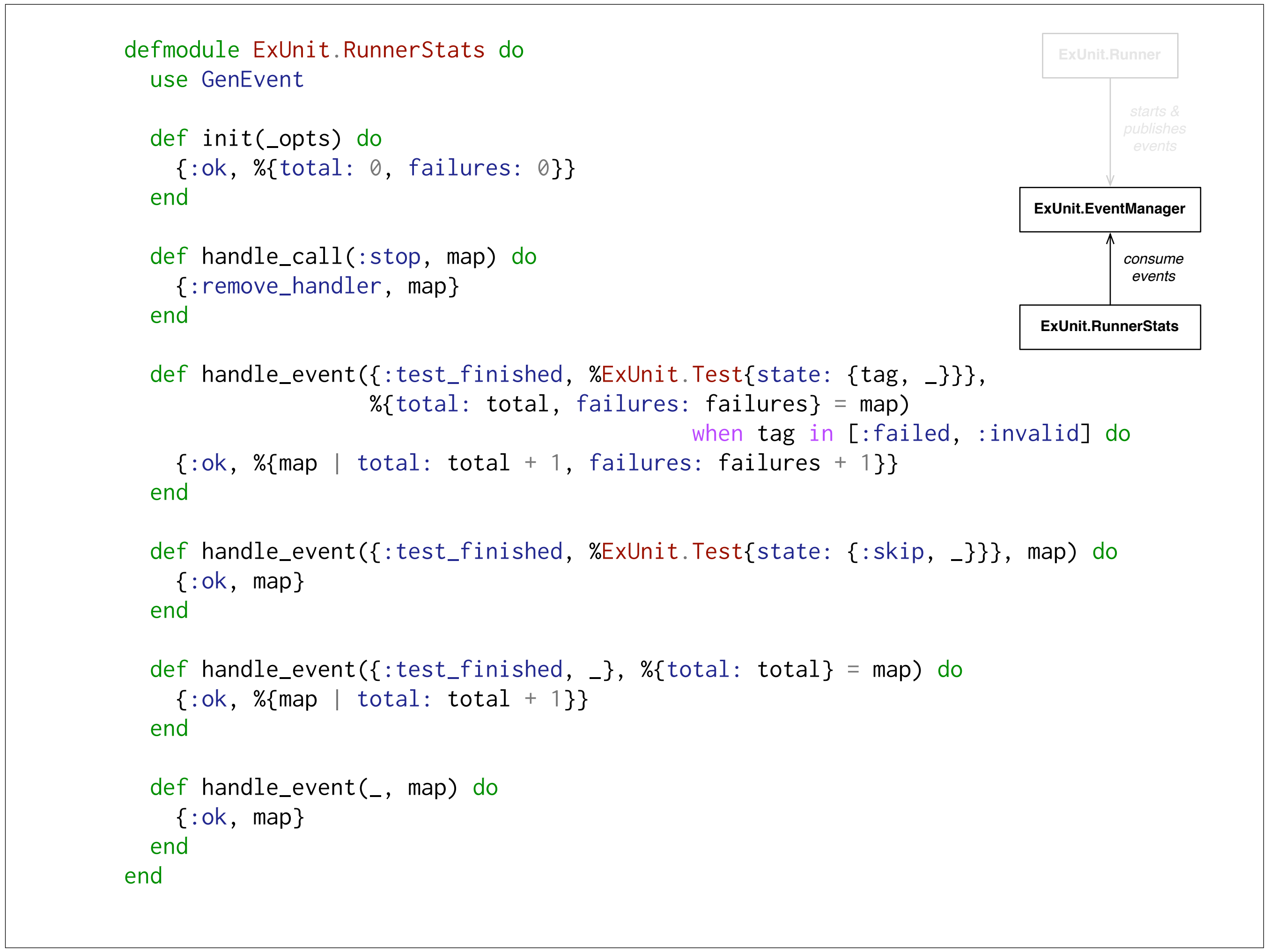
Starts the Event Manager and registers event consumers



```
defmodule ExUnit.EventManager do
  @moduledoc false

  def start_link() do
    :gen_event.start_link()
  end

  def add_handler(ref, handler, args) do
    :gen_event.add_handler(ref, handler, args)
  end

  # ...

  def call(ref, handler, request) do
    :gen_event.call(ref, handler, request)
  end

  # ...

  def suite_started(ref, opts) do
    :gen_event.notify(ref, {:suite_started, opts})
  end

  def suite_finished(ref, load_us, run_us) do
    :gen_event.notify(ref, {:suite_finished, load_us, run_us})
  end

  # Same events for cases and tests
  # ...

end
```

Thin abstraction layer on top of gen_event (generic pub-sub system part of OTP)

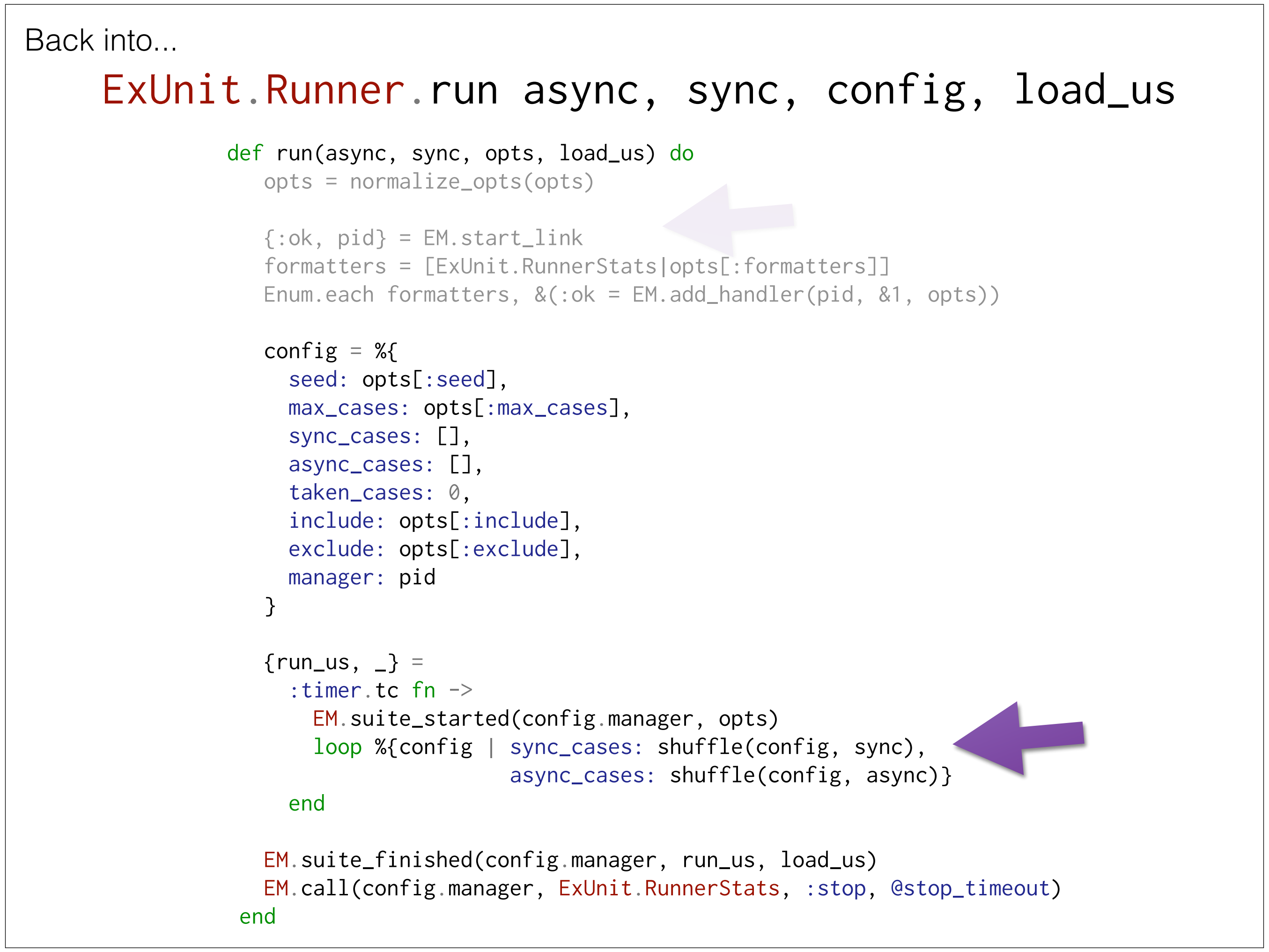
- event handlers can be registered
- the call function allows synchronous call of event handlers
- other notifications are asynchronous



```
defmodule ExUnit.RunnerStats do
  use GenEvent

  def init(_opts) do
    {:ok, %{total: 0, failures: 0}}
  end

  def handle_call(:stop, map) do
    {:remove_handler, map}
  end

  def handle_event({:test_finished, %ExUnit.Test{state: {tag, _}}},
                   %{total: total, failures: failures} = map)
                                            when tag in [:failed, :invalid] do
    {:ok, %{map | total: total + 1, failures: failures + 1}}
  end

  def handle_event({:test_finished, %ExUnit.Test{state: {:skip, _}}}, map) do
    {:ok, map}
  end

  def handle_event({:test_finished, _}, %{total: total} = map) do
    {:ok, %{map | total: total + 1}}
  end

  def handle_event(_, map) do
    {:ok, map}
  end
end
```

GenEvent consumer which keeps count of total number of tests and number of failures

- init with counters set to 0
- when it’s stopped (sync call) it returns its state (the stats)
- uses deep pattern matching on the event payload and its state



Back into...

ExUnit.Runner.run async, sync, config, load_us

```elixir
def run(async, sync, opts, load_us) do
  opts = normalize_opts(opts)

  {:ok, pid} = EM.start_link
  formatters = [ExUnit.RunnerStats|opts[:formatters]]
  Enum.each formatters, &(:ok = EM.add_handler(pid, &1, opts))

  config = %{
    seed: opts[:seed],
    max_cases: opts[:max_cases],
    sync_cases: [],
    async_cases: [],
    taken_cases: 0,
    include: opts[:include],
    exclude: opts[:exclude],
    manager: pid
  }

  {run_us, _} =
    :timer.tc fn ->
      EM.suite_started(config.manager, opts)
      loop %{config | sync_cases: shuffle(config, sync),
                      async_cases: shuffle(config, async)}
    end

  EM.suite_finished(config.manager, run_us, load_us)
  EM.call(config.manager, ExUnit.RunnerStats, :stop, @stop_timeout)
end
```

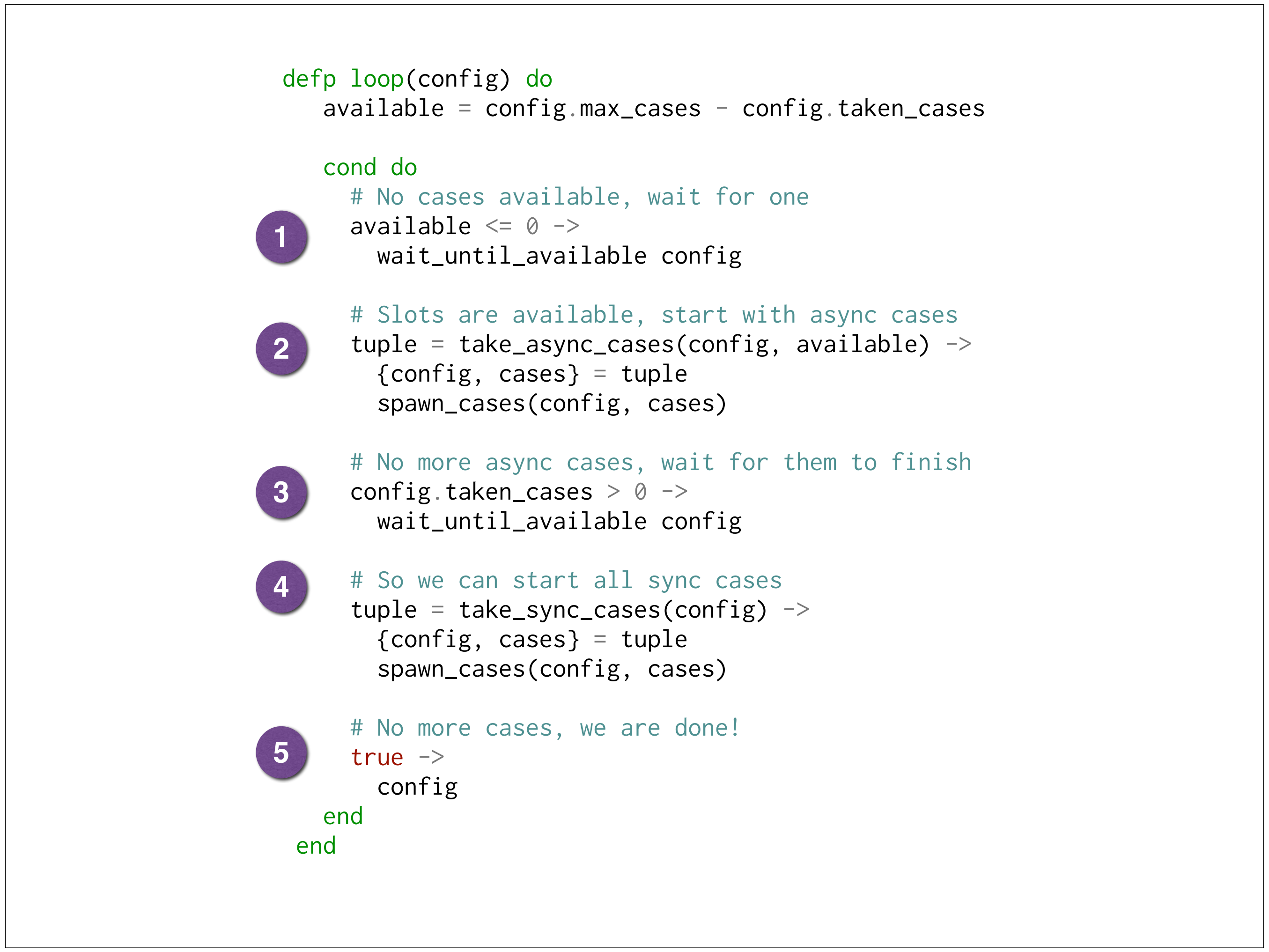
config holds the state of the test suite being executed

:timer.tc returns the time spent executing the given function

EM.call issues a sync call to the registered handler

Main Loop Algorithm

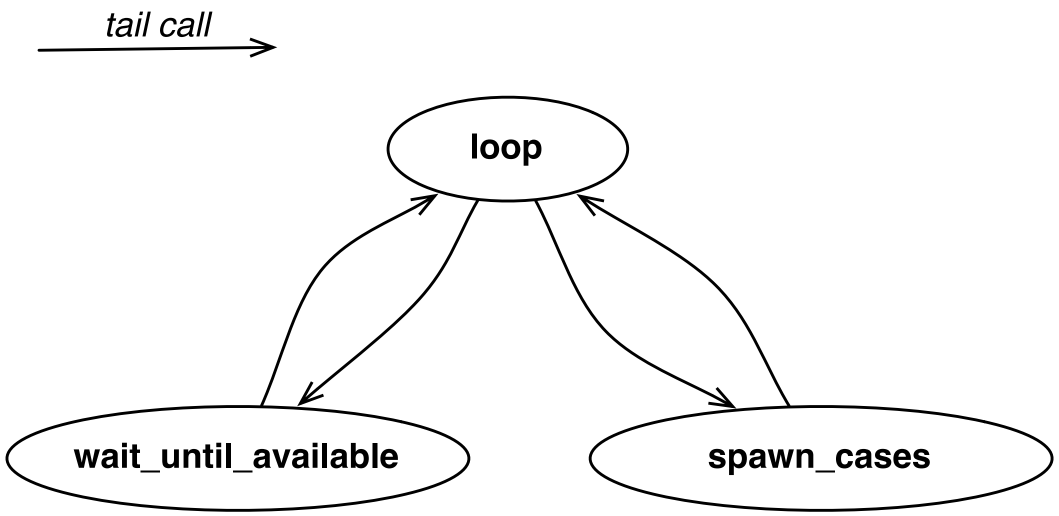
- As long as there are cases to run
  - As long as there are **async cases** to run
    - As soon as there are slots available
      - Execute as many async cases as possible
  - Wait for all running async cases to complete
  - Execute **sync cases** sequentially

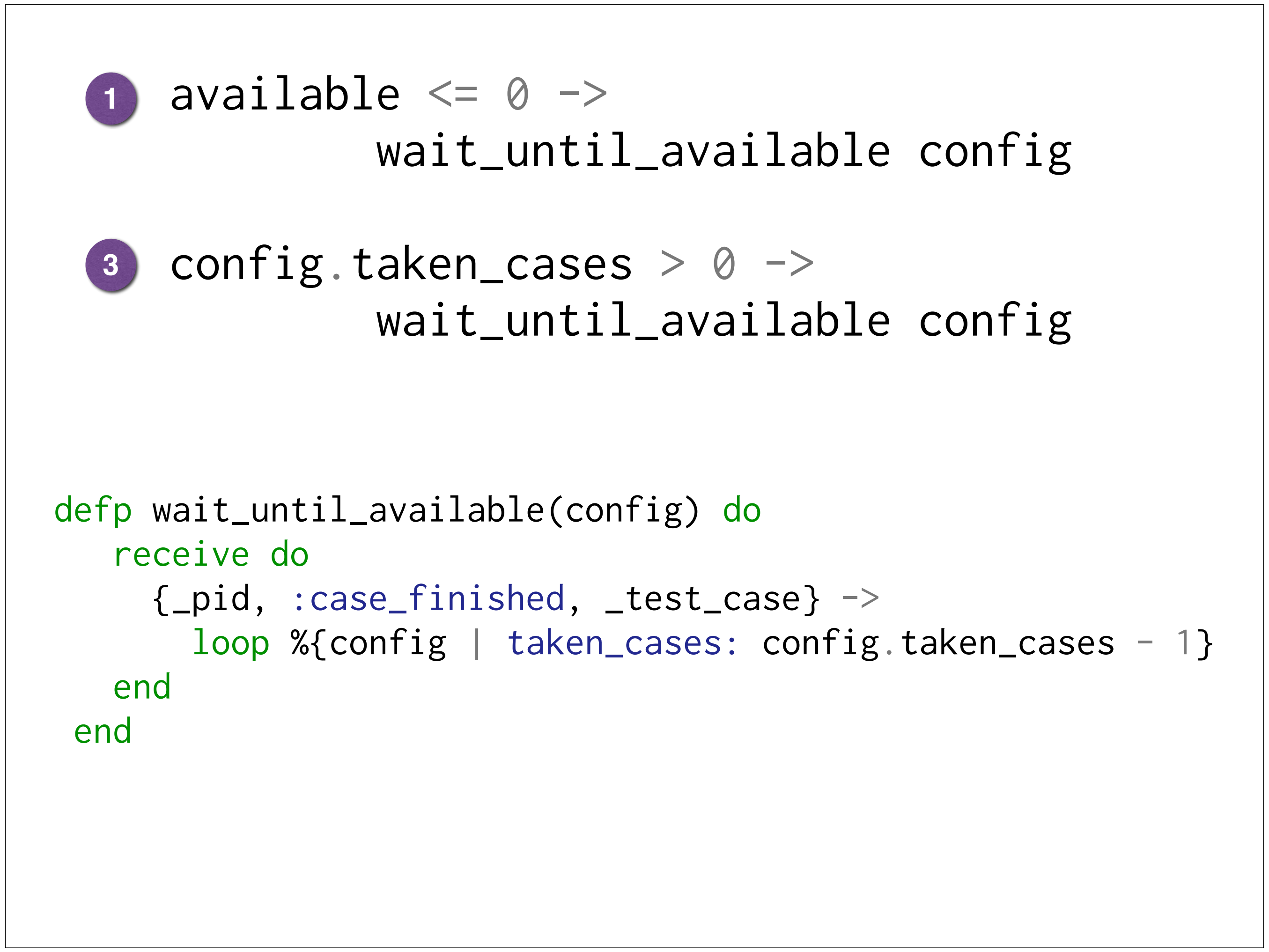


```
   defp loop(config) do
      available = config.max_cases - config.taken_cases

      cond do
        # No cases available, wait for one
1       available <= 0 ->
          wait_until_available config

        # Slots are available, start with async cases
2       tuple = take_async_cases(config, available) ->
          {config, cases} = tuple
          spawn_cases(config, cases)

        # No more async cases, wait for them to finish
3       config.taken_cases > 0 ->
          wait_until_available config

4       # So we can start all sync cases
        tuple = take_sync_cases(config) ->
          {config, cases} = tuple
          spawn_cases(config, cases)

        # No more cases, we are done!
5       true ->
          config
      end
    end
```

