Elixir Tooling

Exploring Beyond the Language

@emjii

What’s happening?

- 1.0
- ElixirConf
- Hex

Topics

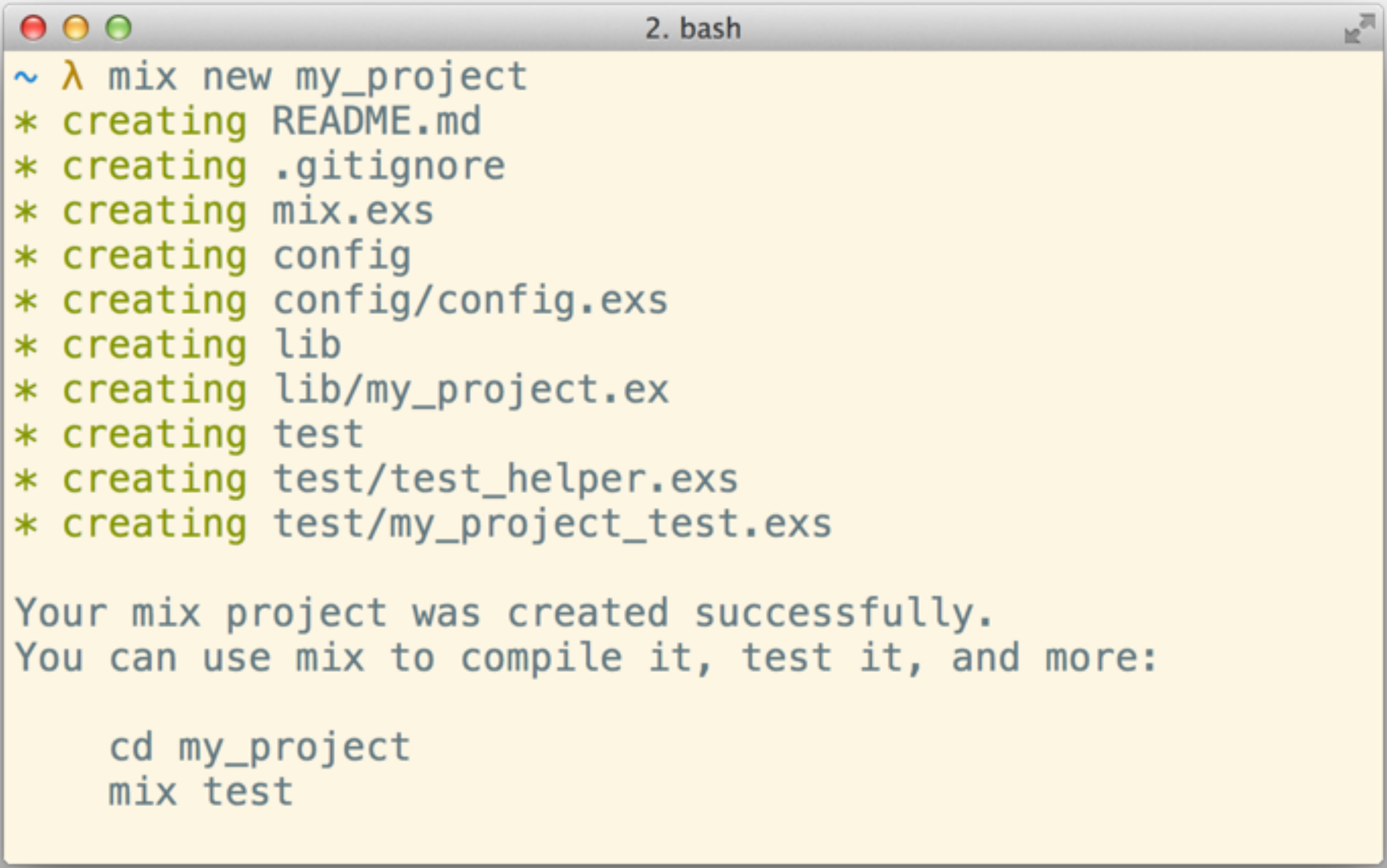
- Mix
- Hex
- IEx
- ExUnit
- Standard Library