available = the number of async cases we can start on this iteration

Indirect Tail Recursion





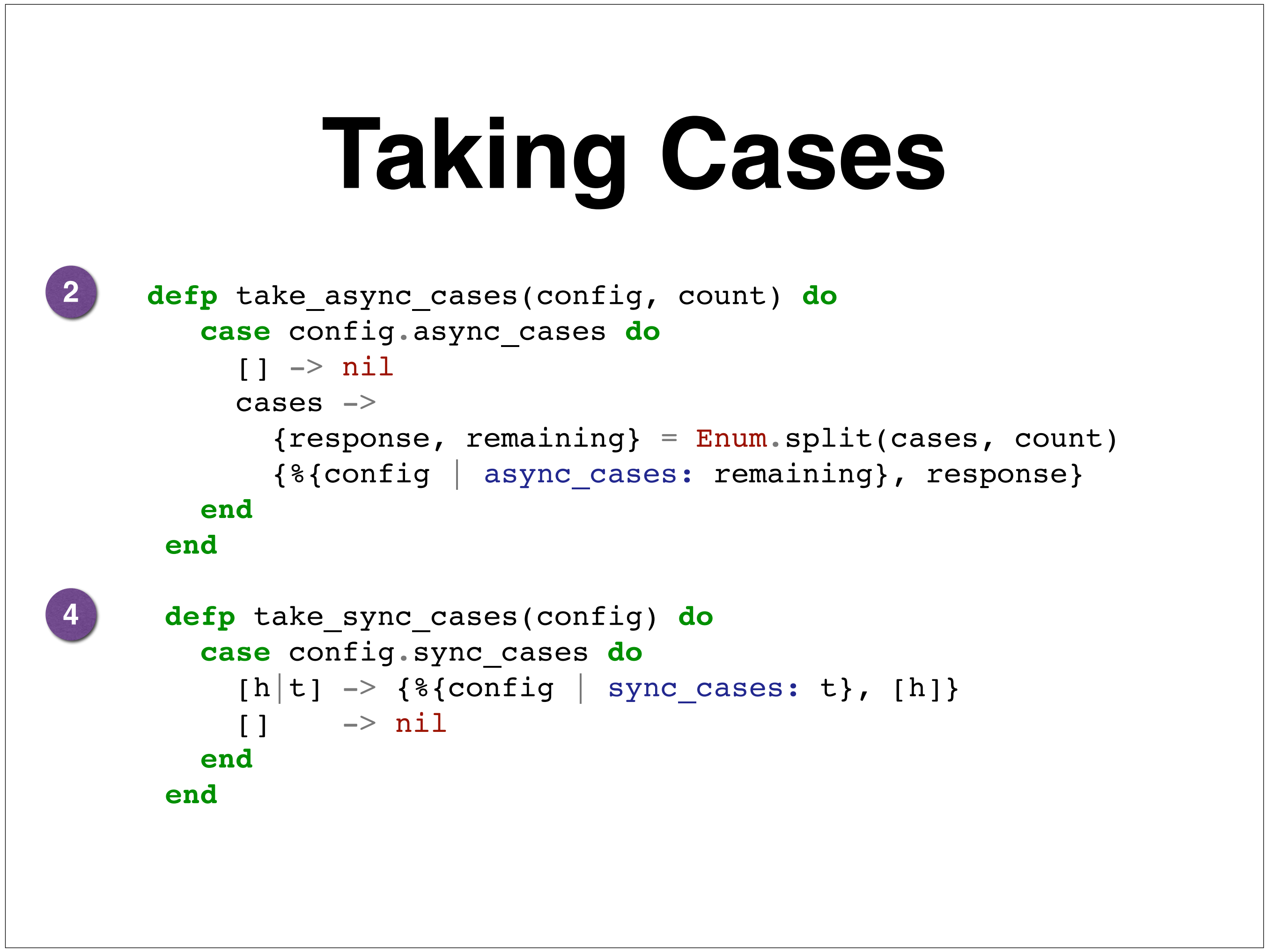
```
1 available <= 0 ->
          wait_until_available config

3 config.taken_cases > 0 ->
          wait_until_available config
```

```
defp wait_until_available(config) do
   receive do
     {_pid, :case_finished, _test_case} ->
       loop %{config | taken_cases: config.taken_cases - 1}
   end
 end
```

Blocks the runner until the execution of a test case has been completed.

As soon as a case finishes, a new iteration is started with the adjusted "currently running tests" count

Not to be mixed up with the events sent to the EventManager, this is an inter-process message used to between the test runner and its spawned child processes



Taking Cases

2

```
defp take_async_cases(config, count) do
   case config.async_cases do
     [] -> nil
     cases ->
       {response, remaining} = Enum.split(cases, count)
       {%{config | async_cases: remaining}, response}
   end
 end
```

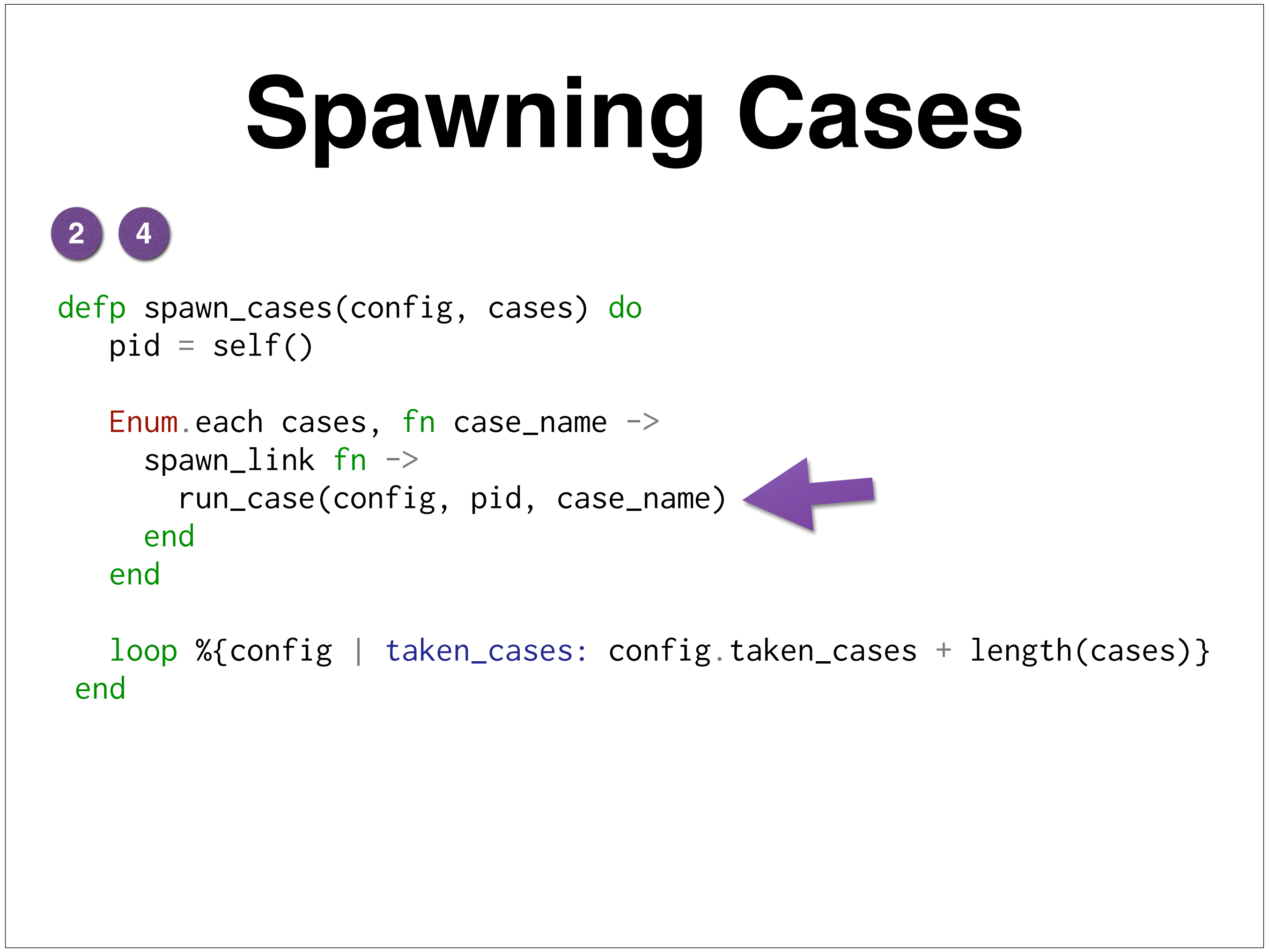
4

```
 defp take_sync_cases(config) do
   case config.sync_cases do
     [h|t] -> {%{config | sync_cases: t}, [h]}
     []    -> nil
   end
 end
```

take_async_cases returns up to count cases

take_sync_cases always return a single case (in a list for API consistency with async flow)

returning nil when the list is empty ensure that the cond pattern won’t match



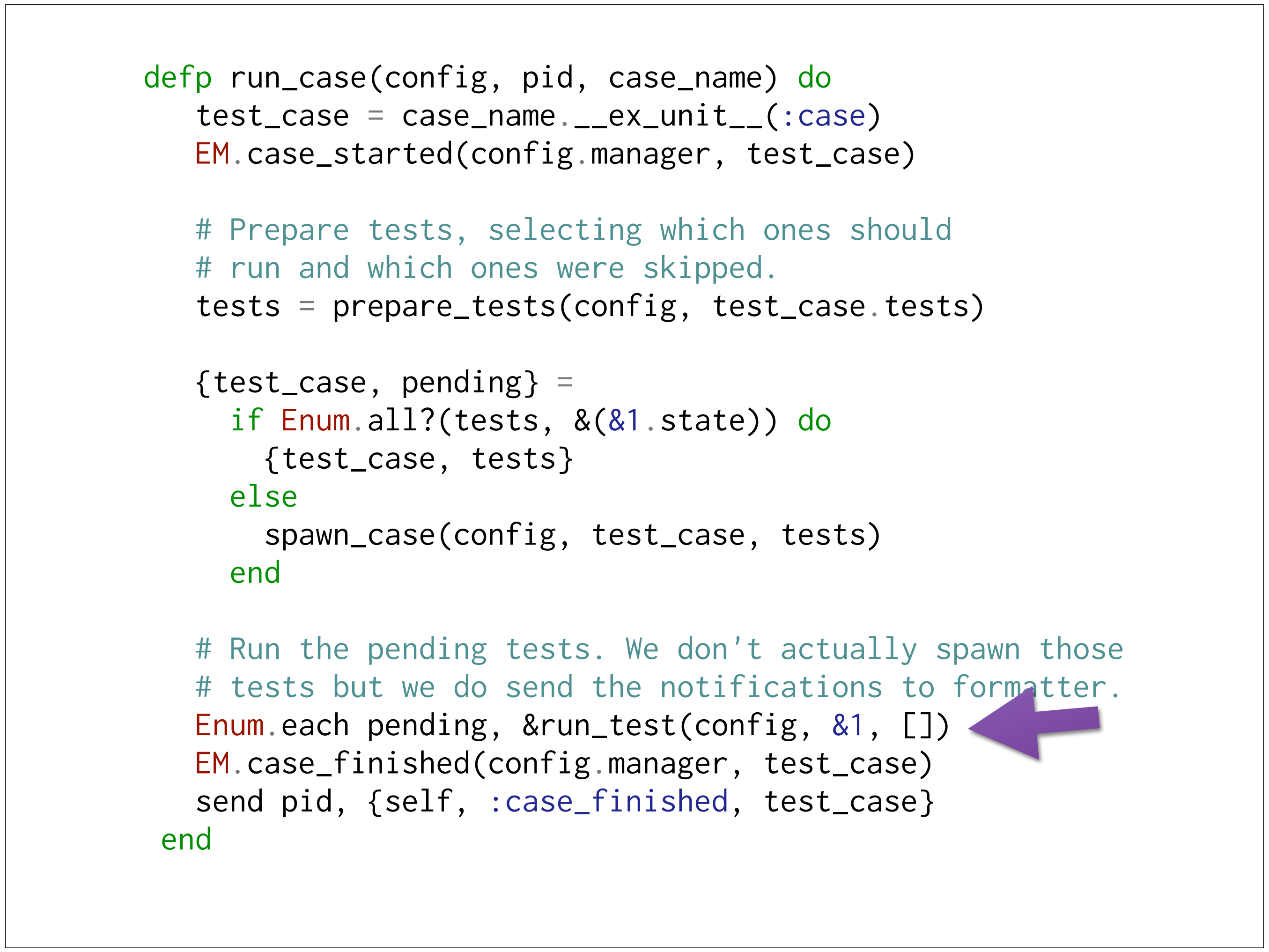
Spawning Cases

2 4

```
defp spawn_cases(config, cases) do
   pid = self()

   Enum.each cases, fn case_name ->
     spawn_link fn ->
       run_case(config, pid, case_name)
     end
   end

   loop %{config | taken_cases: config.taken_cases + length(cases)}
 end
```

Each case (sync and async alike) is run in a separate process.

When dealing with sync cases this function only receive a collection containing a single case but it’s nonetheless run in a process distinct from the test runner

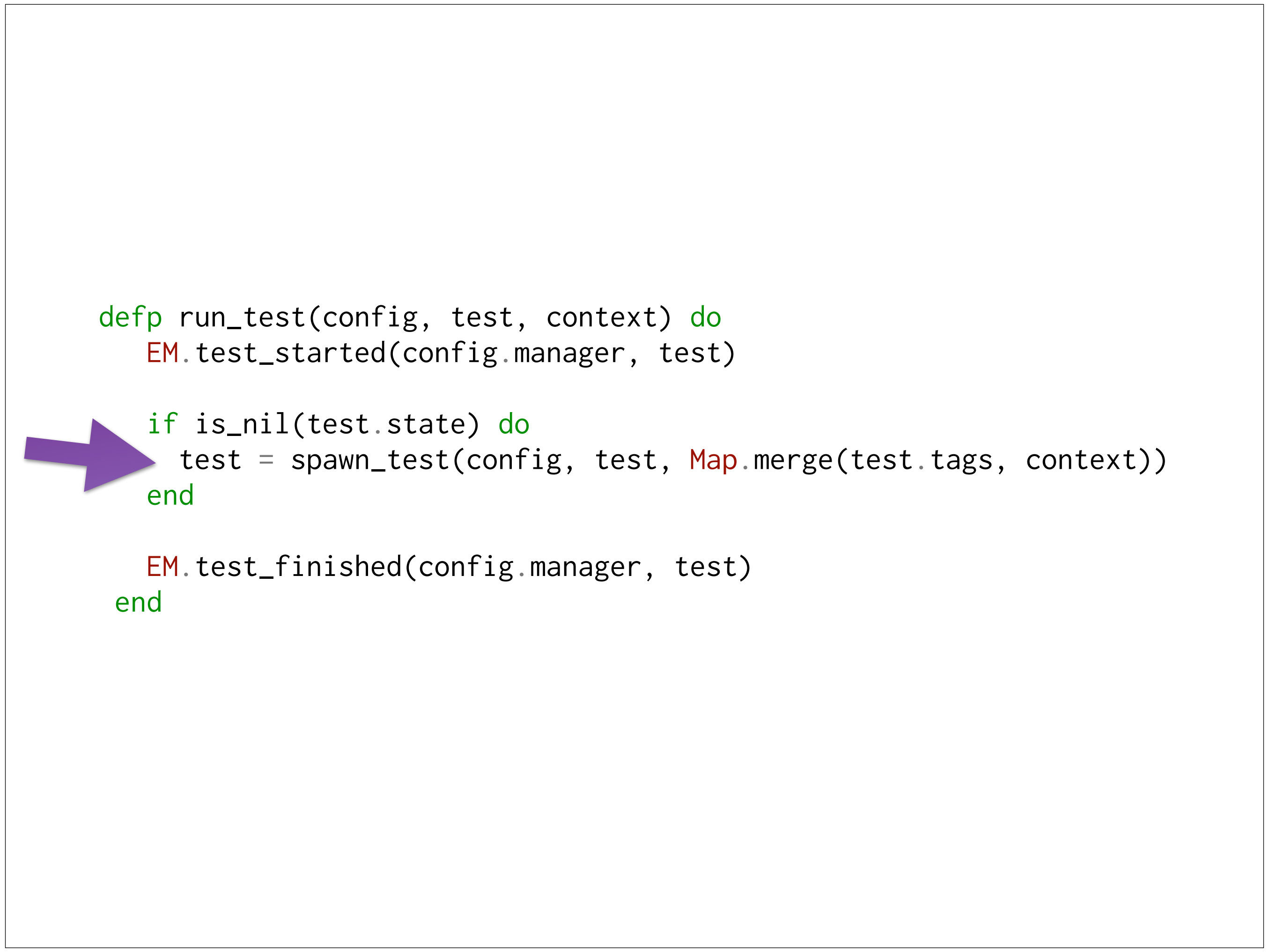


```
defp run_case(config, pid, case_name) do
   test_case = case_name.__ex_unit__(:case)
   EM.case_started(config.manager, test_case)

   # Prepare tests, selecting which ones should
   # run and which ones were skipped.
   tests = prepare_tests(config, test_case.tests)

   {test_case, pending} =
     if Enum.all?(tests, &(&1.state)) do
       {test_case, tests}
     else
       spawn_case(config, test_case, tests)
     end

   # Run the pending tests. We don't actually spawn those
   # tests but we do send the notifications to formatter.
   Enum.each pending, &run_test(config, &1, [])
   EM.case_finished(config.manager, test_case)
   send pid, {self, :case_finished, test_case}
 end
```

case_name is actually the test module, a specially defined function returns the %TestCase struct associated with that module

prepare_tests is where the shuffling and filtering (include and exclude options) occurs

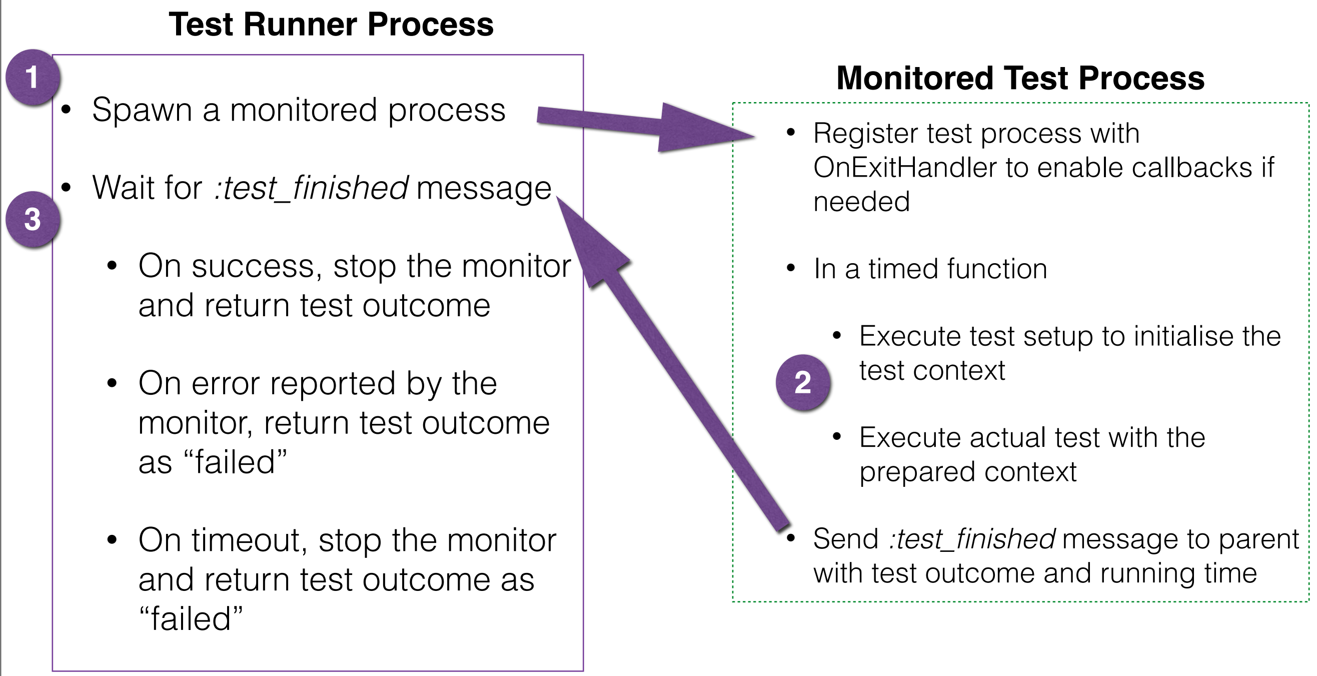
at the end, the runner sends itself a message to be received by *wait_until_available*



```elixir
defp run_test(config, test, context) do
   EM.test_started(config.manager, test)

   if is_nil(test.state) do
     test = spawn_test(config, test, Map.merge(test.tags, context))
   end

   EM.test_finished(config.manager, test)
 end
```

The test may have been marked as “skipped” by prepare_tests, we only spawn the test if its state is nil.

Test Process Algorithm



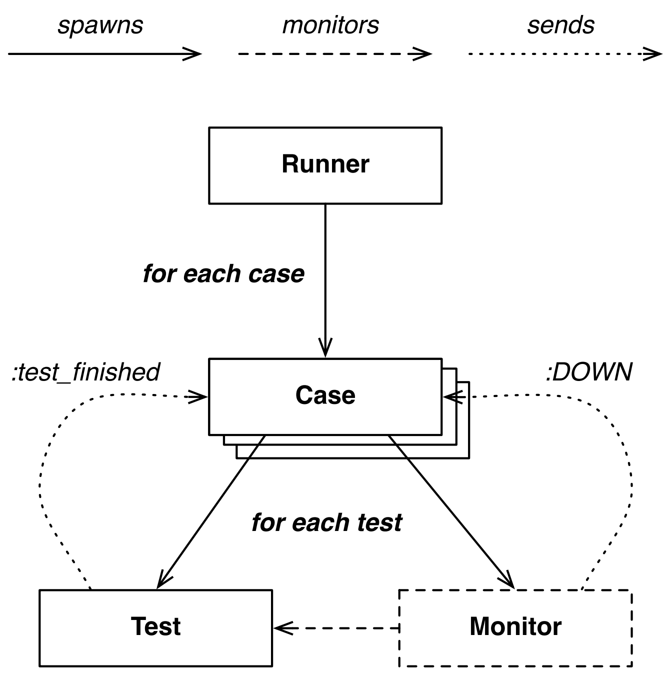
Test Runner Process

- Spawn a monitored process
- Wait for *:test_finished* message
    - On success, stop the monitor and return test outcome
    - On error reported by the monitor, return test outcome as “failed”
    - On timeout, stop the monitor and return test outcome as “failed”

Monitored Test Process

- Register test process with OnExitHandler to enable callbacks if needed
- In a timed function
    - Execute test setup to initialise the test context
    - Execute actual test with the prepared context
- Send *:test_finished* message to parent with test outcome and running time

Process Structure



spawns

monitors

sends

Runner

for each case

Case

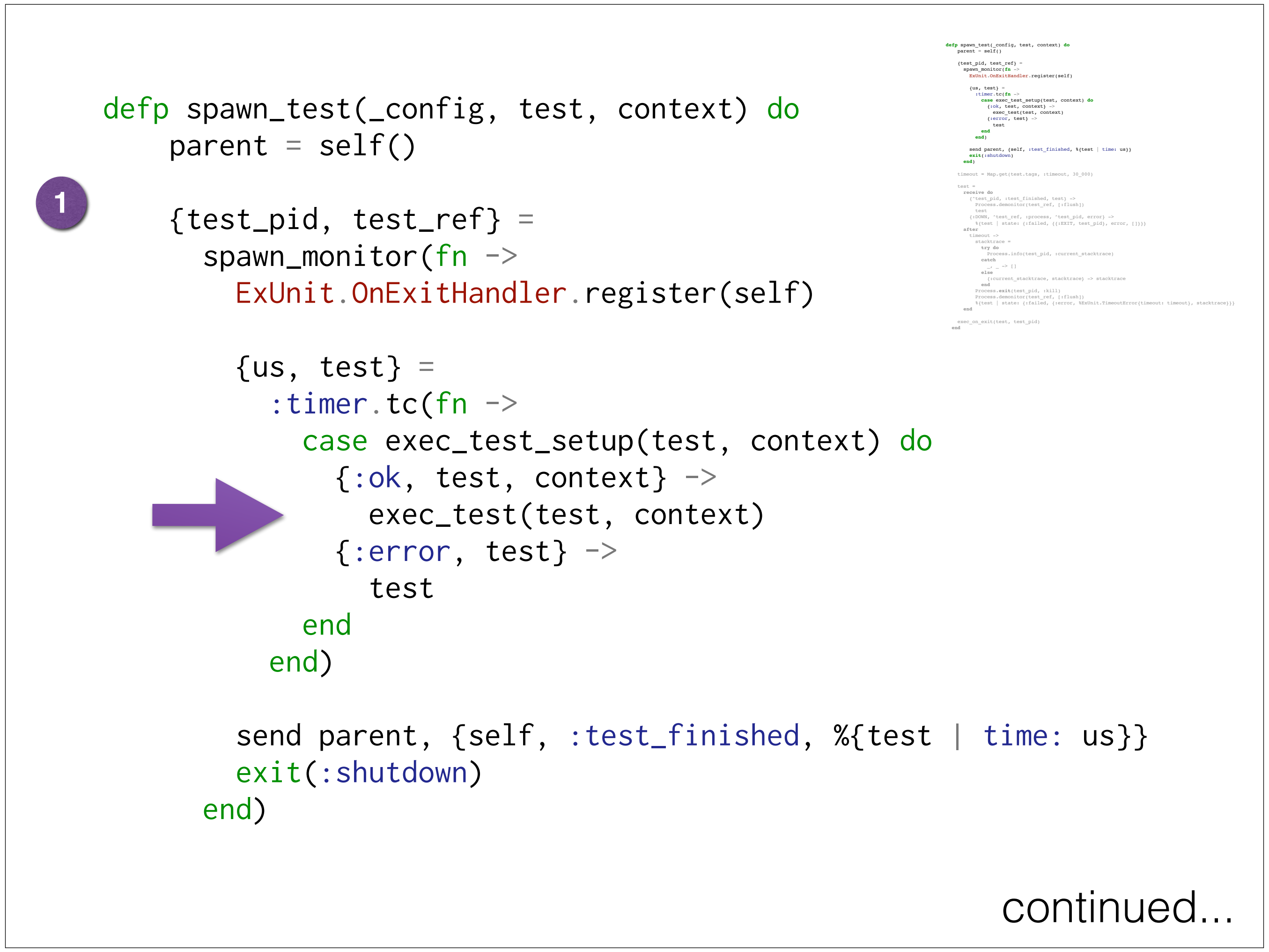
:test_finished

:DOWN

for each test

Test

Monitor



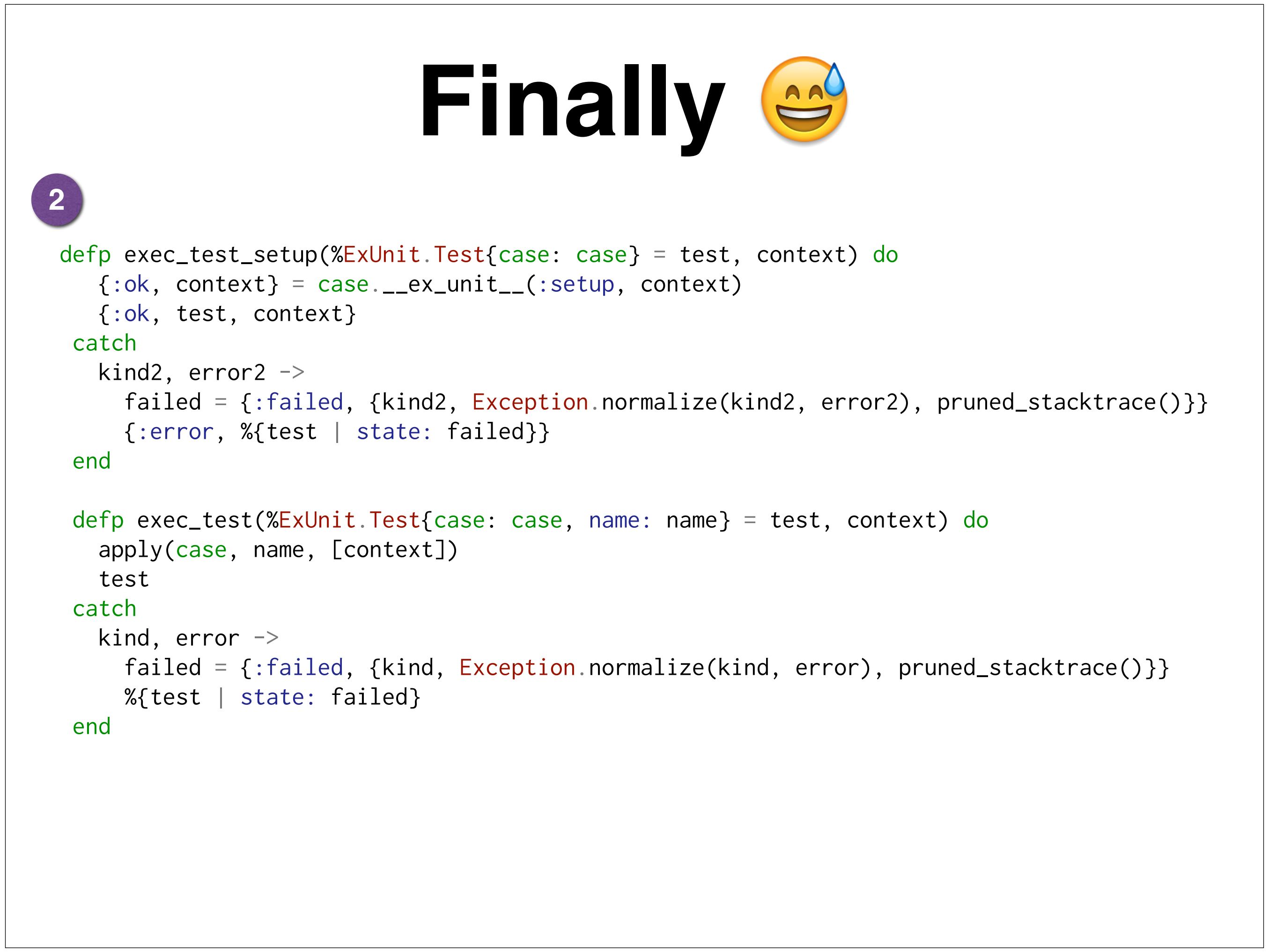
1

```
defp spawn_test(_config, test, context) do
    parent = self()

    {test_pid, test_ref} =
      spawn_monitor(fn ->
        ExUnit.OnExitHandler.register(self)

        {us, test} =
          :timer.tc(fn ->
            case exec_test_setup(test, context) do
              {:ok, test, context} ->
                exec_test(test, context)
              {:error, test} ->
                test
            end
          end)

        send parent, {self, :test_finished, %{test | time: us}}
        exit(:shutdown)
      end)
```

```
defp spawn_test(_config, test, context) do
    parent = self()

    {test_pid, test_ref} =
      spawn_monitor(fn ->
        ExUnit.OnExitHandler.register(self)

        {us, test} =
          :timer.tc(fn ->
            case exec_test_setup(test, context) do
              {:ok, test, context} ->
                exec_test(test, context)
              {:error, test} ->
                test
            end
          end)

        send parent, {self, :test_finished, %{test | time: us}}
        exit(:shutdown)
      end)

    timeout = Map.get(test.tags, :timeout, 30_000)

    test =
      receive do
        {^test_pid, :test_finished, test} ->
          Process.demonitor(test_ref, [:flush])
          test
        {:DOWN, ^test_ref, :process, ^test_pid, error} ->
          %{test | state: {:failed, {{:EXIT, test_pid}, error, []}}}
      after
        timeout ->
          stacktrace =
            try do
              Process.info(test_pid, :current_stacktrace)
            catch
              _, _ -> []
            else
              {:current_stacktrace, stacktrace} -> stacktrace
            end
          Process.exit(test_pid, :kill)
          Process.demonitor(test_ref, [:flush])
          %{test | state: {:failed, {:error, %ExUnit.TimeoutError{timeout: timeout}, stacktrace}}}
      end

    exec_on_exit(test, test_pid)
  end
```

continued...

In addition to the PID spawn_monitor returns a reference to the process monitor



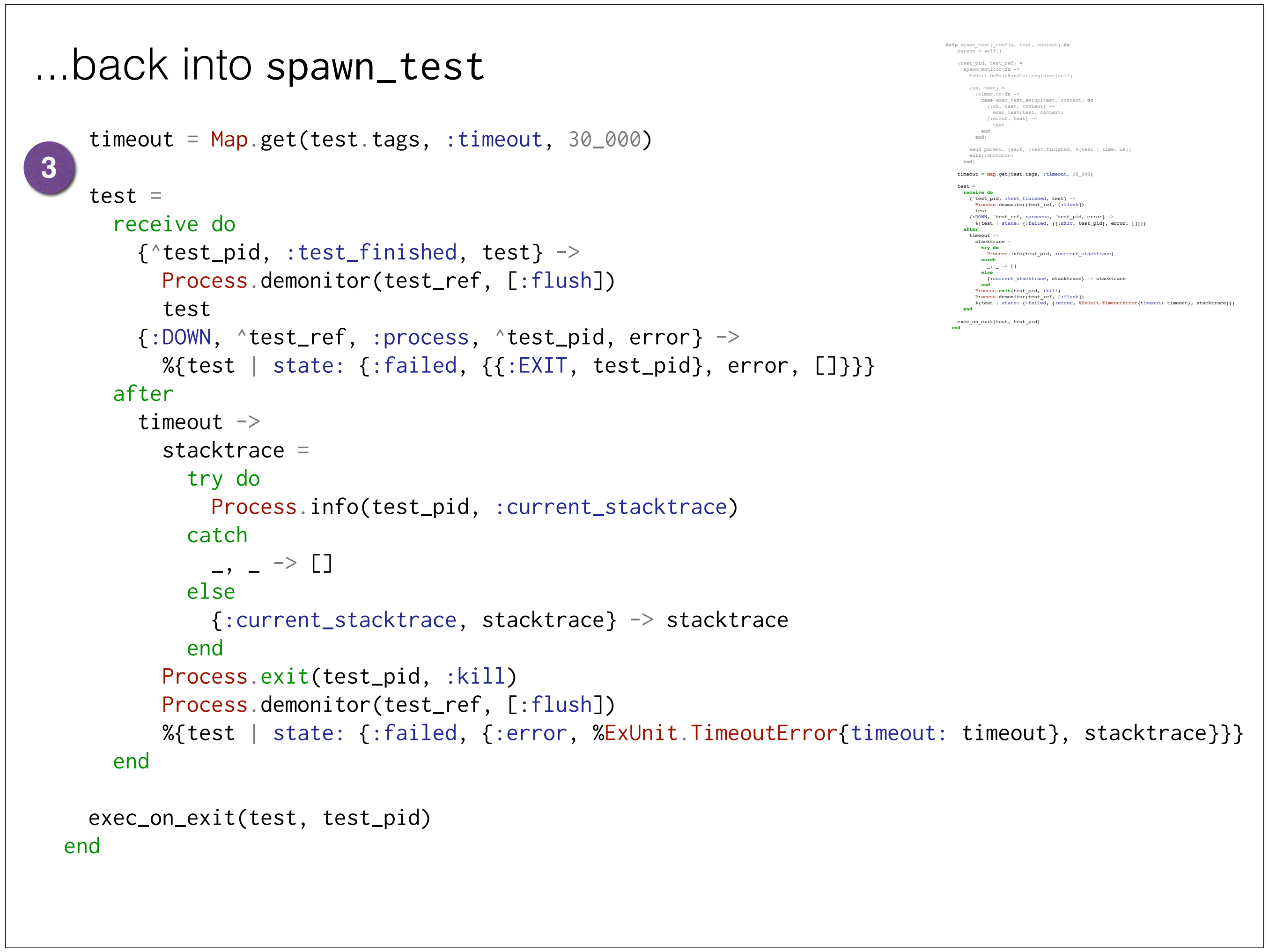
Finally 😅

2

```
defp exec_test_setup(%ExUnit.Test{case: case} = test, context) do
  {:ok, context} = case.__ex_unit__(:setup, context)
  {:ok, test, context}
catch
  kind2, error2 ->
    failed = {:failed, {kind2, Exception.normalize(kind2, error2), pruned_stacktrace()}}
    {:error, %{test | state: failed}}
end

defp exec_test(%ExUnit.Test{case: case, name: name} = test, context) do
  apply(case, name, [context])
  test
catch
  kind, error ->
    failed = {:failed, {kind, Exception.normalize(kind, error), pruned_stacktrace()}}
    %{test | state: failed}
end
```

exec_test_setup calls the setup function of the test case to obtain the context

The test function is called using apply(module, fun, args)



...back into spawn_test

3

```
 timeout = Map.get(test.tags, :timeout, 30_000)

 test =
  receive do
   {^test_pid, :test_finished, test} ->
    Process.demonitor(test_ref, [:flush])
    test
   {:DOWN, ^test_ref, :process, ^test_pid, error} ->
    %{test | state: {:failed, {{:EXIT, test_pid}, error, []}}}
  after
   timeout ->
    stacktrace =
     try do
      Process.info(test_pid, :current_stacktrace)
     catch
      _, _ -> []
     else
      {:current_stacktrace, stacktrace} -> stacktrace
     end
    Process.exit(test_pid, :kill)
    Process.demonitor(test_ref, [:flush])
    %{test | state: {:failed, {:error, %ExUnit.TimeoutError{timeout: timeout}, stacktrace}}}
  end

 exec_on_exit(test, test_pid)
end
```

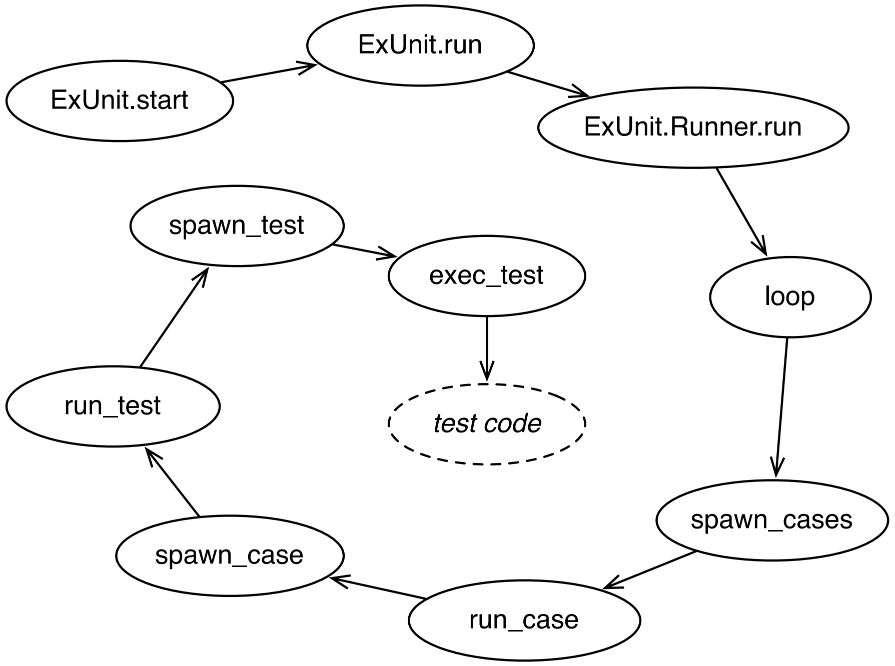
On success, we stop the monitor

If the test process went down, mark the test as failed and save the reason

On timeout, try to get the stacktrace from the test process to find where it’s stuck

Make sure we perform any user-defined clean up with exec_on_exit (cfr OnExitCallback)

The Journey So Far



Great, but...

- How did we get our test code in there?
- How are those test functions created?
- How are the sync and async test cases collected?
- Where does the `__ex_unit__` function come from?

Test Cases

```
defmodule ExampleTest do

  use ExUnit.Case, async: false

  test ”assert example” do
    assert 1 = 1
  end

end
```

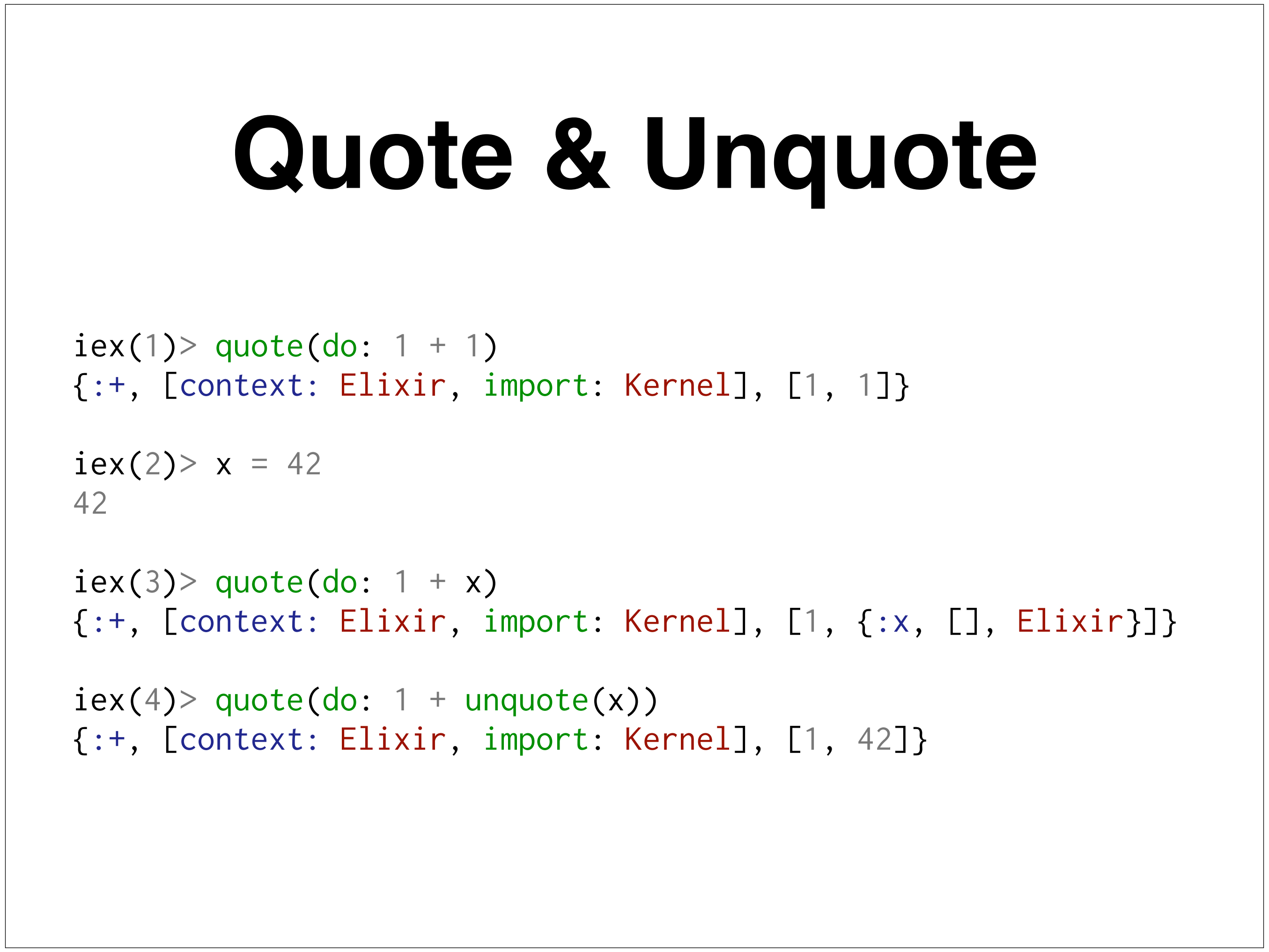
ExUnit Heavily Uses Metaprogramming

- Externally
  - To provide a **simple DSL** for test authors
  - To give **detailed feedback** thanks to code introspection
- Internally
  - To collect **lists of cases and tests**
  - To **wrap user code** into more convenient executable units
  - To keep repetitive code snippets **DRY**

Macros Crash Course

Macros

- **Compile-time** metaprogramming
- Code generation through **internal representation manipulation** using `quote` / `unquote`
- Macros are functions which receive and return **quoted expressions**



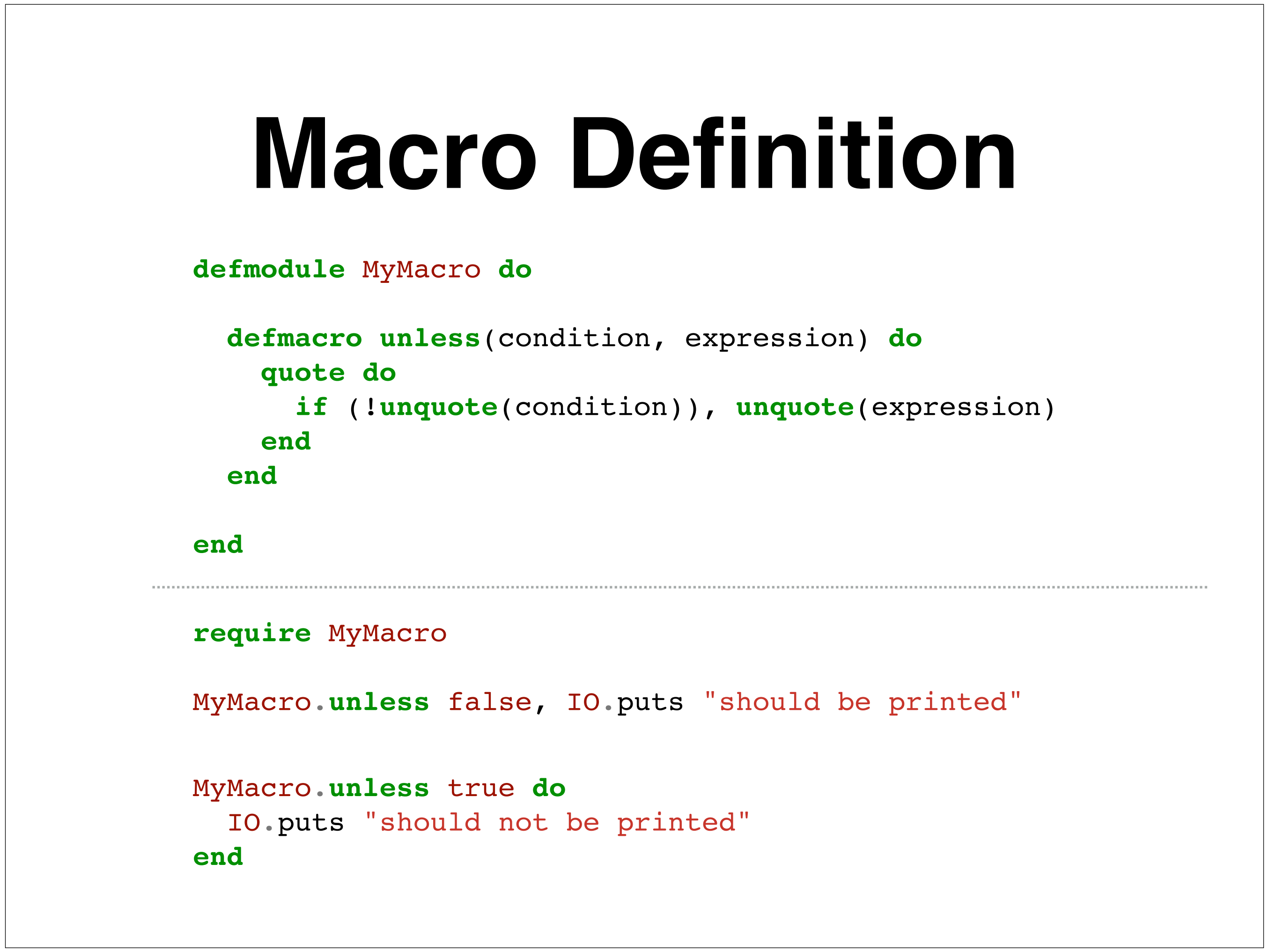
Quote & Unquote

```
iex(1)> quote(do: 1 + 1)
{:+, [context: Elixir, import: Kernel], [1, 1]}

iex(2)> x = 42
42

iex(3)> quote(do: 1 + x)
{:+, [context: Elixir, import: Kernel], [1, {:x, [], Elixir}]}

iex(4)> quote(do: 1 + unquote(x))
{:+, [context: Elixir, import: Kernel], [1, 42]}
```

quote and unquote are special forms (e.g. it looks like a regular function call but it’s actually some building block of the language which cannot be overridden)

quote returns the internal representation of the given expression

unquote can only be called from inside a macro and it will inject the given expression as-is in the AST



Macro Definition

```
defmodule MyMacro do

  defmacro unless(condition, expression) do
    quote do
      if (!unquote(condition)), unquote(expression)
    end
  end

end
```

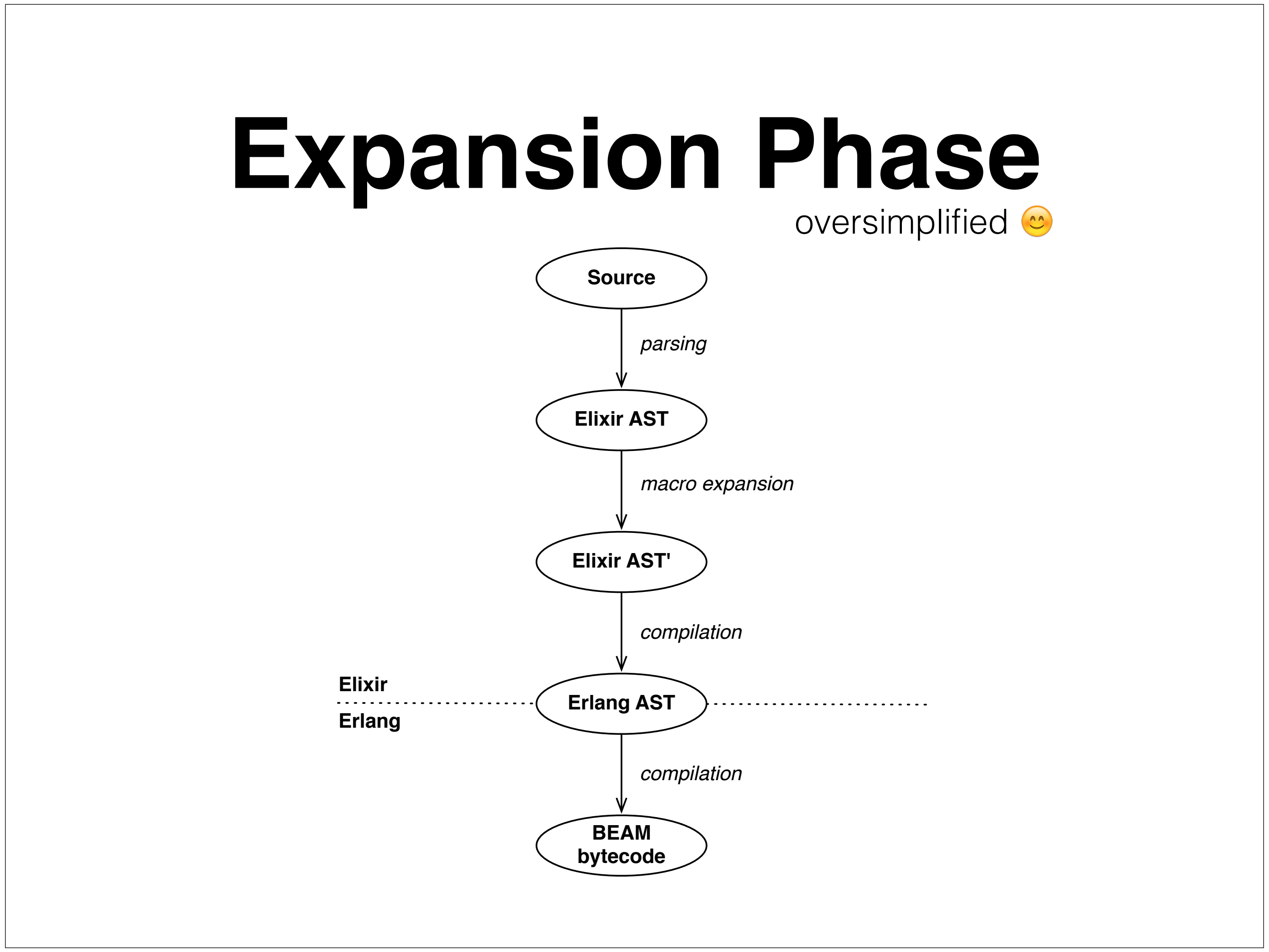
```
require MyMacro

MyMacro.unless false, IO.puts "should be printed"


MyMacro.unless true do
  IO.puts "should not be printed"
end
```

Note: the module must be compiled to trigger the macro expansion

Macro Expansion

```
MyMacro.unless false, IO.puts "should be printed"
```

```
# First argument passed to the macro: quote(do: false)
false

# Second argument passed to the macro: quote(do: IO.puts “should be printed”)
[do: {{:., [line: 5], [{:__aliases__, [counter: 0, line: 5], [:IO]}, :puts]},
  [line: 5], ["should be printed"]}]

# Return value of the macro:
#   quote do
#     if (!unquote(condition)), unquote(expression)
#   end
{:if, [context: MyMacro, import: Kernel],
 [{:__block__, [], [{:!, [context: MyMacro, import: Kernel], [false]}]},
  [do: {{:., [line: 5], [{:__aliases__, [counter: 0, line: 5], [:IO]}, :puts]},
    [line: 5], ["should be printed"]}]]}
```



Expansion Phase

oversimplified 😊

Source

parsing

Elixir AST

macro expansion

Elixir AST'

compilation

Elixir

Erlang AST

Erlang

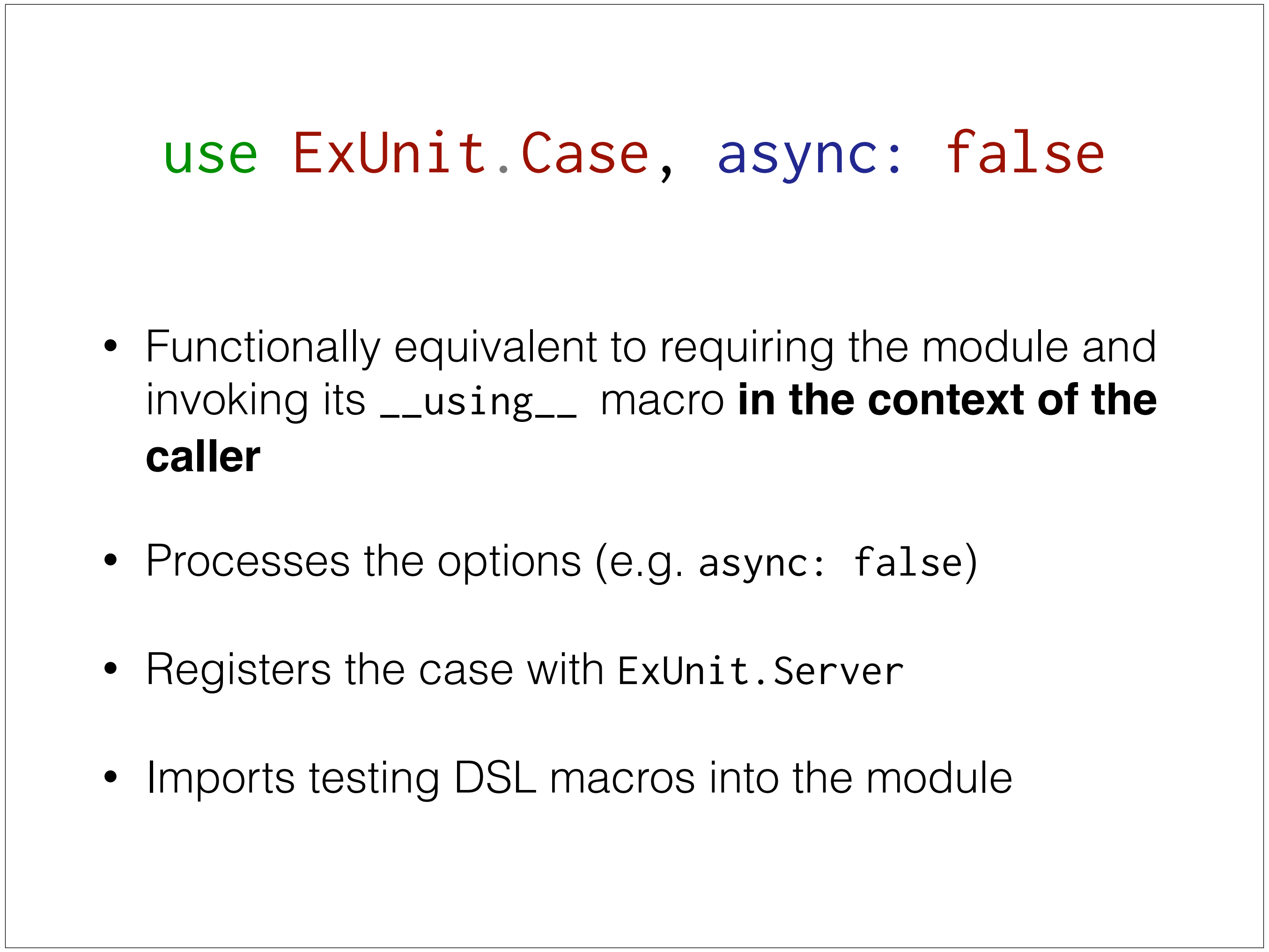
compilation

BEAM bytecode

In the first phase macro definition are just parsed as any elixir code

In a second phase, the compiler expands all the macros until there’s no code left to expand

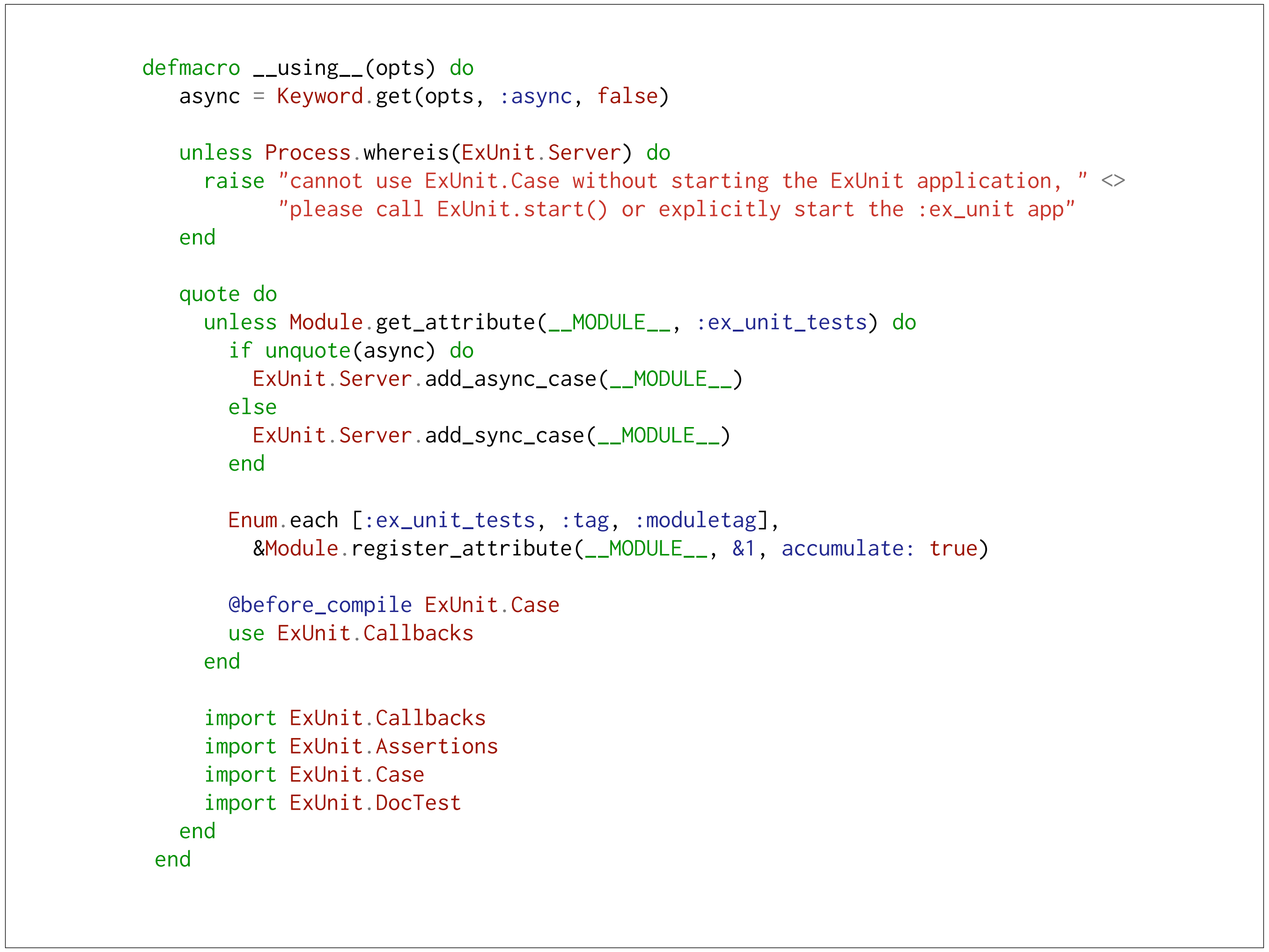
Eventually the Elixir compiler transforms the Elixir AST into Erlang Abstract Format which is then processed by the Erlang compiler to produce bytecode



```
use ExUnit.Case, async: false
```

- Functionally equivalent to requiring the module and invoking its `__using__` macro **in the context of the caller**
- Processes the options (e.g. `async: false`)
- Registers the case with `ExUnit.Server`
- Imports testing DSL macros into the module

The use statement calls the __using__ macro on the ExUnit.Case module



```
defmacro __using__(opts) do
  async = Keyword.get(opts, :async, false)

  unless Process.whereis(ExUnit.Server) do
    raise "cannot use ExUnit.Case without starting the ExUnit application, " <>
          "please call ExUnit.start() or explicitly start the :ex_unit app"
  end

  quote do
    unless Module.get_attribute(__MODULE__, :ex_unit_tests) do
      if unquote(async) do
        ExUnit.Server.add_async_case(__MODULE__)
      else
        ExUnit.Server.add_sync_case(__MODULE__)
      end

      Enum.each [:ex_unit_tests, :tag, :moduletag],
        &Module.register_attribute(__MODULE__, &1, accumulate: true)

      @before_compile ExUnit.Case
      use ExUnit.Callbacks
    end

    import ExUnit.Callbacks
    import ExUnit.Assertions
    import ExUnit.Case
    import ExUnit.DocTest
  end
end
```

Adding “use ExUnit.Case” to a module will execute this function in the context of the host module (similar in some way to MixIns in other languages)

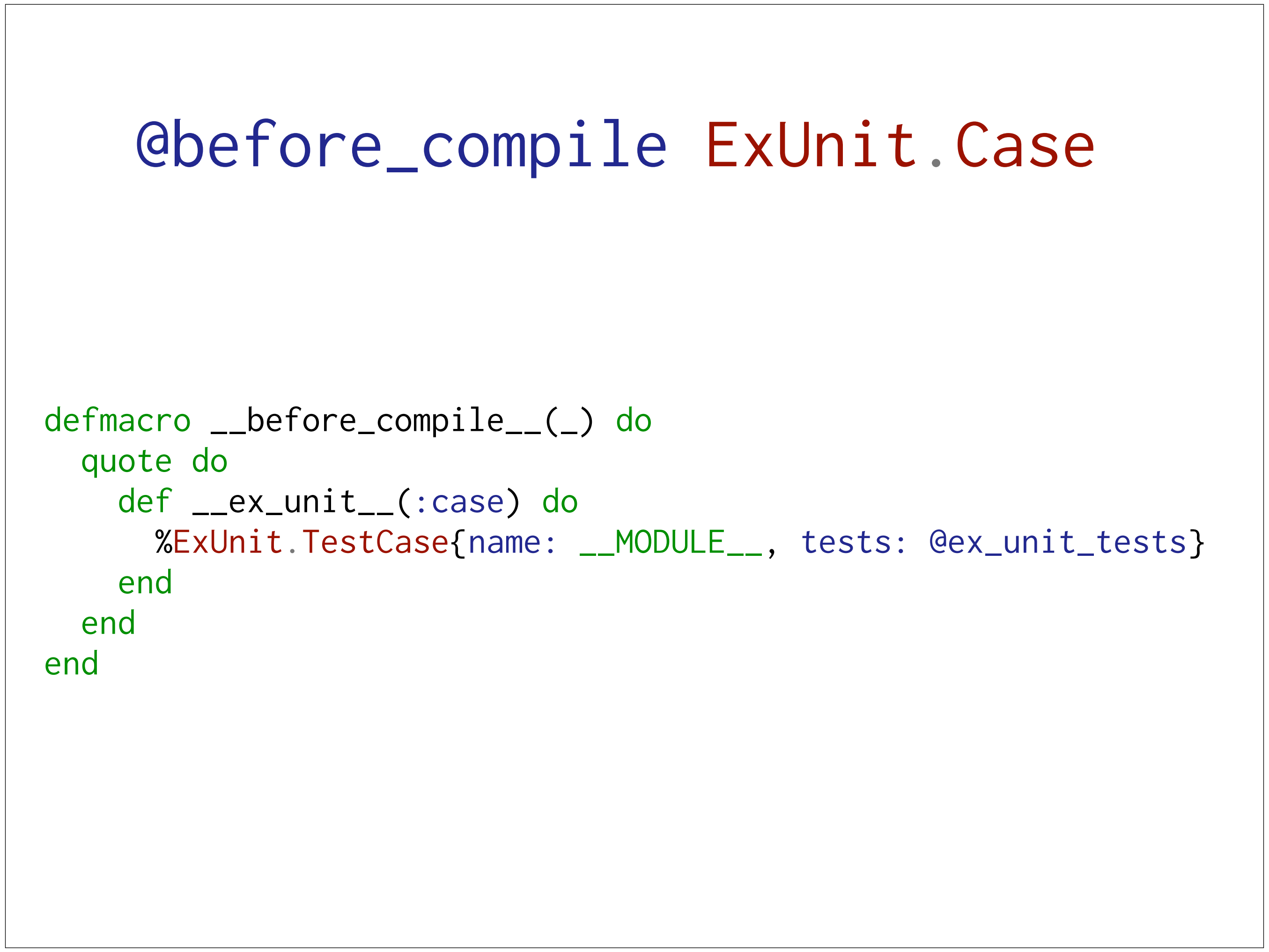
Note that __MODULE__ is **not** unquoted, **why?** -> because otherwise its value would always be ExUnit.Case instead of the actual module where the tests are defined

Create new module attributes to hold test cases and tags, these attributes will be a list of values (accumulate: true)

The @ex_unit_tests module attribute will contain a list of %ExUnit.Test structs

use ExUnit.Callbacks will setup the callback related module attributes by calling __using__ on the Callbacks module

@before_compile will call ExUnit.Case.__before_compile__ when the module definition is complete



@before_compile ExUnit.Case

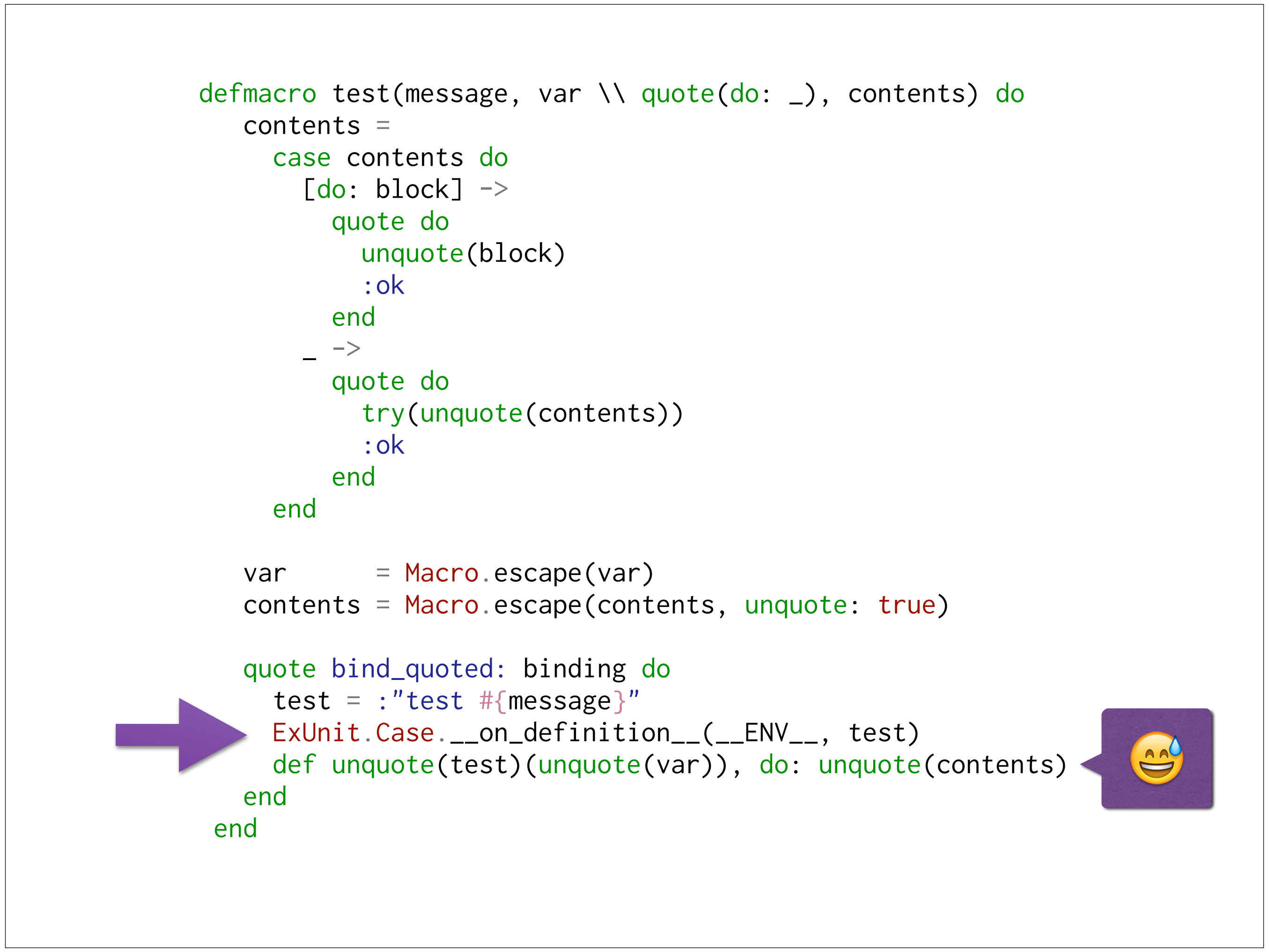
```
defmacro __before_compile__(_) do
  quote do
    def __ex_unit__(:case) do
      %ExUnit.TestCase{name: __MODULE__, tests: @ex_unit_tests}
    end
  end
end
```

Defines the factory function used in run_case to access the list of tests

This callback will be called once the module has been completely defined, so we’re sure that all tests have been collected

```
test ”assert example”, do: assert 1 = 1
```

- May optionally accept a `context` parameter
- Creates a function encapsulating the actual test code
- Registers the test with the test case



```
defmacro test(message, var \\ quote(do: _), contents) do
  contents =
    case contents do
      [do: block] ->
        quote do
          unquote(block)
          :ok
        end
      _ ->
        quote do
          try(unquote(contents))
          :ok
        end
    end

  var      = Macro.escape(var)
  contents = Macro.escape(contents, unquote: true)

  quote bind_quoted: binding do
    test = :"test #{message}"
    ExUnit.Case.__on_definition__(__ENV__, test)
    def unquote(test)(unquote(var)), do: unquote(contents)
  end
end
```

😅

A bit of juggling to support calling the macro with or without context parameter (named "var", for some reason)

The test code can be either passed as a block or inlined as a parameter

The test code is wrapped in order to return :ok when it completes successfully

Macro.escape ensures that a value can be safely inserted in a syntax tree, optionally leaving unquoting unescaped. It basically prevents tuples from being mistakenly taken for AST nodes.

Before actually generating the test function, we register it with __on_definition__

__ENV__ is the current environment of the macro (vs __CALLER__)

Sidenote
About do Blocks

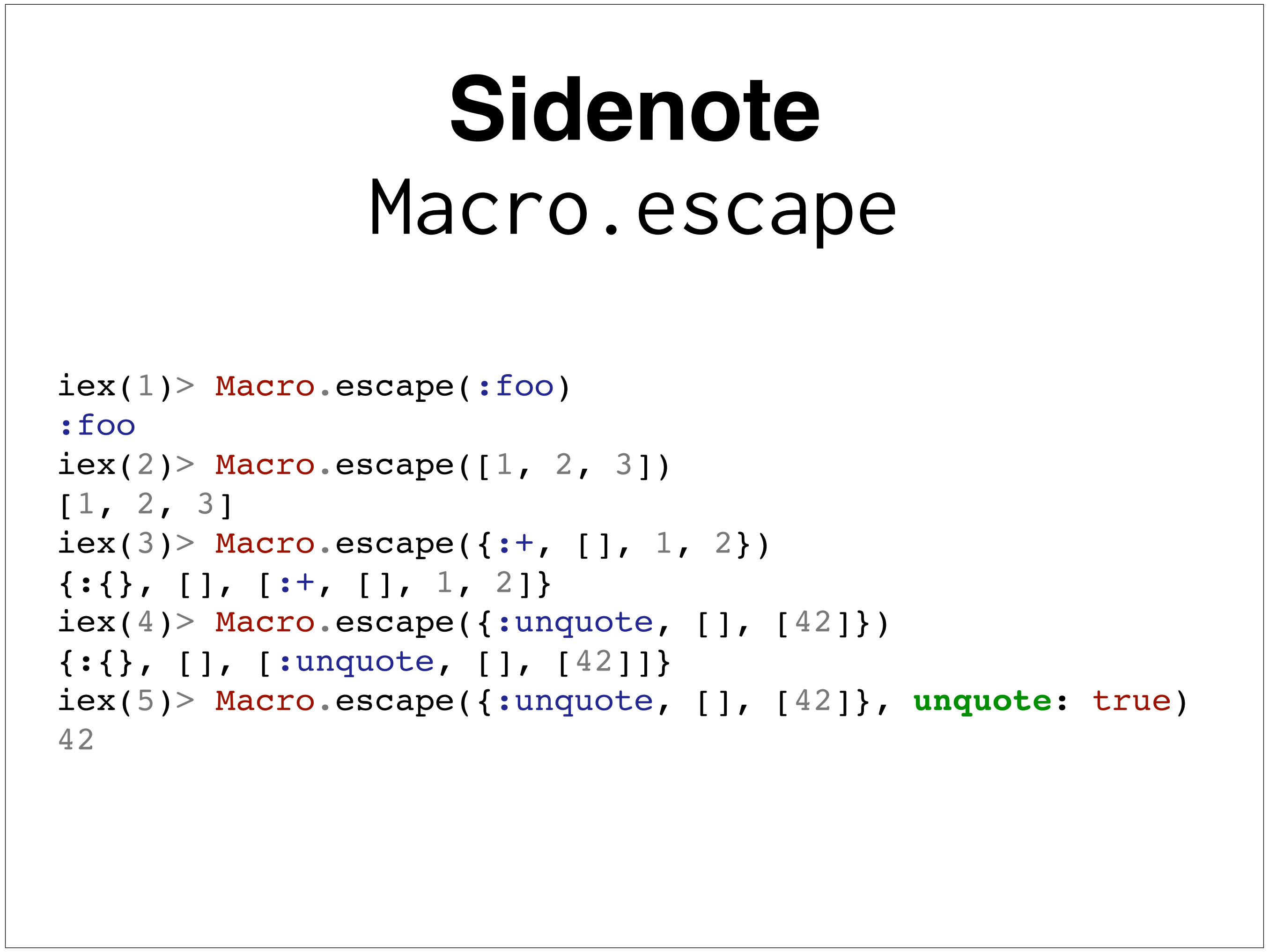
`do/end` blocks are a convenience for passing a group of expressions

actual expression

```
iex> if true, do: 1 + 2
3
iex> if false, do: :this, else: :that
:that
```

syntactic sugar

```
iex> if true do
...>   a = 1 + 2
...>   a + 10
...> end
13
iex> if true, do: (
...>   a = 1 + 2
...>   a + 10
...> )
13
```

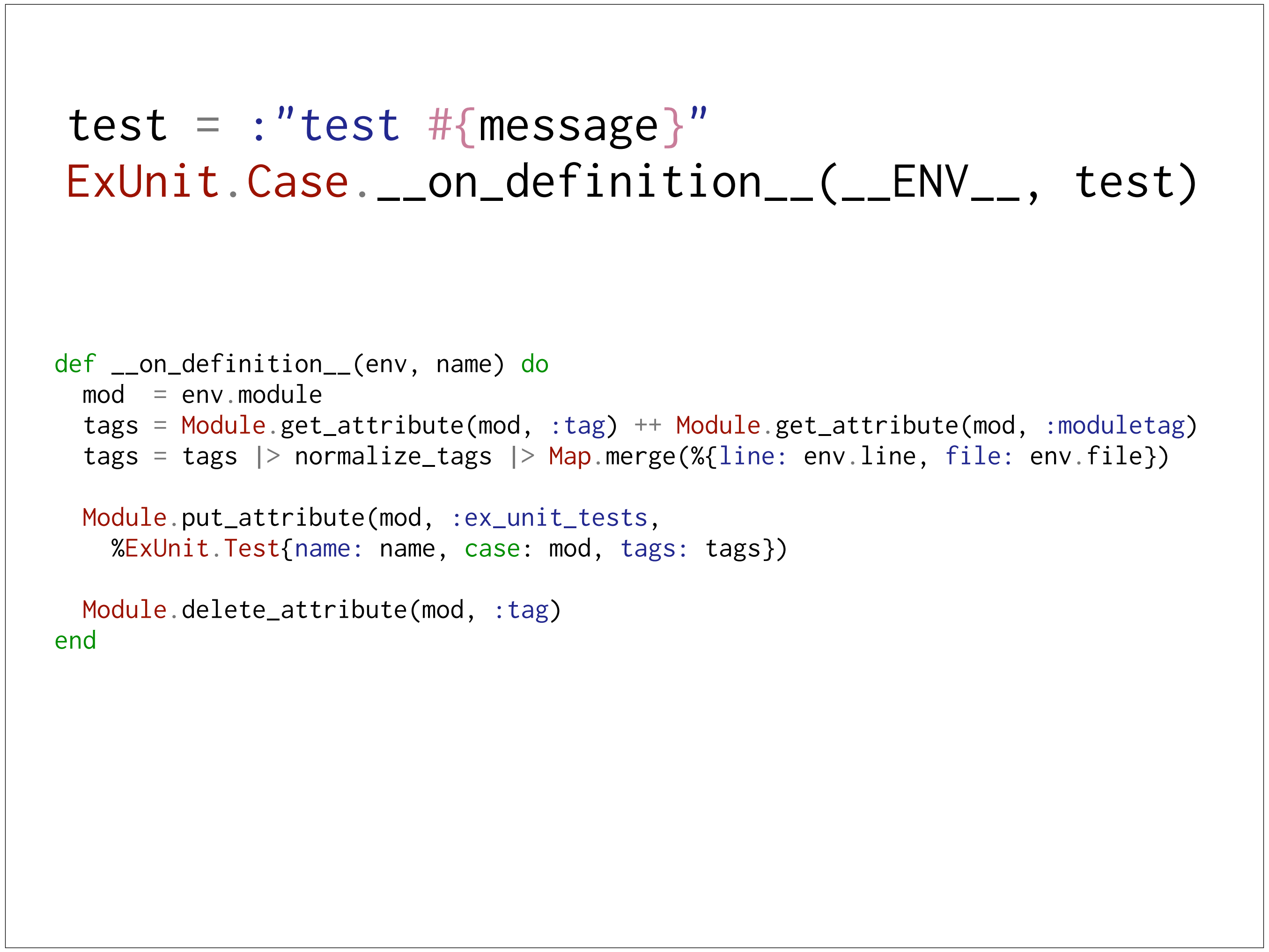


Sidenote

Macro.escape

```
iex(1)> Macro.escape(:foo)
:foo
iex(2)> Macro.escape([1, 2, 3])
[1, 2, 3]
iex(3)> Macro.escape({:+, [], 1, 2})
{:{}, [], [:+, [], 1, 2]}
iex(4)> Macro.escape({:unquote, [], [42]})
{:{}, [], [:unquote, [], [42]]}
iex(5)> Macro.escape({:unquote, [], [42]}, unquote: true)
42
```

By default, unquote clauses are escaped (i.e. quoted as a tuple)



```
test = :"test #{message}"
ExUnit.Case.__on_definition__(__ENV__, test)
```

```
def __on_definition__(env, name) do
  mod  = env.module
  tags = Module.get_attribute(mod, :tag) ++ Module.get_attribute(mod, :moduletag)
  tags = tags |> normalize_tags |> Map.merge(%{line: env.line, file: env.file})

  Module.put_attribute(mod, :ex_unit_tests,
    %ExUnit.Test{name: name, case: mod, tags: tags})

  Module.delete_attribute(mod, :tag)
end
```

Notice how there’s no need to do escaping to turn the string into a function name.

The test is appended to the list of the tests included in the module (remember that @ex_unit_tests has been registered with accumulate: true)

Test Case Definition Summary

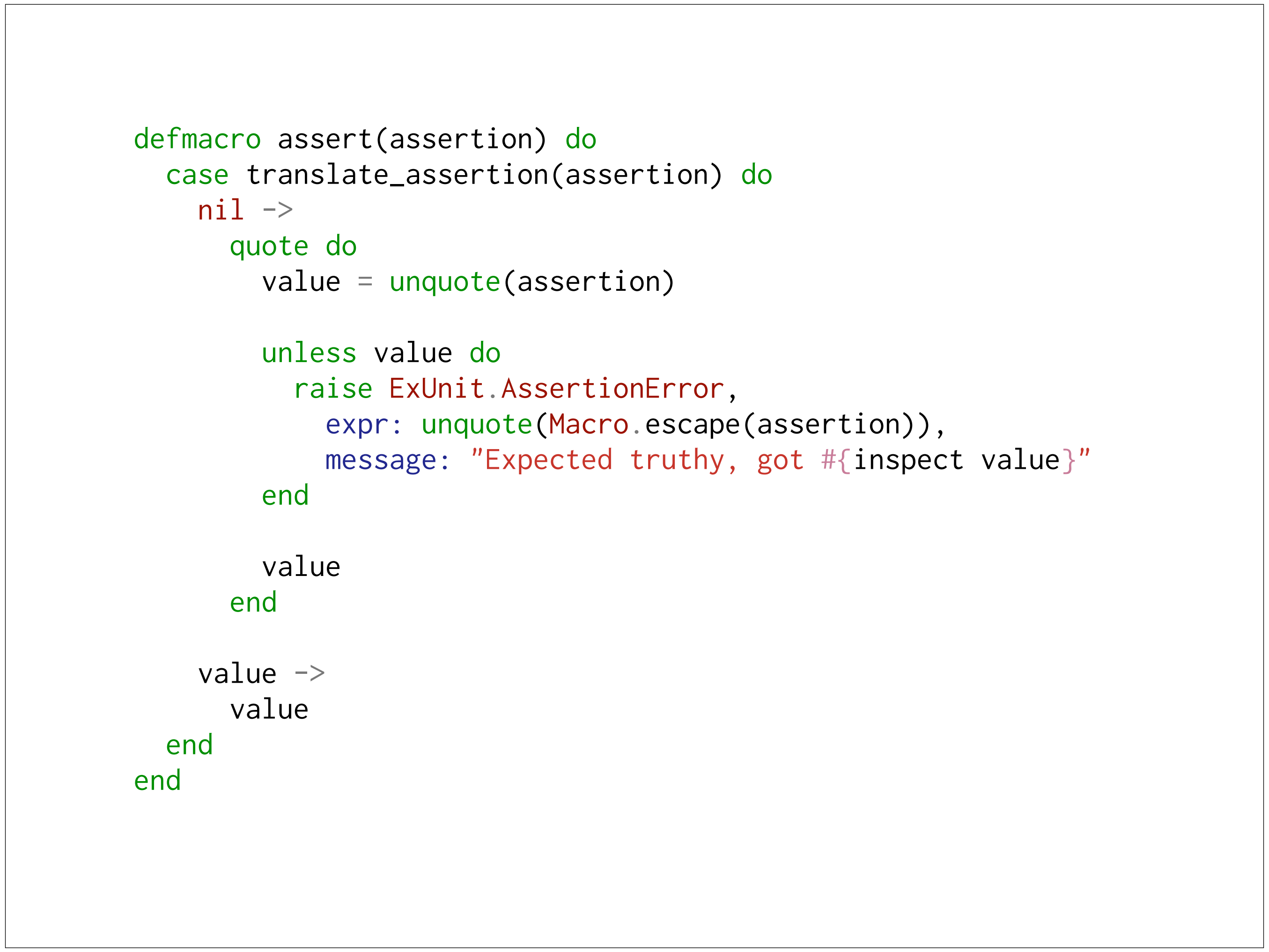
- When `ExUnit.Case` is injected in host module: register case to the `ExUnit.Server`

- When test case macros are expanded, each test function is registered in the module's `@ex_unit_tests`

- When the module definition is complete, a final `__ex_unit__` function is defined to return the populated `Case` struct

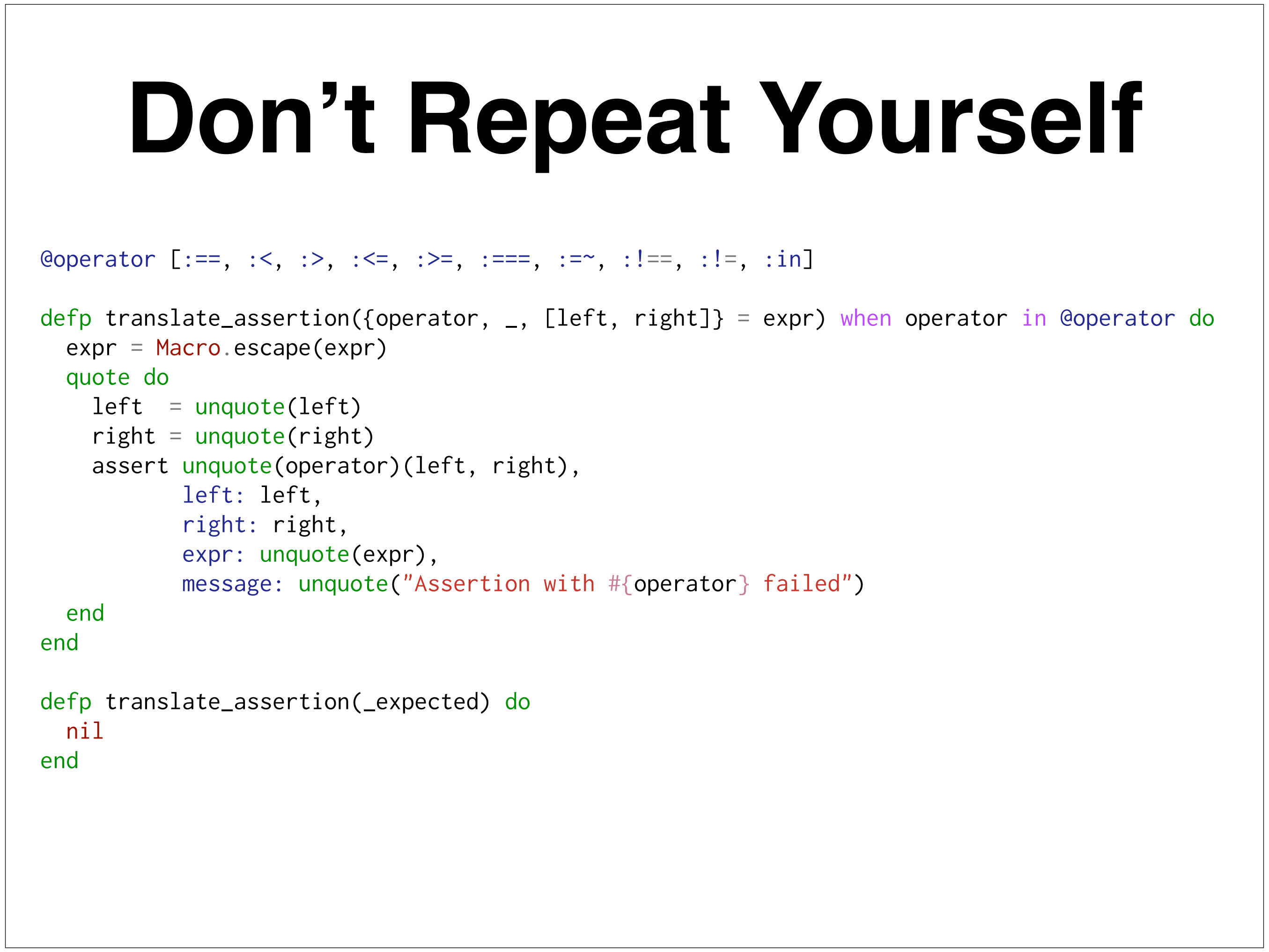
Assertions

- ExUnit actually only implements a couple of trivial cases of assertions:
  - `assert <truthy>`
  - `assert a = b`
- Translate all other assertions into simple `assert` forms
- A few additional custom `assert_` functions to deal with special cases like exceptions, messages expectations, …



```
defmacro assert(assertion) do
  case translate_assertion(assertion) do
    nil ->
      quote do
        value = unquote(assertion)

        unless value do
          raise ExUnit.AssertionError,
            expr: unquote(Macro.escape(assertion)),
            message: "Expected truthy, got #{inspect value}"
        end

        value
      end

    value ->
      value
  end
end
```

If the given assertion was translatable, expand the translated code

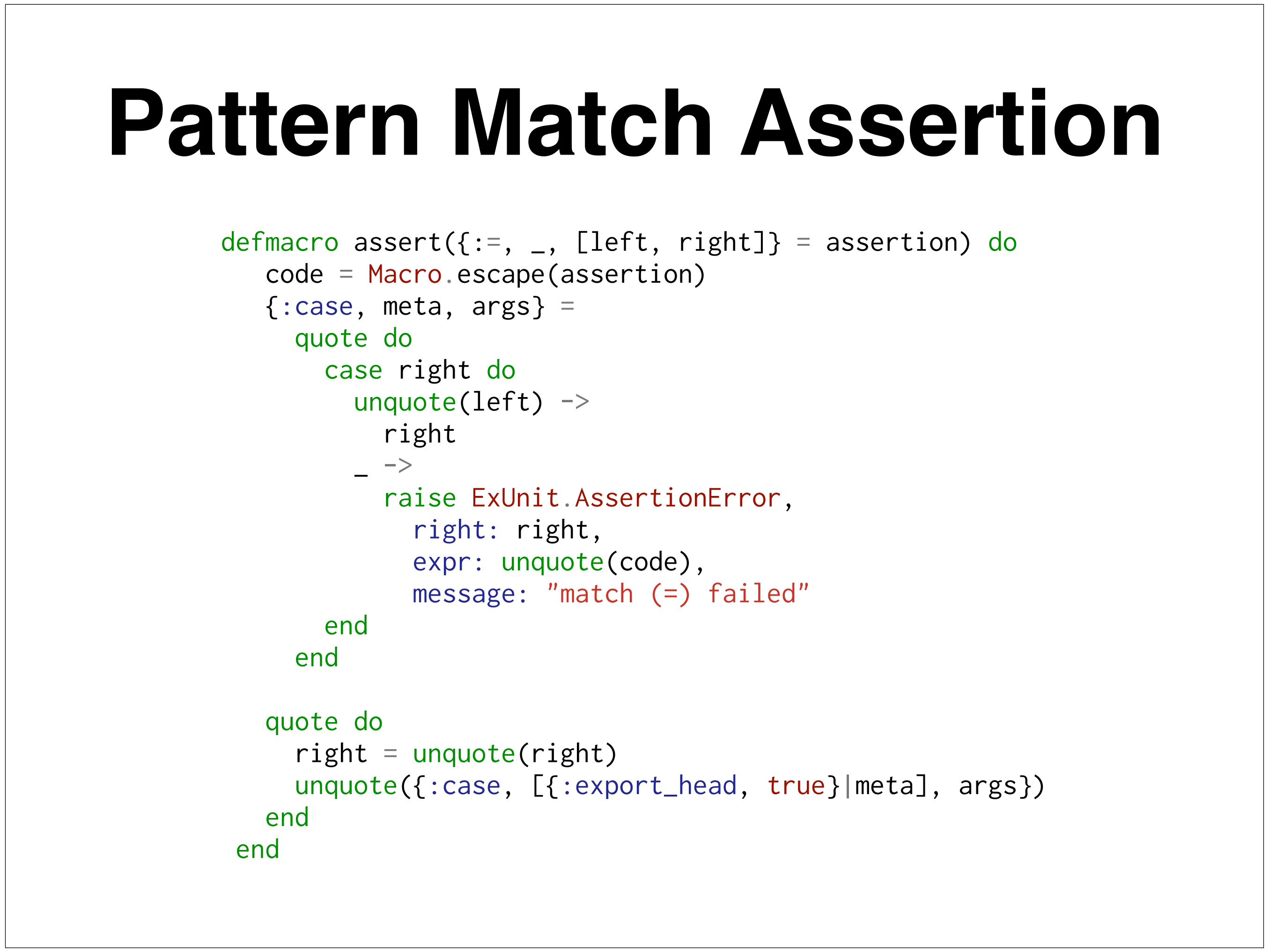


Don’t Repeat Yourself

```
@operator [:==, :<, :>, :<=, :>=, :===, :=~, :!==, :!=, :in]

defp translate_assertion({operator, _, [left, right]} = expr) when operator in @operator do
  expr = Macro.escape(expr)
  quote do
    left  = unquote(left)
    right = unquote(right)
    assert unquote(operator)(left, right),
           left: left,
           right: right,
           expr: unquote(expr),
           message: unquote("Assertion with #{operator} failed”)
  end
end

defp translate_assertion(_expected) do
  nil
end
```

Generate code which uses assert a = b

We ensure that the operator is actually an operator and not the name of any two argument function which would produce the same AST



Pattern Match Assertion

```
defmacro assert({:=, _, [left, right]} = assertion) do
   code = Macro.escape(assertion)
   {:case, meta, args} =
     quote do
       case right do
         unquote(left) ->
           right
         _ ->
           raise ExUnit.AssertionError,
             right: right,
             expr: unquote(code),
             message: "match (=) failed"
       end
     end

   quote do
     right = unquote(right)
     unquote({:case, [{:export_head, true}|meta], args})
   end
 end
```

Macros are functions so you can use pattern matching on the given AST

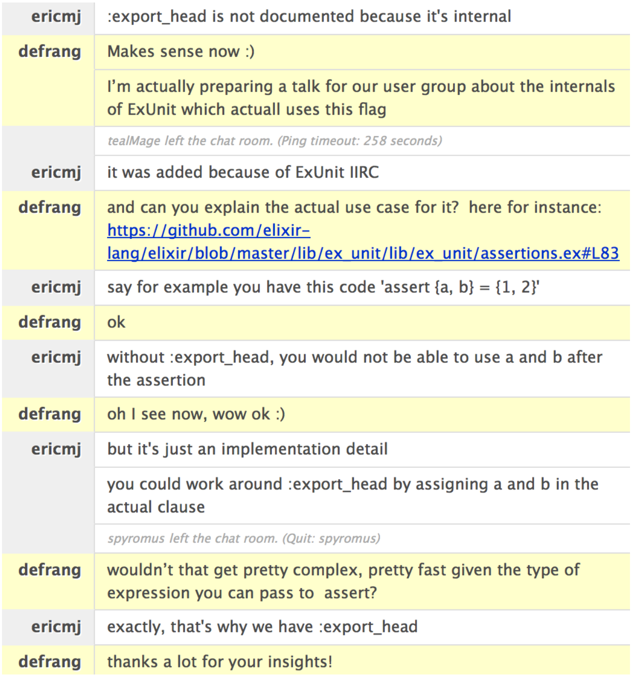
This definition will match “assert left = right”

We build a quoted case statement (with an unquoted right variable), the actual generated code sets up the right variable and then executes the prepared case clause generated back from the AST

the export_head key set to true apparently has something to do with making the variables available to the rest of the function (TBC)

#elixir-lang
to the rescue!

Sidenote
`export_head`



ericmj	:export_head is not documented because it's internal
defrang	Makes sense now :)
	I'm actually preparing a talk for our user group about the internals of ExUnit which actuall uses this flag
	tealMage left the chat room. (Ping timeout: 258 seconds)
ericmj	it was added because of ExUnit IIRC
defrang	and can you explain the actual use case for it? here for instance: https://github.com/elixir-lang/elixir/blob/master/lib/ex_unit/lib/ex_unit/assertions.ex#L83
ericmj	say for example you have this code 'assert {a, b} = {1, 2}'
defrang	ok
ericmj	without :export_head, you would not be able to use a and b after the assertion
defrang	oh I see now, wow ok :)
ericmj	but it's just an implementation detail
	you could work around :export_head by assigning a and b in the actual clause
	spyromus left the chat room. (Quit: spyromus)
defrang	wouldn't that get pretty complex, pretty fast given the type of expression you can pass to assert?
ericmj	exactly, that's why we have :export_head
defrang	thanks a lot for your insights!

TL;DR
macros are **hygienic**

they do not leak variable bindings unless explicitly told so with `export_head`

See "Hygiene in variables"

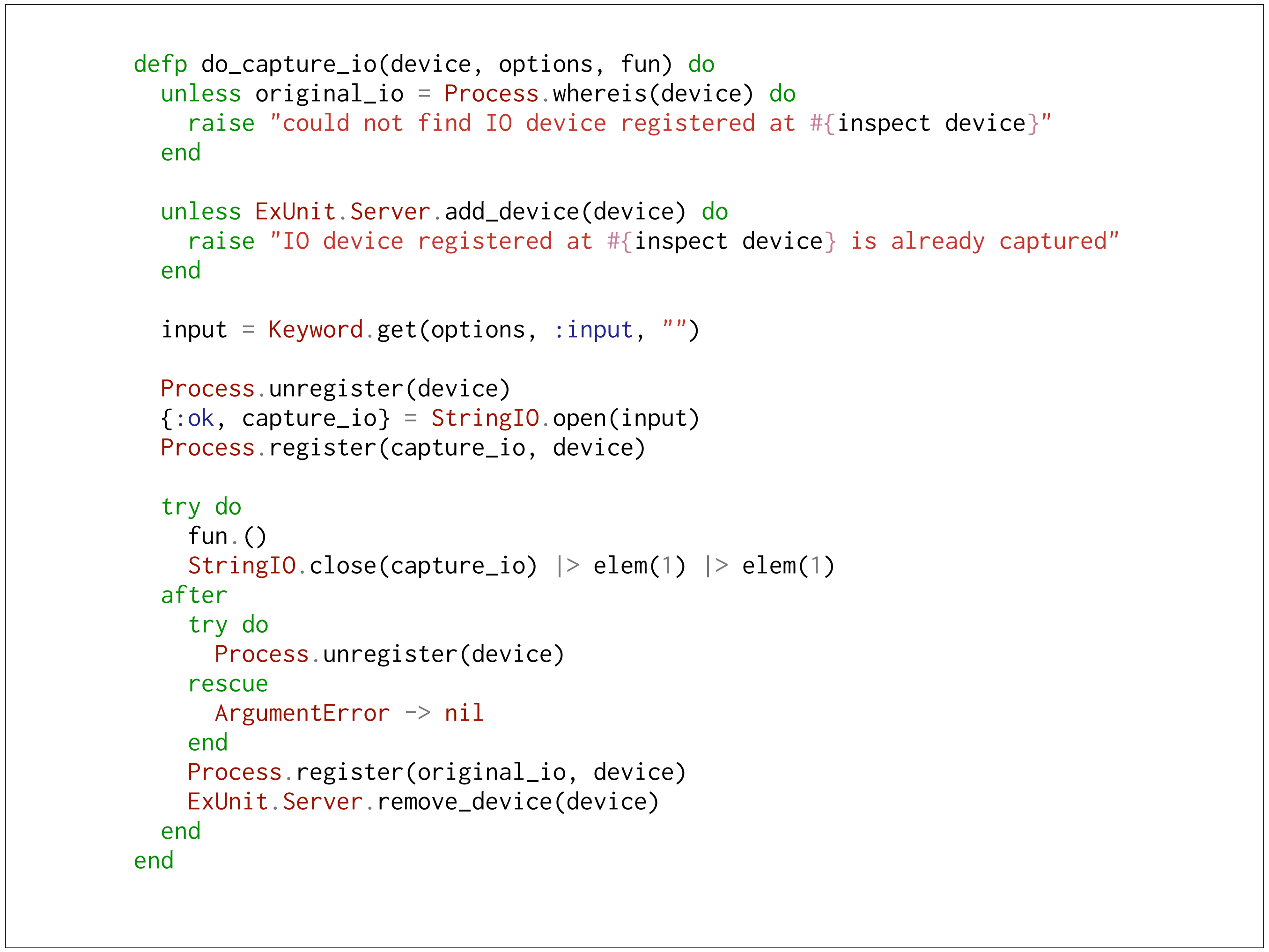
http://elixir-lang.org/docs/stable/elixir/Kernel.SpecialForms.html#quote/2

MACROS ARE AWESOME

Sweet Tricks

Capturing IO

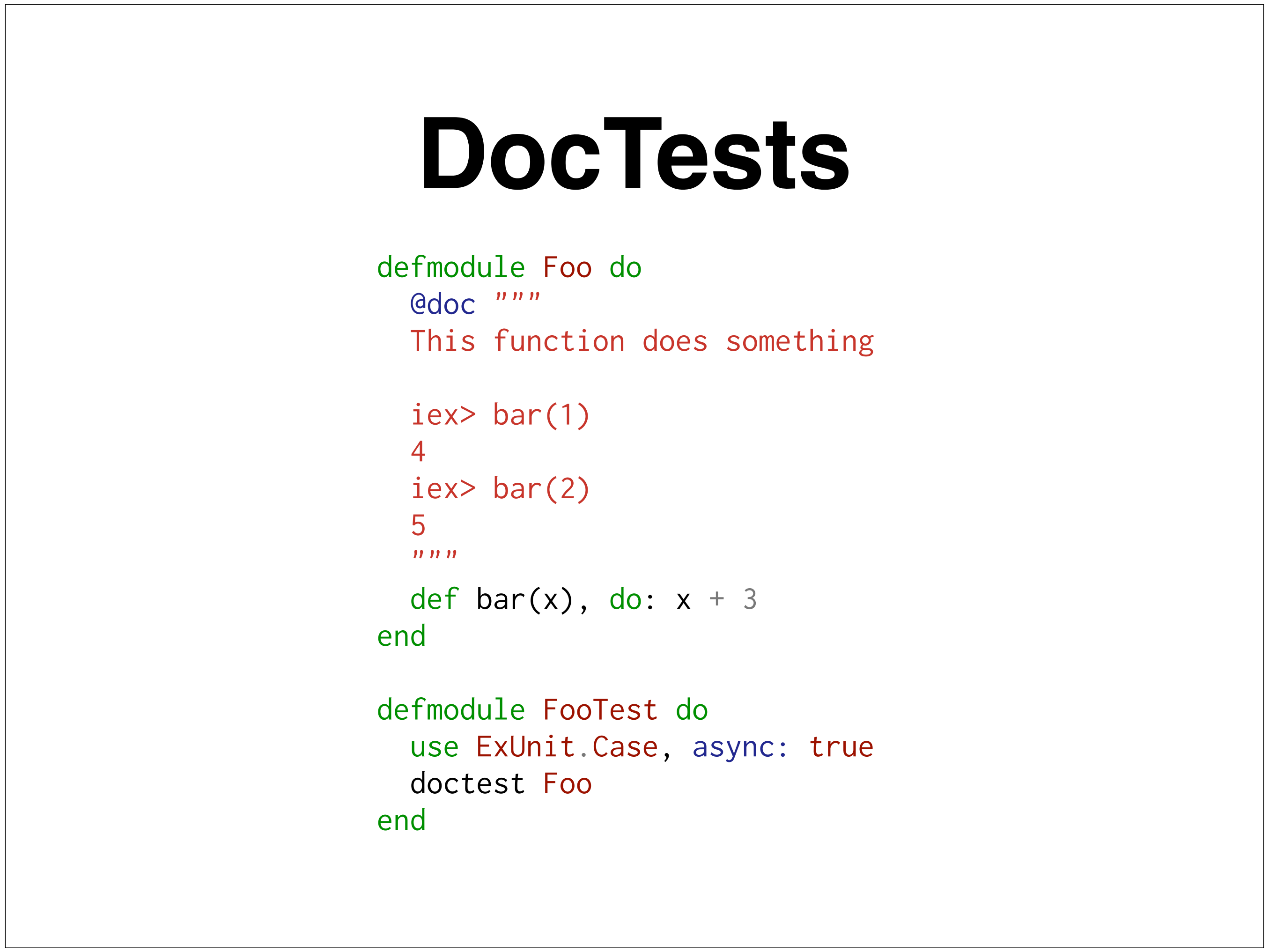
```
test “Capture console output” do
  assert capture_io(fn ->
    IO.puts “a”
  end) == “a\n”
end
```



```elixir
defp do_capture_io(device, options, fun) do
  unless original_io = Process.whereis(device) do
    raise "could not find IO device registered at #{inspect device}"
  end

  unless ExUnit.Server.add_device(device) do
    raise "IO device registered at #{inspect device} is already captured"
  end

  input = Keyword.get(options, :input, "")

  Process.unregister(device)
  {:ok, capture_io} = StringIO.open(input)
  Process.register(capture_io, device)

  try do
    fun.()
    StringIO.close(capture_io) |> elem(1) |> elem(1)
  after
    try do
      Process.unregister(device)
    rescue
      ArgumentError -> nil
    end
    Process.register(original_io, device)
    ExUnit.Server.remove_device(device)
  end
end
```

We simply play switcheroo and temporarily the process managing the given device by our own StringIO process

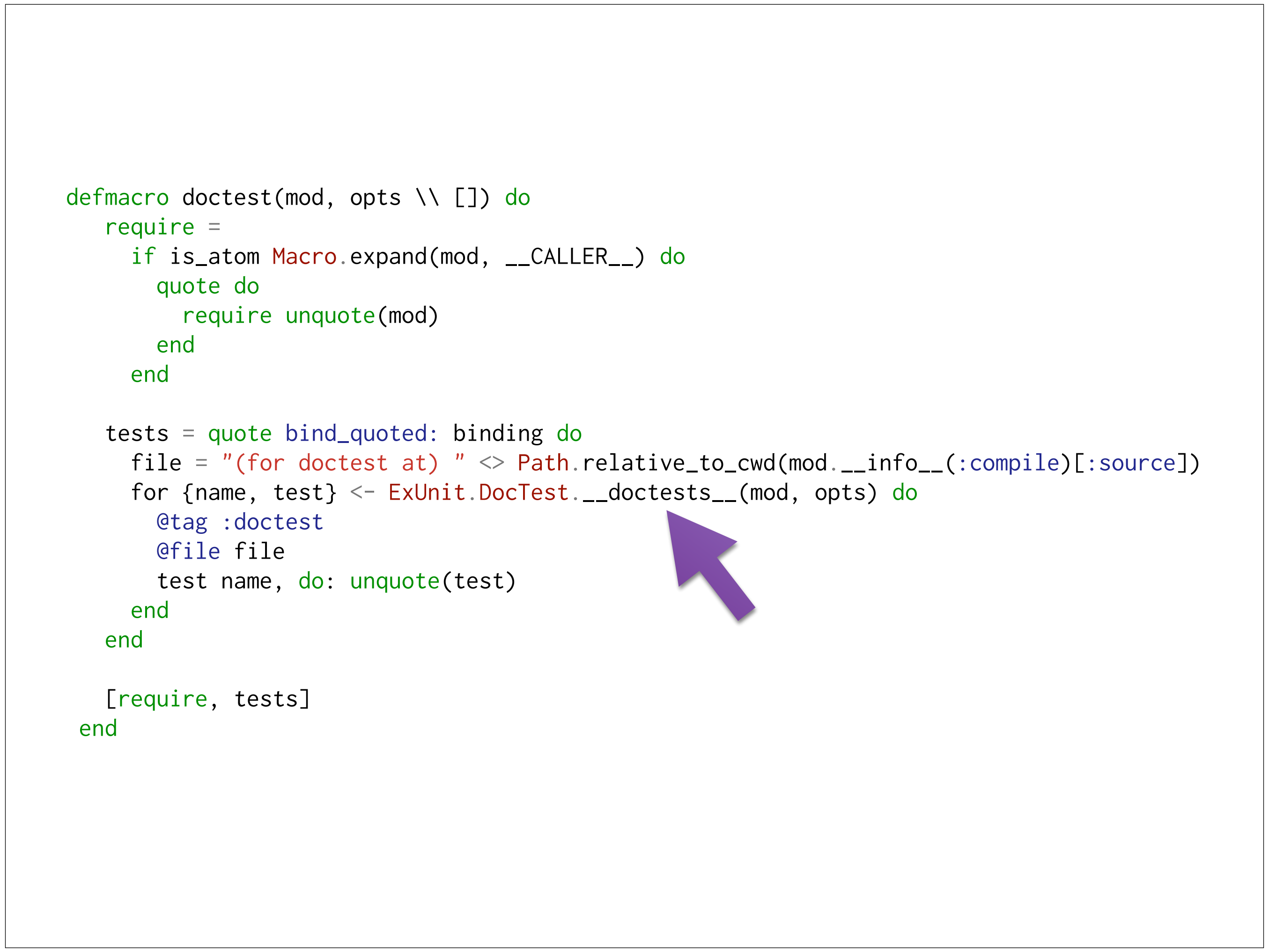
StringIO.close returns {:ok, {input_buffer, output_buffer}}



DocTests

```
defmodule Foo do
  @doc """
  This function does something

  iex> bar(1)
  4
  iex> bar(2)
  5
  """
  def bar(x), do: x + 3
end

defmodule FooTest do
  use ExUnit.Case, async: true
  doctest Foo
end
```

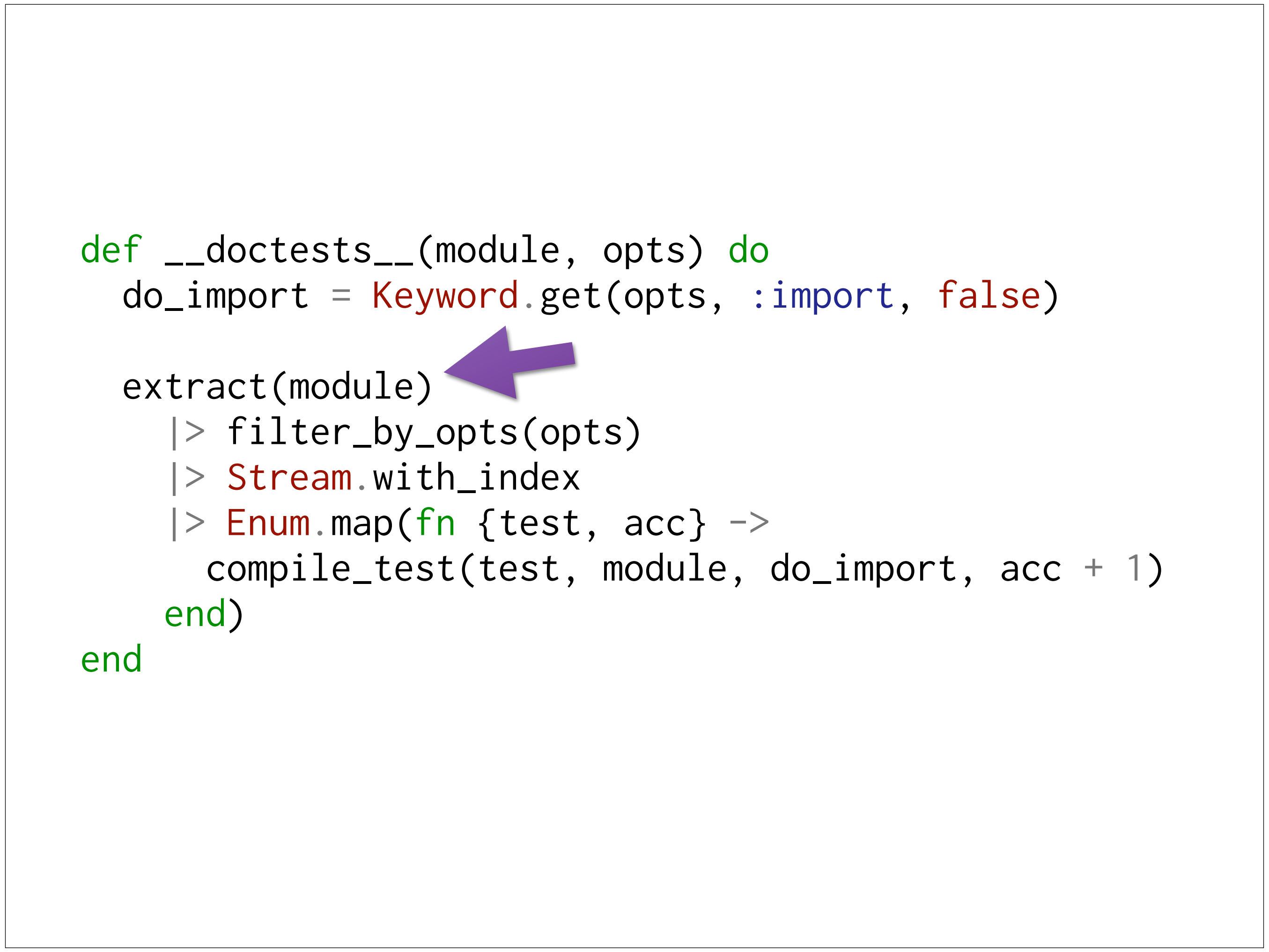
Calling the doctest macro in a Case will generate the tests based on the module and functions documentation.



```
defmacro doctest(mod, opts \\ []) do
   require =
     if is_atom Macro.expand(mod, __CALLER__) do
       quote do
         require unquote(mod)
       end
     end

   tests = quote bind_quoted: binding do
     file = "(for doctest at) " <> Path.relative_to_cwd(mod.__info__(:compile)[:source])
     for {name, test} <- ExUnit.DocTest.__doctests__(mod, opts) do
       @tag :doctest
       @file file
       test name, do: unquote(test)
     end
   end

   [require, tests]
 end
```

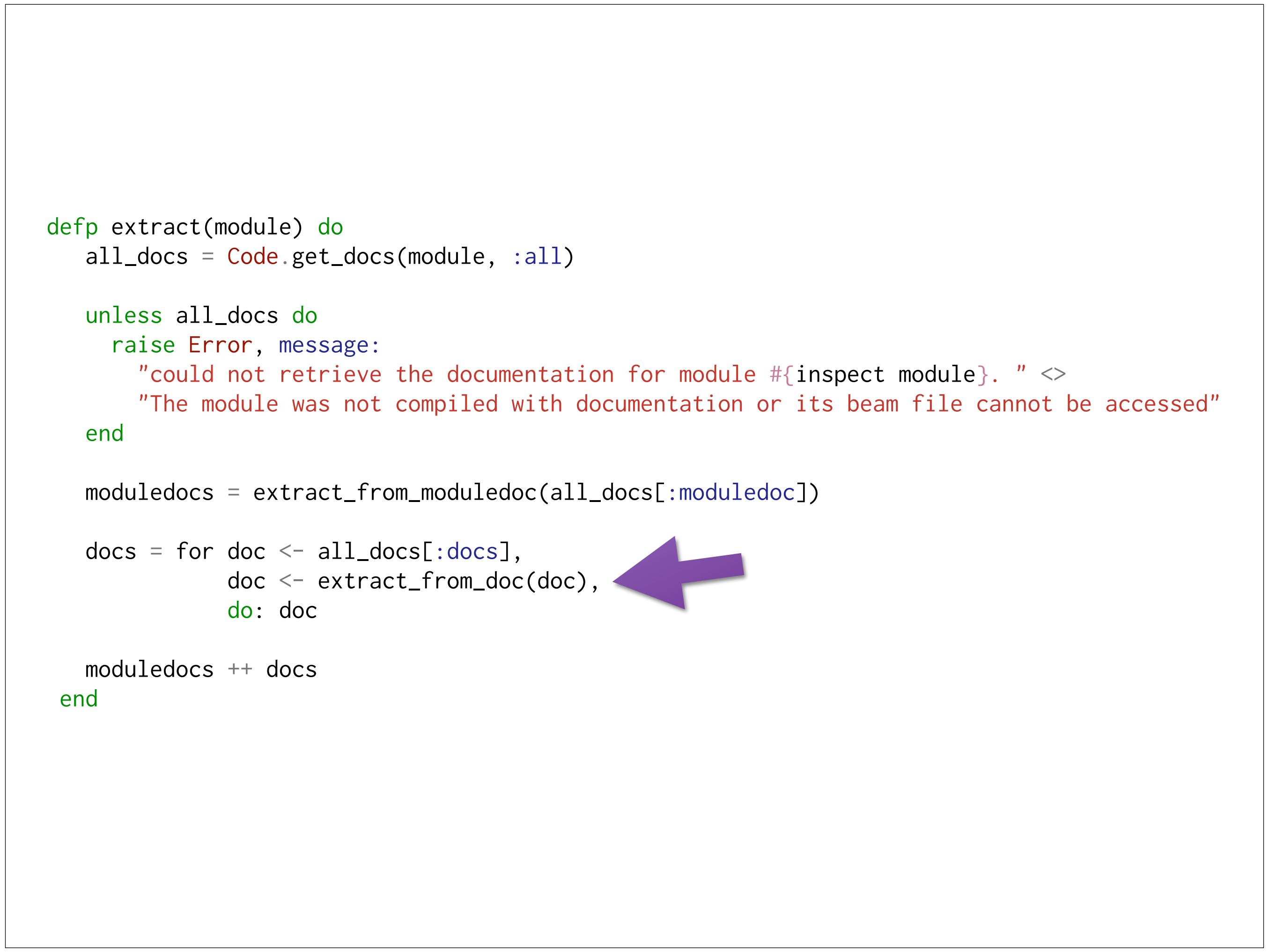
The output of the macro is an optional require clause (unless the module has already been expanded) and a list of generated calls to the test macro

Then we collect the tests via introspection (__info__ returns compilation metadata, including original source path) and eventually generate the test cases with a couple of additional annotations



```
def __doctests__(module, opts) do
  do_import = Keyword.get(opts, :import, false)

  extract(module)
    |> filter_by_opts(opts)
    |> Stream.with_index
    |> Enum.map(fn {test, acc} ->
      compile_test(test, module, do_import, acc + 1)
    end)
end
```

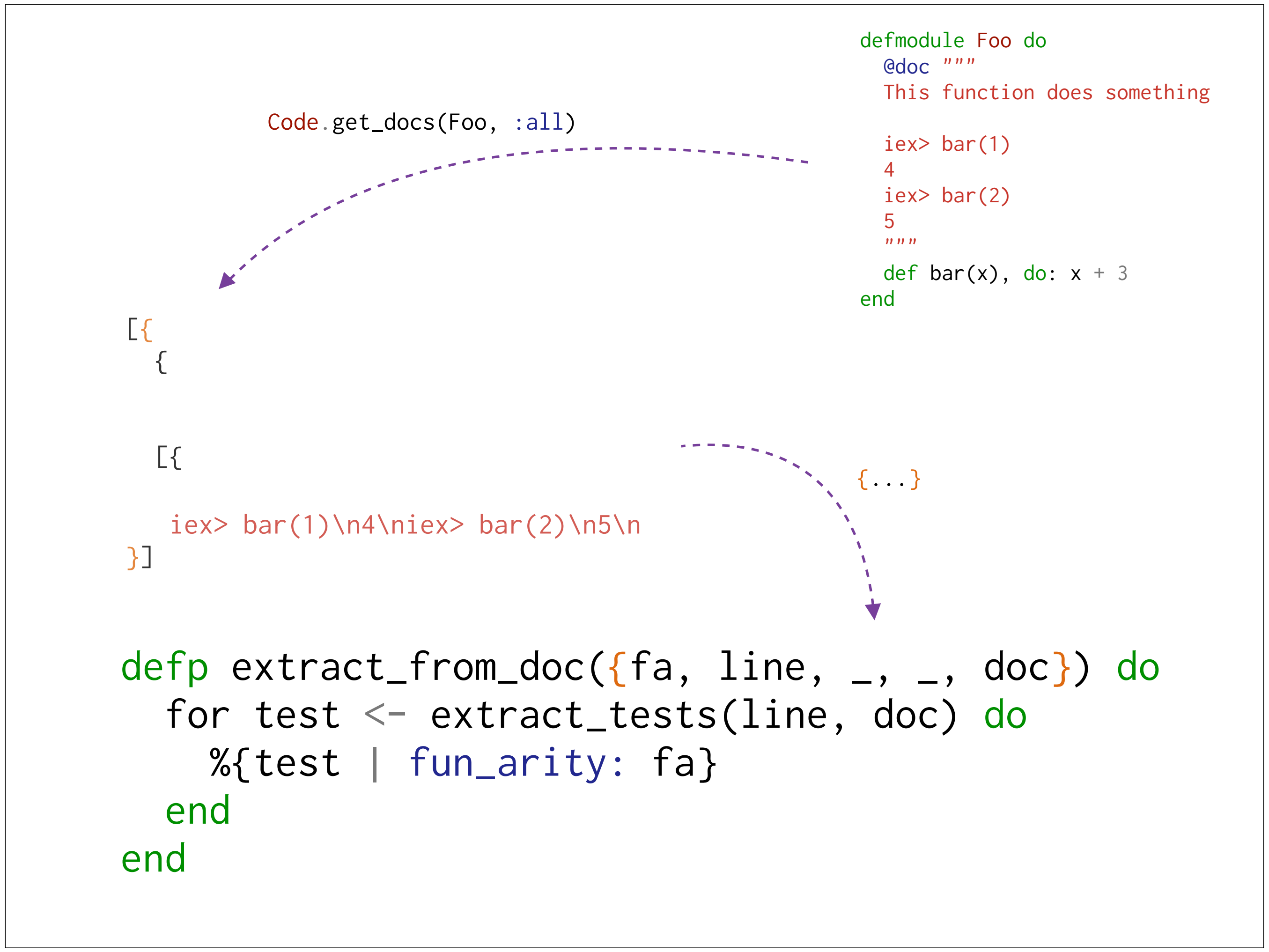
Returns a list of {name, quoted_test_code} pairs



```
defp extract(module) do
   all_docs = Code.get_docs(module, :all)

   unless all_docs do
     raise Error, message:
       "could not retrieve the documentation for module #{inspect module}. " <>
       "The module was not compiled with documentation or its beam file cannot be accessed"
   end

   moduledocs = extract_from_moduledoc(all_docs[:moduledoc])

   docs = for doc <- all_docs[:docs],
              doc <- extract_from_doc(doc),
              do: doc

   moduledocs ++ docs
 end
```

get_docs return the context (line, function signature, ...) and actual text of the @doc and @moduledoc attributes

extract_from_doc contains the actual parsing



```
defmodule Foo do
  @doc """
  This function does something

  iex> bar(1)
  4
  iex> bar(2)
  5
  """
  def bar(x), do: x + 3
end
```

```
Code.get_docs(Foo, :all)
```

```
[{
  {


  [{

   iex> bar(1)\n4\niex> bar(2)\n5\n
}]
```

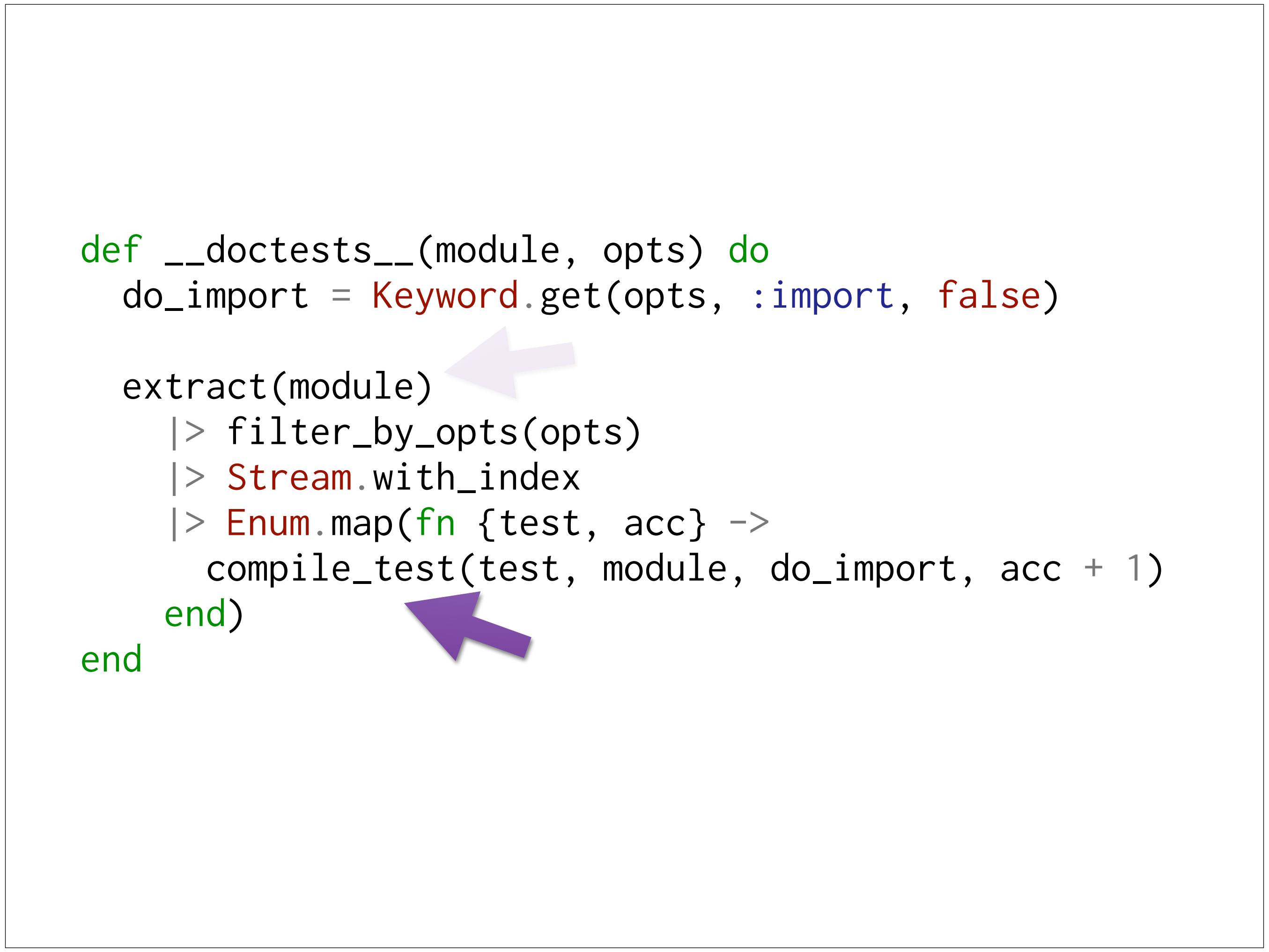
```
{...}
```

```
defp extract_from_doc({fa, line, _, _, doc}) do
  for test <- extract_tests(line, doc) do
    %{test | fun_arity: fa}
  end
end
```

Get the the arity and name of the function under test from the module metadata

`extract_test` **TL;DR**

- Split lines, clear indentation, scan for `iex>` prompts
- Extract expressions, extract expected results
- Expected results can be either:
  - a literal ①
  - the output of `inspect` (aggregates, opaque types) ②
  - an error (exception) ③



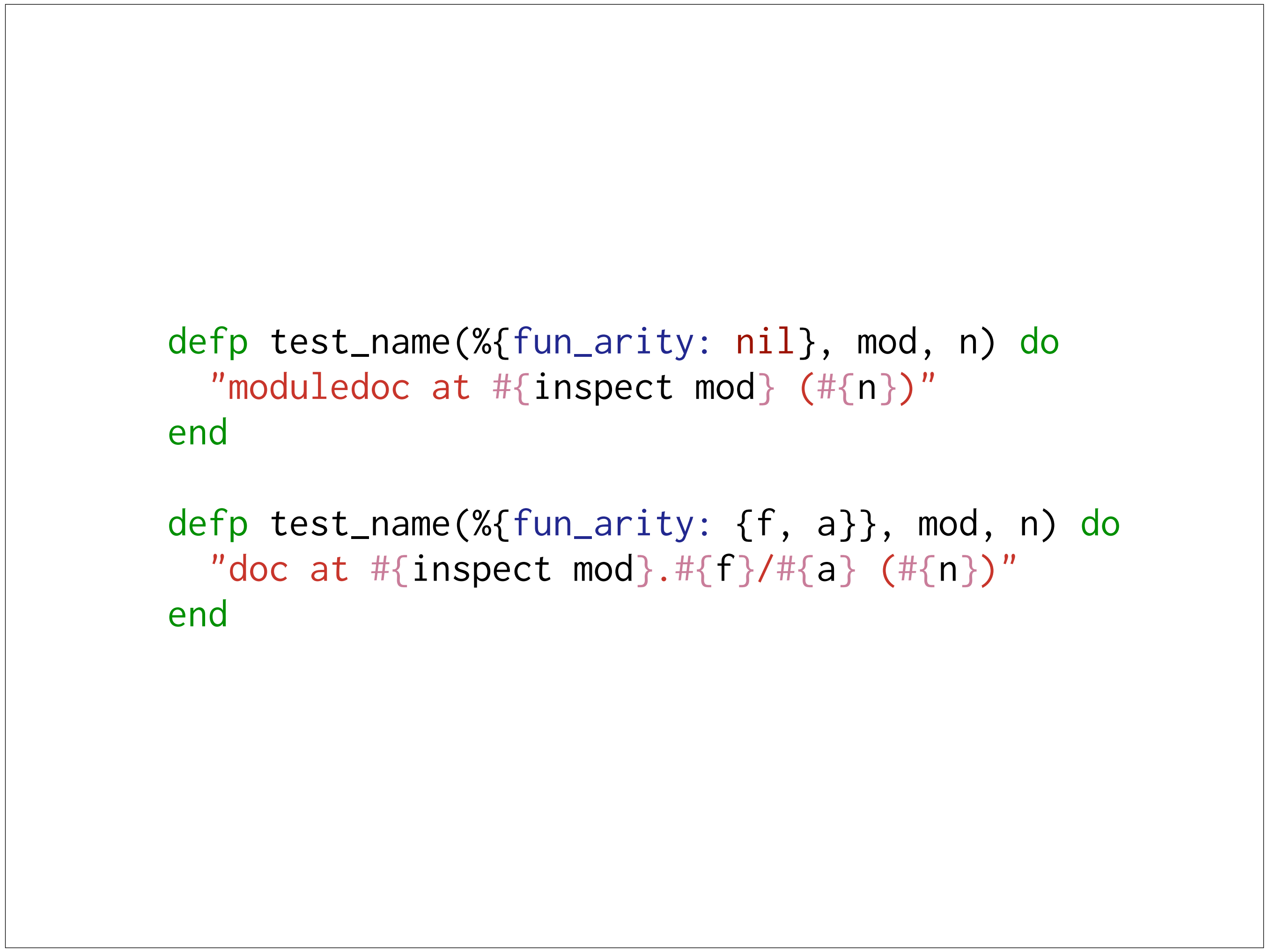
```
def __doctests__(module, opts) do
  do_import = Keyword.get(opts, :import, false)

  extract(module)
    |> filter_by_opts(opts)
    |> Stream.with_index
    |> Enum.map(fn {test, acc} ->
      compile_test(test, module, do_import, acc + 1)
    end)
end
```

Returns a list of {name, quoted_test_code} pairs

```
defp compile_test(test, module, do_import, n) do
  {test_name(test, module, n), test_content(test, module, do_import)}
end
```

Used in the `doctest` macro:

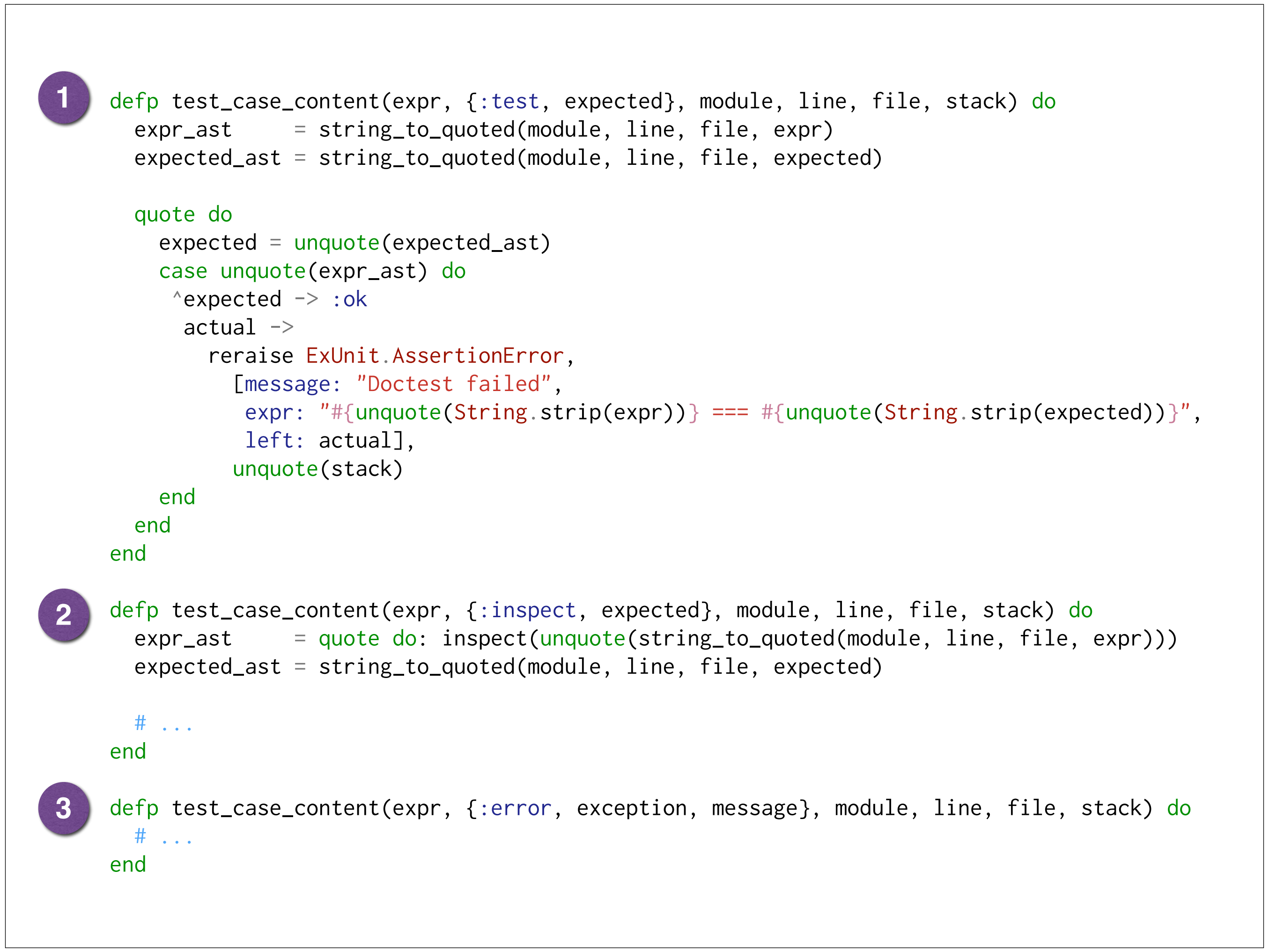
```
...
for {name, test} <- ExUnit.DocTest.__doctests__(mod, opts) do
  @tag :doctest
  @file file
  test name, do: unquote(test)
end
...
```



```
defp test_name(%{fun_arity: nil}, mod, n) do
  "moduledoc at #{inspect mod} (#{n})"
end

defp test_name(%{fun_arity: {f, a}}, mod, n) do
  "doc at #{inspect mod}.#{f}/#{a} (#{n})"
end
```

Generation of the doctest name for a module or function

n is the the # of the test since there can be more than one per docstring



```
defp test_case_content(expr, {:test, expected}, module, line, file, stack) do
  expr_ast     = string_to_quoted(module, line, file, expr)
  expected_ast = string_to_quoted(module, line, file, expected)

  quote do
    expected = unquote(expected_ast)
    case unquote(expr_ast) do
     ^expected -> :ok
      actual ->
        reraise ExUnit.AssertionError,
          [message: "Doctest failed",
           expr: "#{unquote(String.strip(expr))} === #{unquote(String.strip(expected))}",
           left: actual],
          unquote(stack)
    end
  end
end

defp test_case_content(expr, {:inspect, expected}, module, line, file, stack) do
  expr_ast     = quote do: inspect(unquote(string_to_quoted(module, line, file, expr)))
  expected_ast = string_to_quoted(module, line, file, expected)

  # ...
end

defp test_case_content(expr, {:error, exception, message}, module, line, file, stack) do
  # ...
end
```

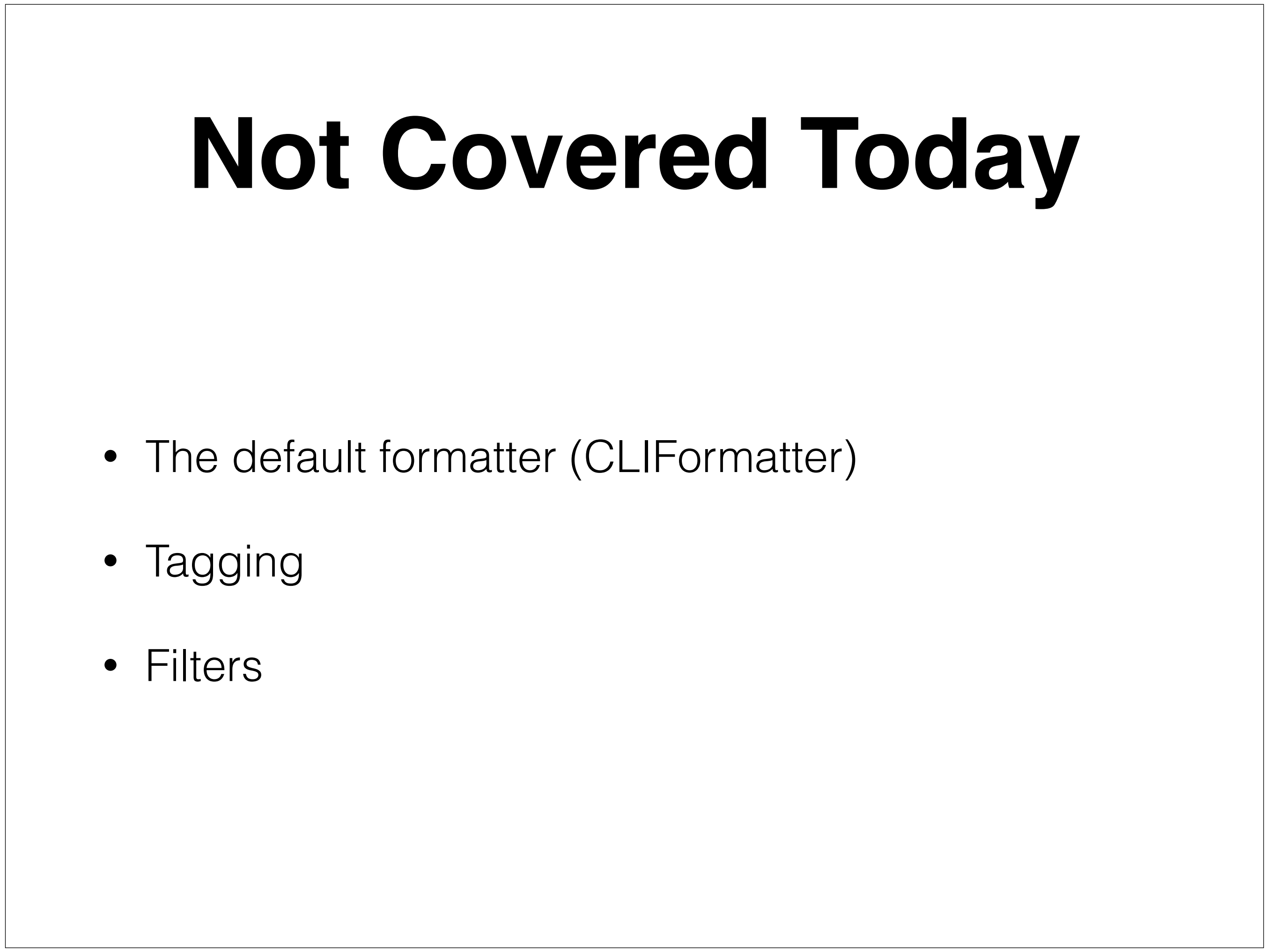
Generation of the test case for an expected inspect value

test_case_content is not a macro but it does return quoted expressions to be used by a macro later on.

string_to_quoted delegates to the Code module but adds better error reporting

expr_ast contains the actual test where the given example is wrapped into an inspect

The code for test_case_content for exceptions is similar enough and left as an exercise



Not Covered Today

- The default formatter (CLIFormatter)
- Tagging
- Filters

Filters allow to include or exclude tests using based on tags

Takeaways

- Macros are mindblowing (both literally and figuratively)
- Interplay between the language and the compiler
- Leverage module attributes and callback mechanisms to build self-descriptive components

Further Readings

- ExUnit API documentation
  http://elixir-lang.org/docs/stable/ex_unit/

- Elixir Getting Started: Metaprogramming
  http://elixir-lang.org/getting_started/meta/1.html (Quote & Unquote)
  http://elixir-lang.org/getting_started/meta/2.html (Macros)
  http://elixir-lang.org/getting_started/meta/3.html (Domain Specific Languages)

- Saša Jurić's "Understanding Elixir Macros" epic series
  http://www.theerlangelist.com/2014/06/understanding-elixir-macros-part-1.html

- Linked Processes, Errors and Monitors
  http://learnyousomeerlang.com/errors-and-processes

- Elixir Standard Library Documentation
  http://elixir-lang.org/docs/stable/elixir/Code.html (Compiler integration)
  http://elixir-lang.org/docs/stable/elixir/Kernel.SpecialForms.html (Details about `quote/2` and the AST)

Thanks!

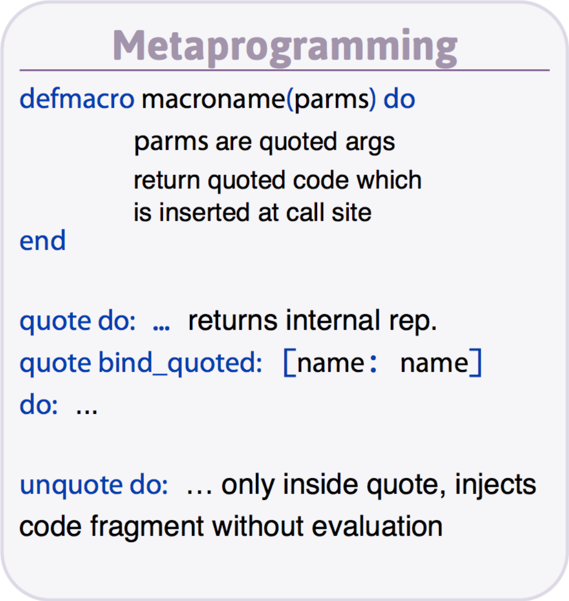
Questions?

@xavierdefrang
github.com/xavier

Exercises

- Write a bunch of macros
  - unless
  - assert
  - generate functions (see Etudes for Elixir Chapter 13)
  - ...
- Write your own ExUnit custom formatter
  (look at the CLIFormatter module in ExUnit source)
- Check out Dave Thomas' screencast "A Simple Elixir Macro"

Cheat Sheet



Metaprogramming

```
defmacro macroname(parms) do
            parms are quoted args
            return quoted code which
            is inserted at call site
end

quote do: … returns internal rep.
quote bind_quoted: [name: name]
do: ...

unquote do: … only inside quote, injects
code fragment without evaluation
```

http://media.pragprog.com/titles/elixir/ElixirCheat.pdf