Mix

Mix

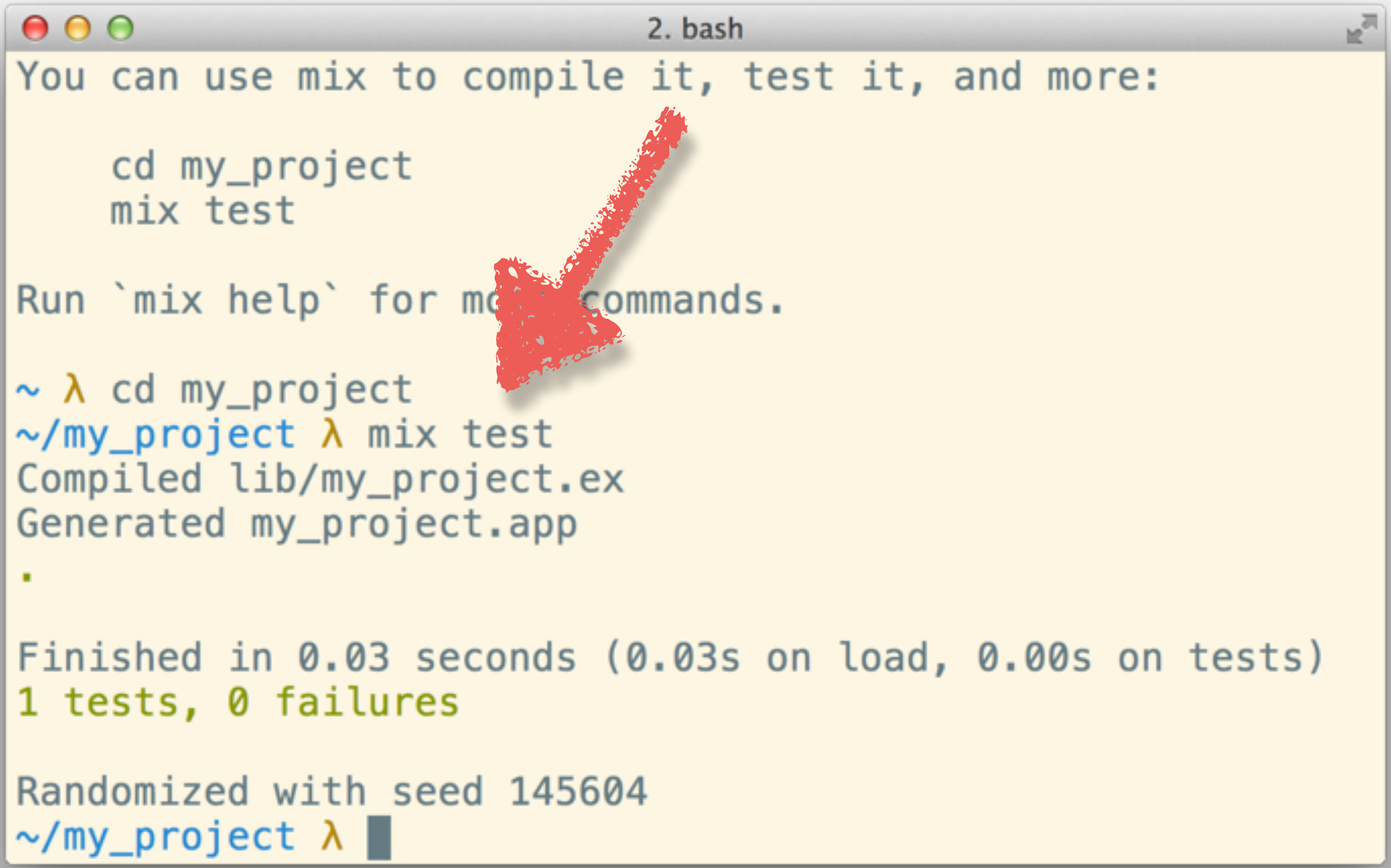
- Generate new projects
- Compile
- Run tests
- Handle dependencies
- Whatever else you can think of

mix new



```
~ λ mix new my_project
* creating README.md
* creating .gitignore
* creating mix.exs
* creating config
* creating config/config.exs
* creating lib
* creating lib/my_project.ex
* creating test
* creating test/test_helper.exs
* creating test/my_project_test.exs

Your mix project was created successfully.
You can use mix to compile it, test it, and more:

    cd my_project
    mix test
```

mix new



```
You can use mix to compile it, test it, and more:

    cd my_project
    mix test

Run `mix help` for more commands.

~ λ cd my_project
~/my_project λ mix test
Compiled lib/my_project.ex
Generated my_project.app
.

Finished in 0.03 seconds (0.03s on load, 0.00s on tests)
1 tests, 0 failures

Randomized with seed 145604
~/my_project λ
```

mix.exs

```
defmodule MyProject.Mixfile do
  use Mix.Project
  def project do
    [ app: :my_project,
      version: "0.1.0",
      elixir: "~> 0.14.0" ]
  end
end
```

Compilation

- Generates .beams and .app
- Compiles erlang code
- And also .leex & .yecc files

Dependencies

```
defp deps do
  [ { :poolboy, github: "devinus/poolboy" },
    { :ecto, "~> 0.2.0" } ]
end
```

Dependencies

- $ mix deps
- Converger
- Repeatable builds
- Rebar dependencies

Extending Mix

```
defmodule Mix.Tasks.MyTask do
  use Mix.Task

  def run(args) do
    IO.puts "Hello world!"
  end
end

$ mix my_task
Hello world!
```

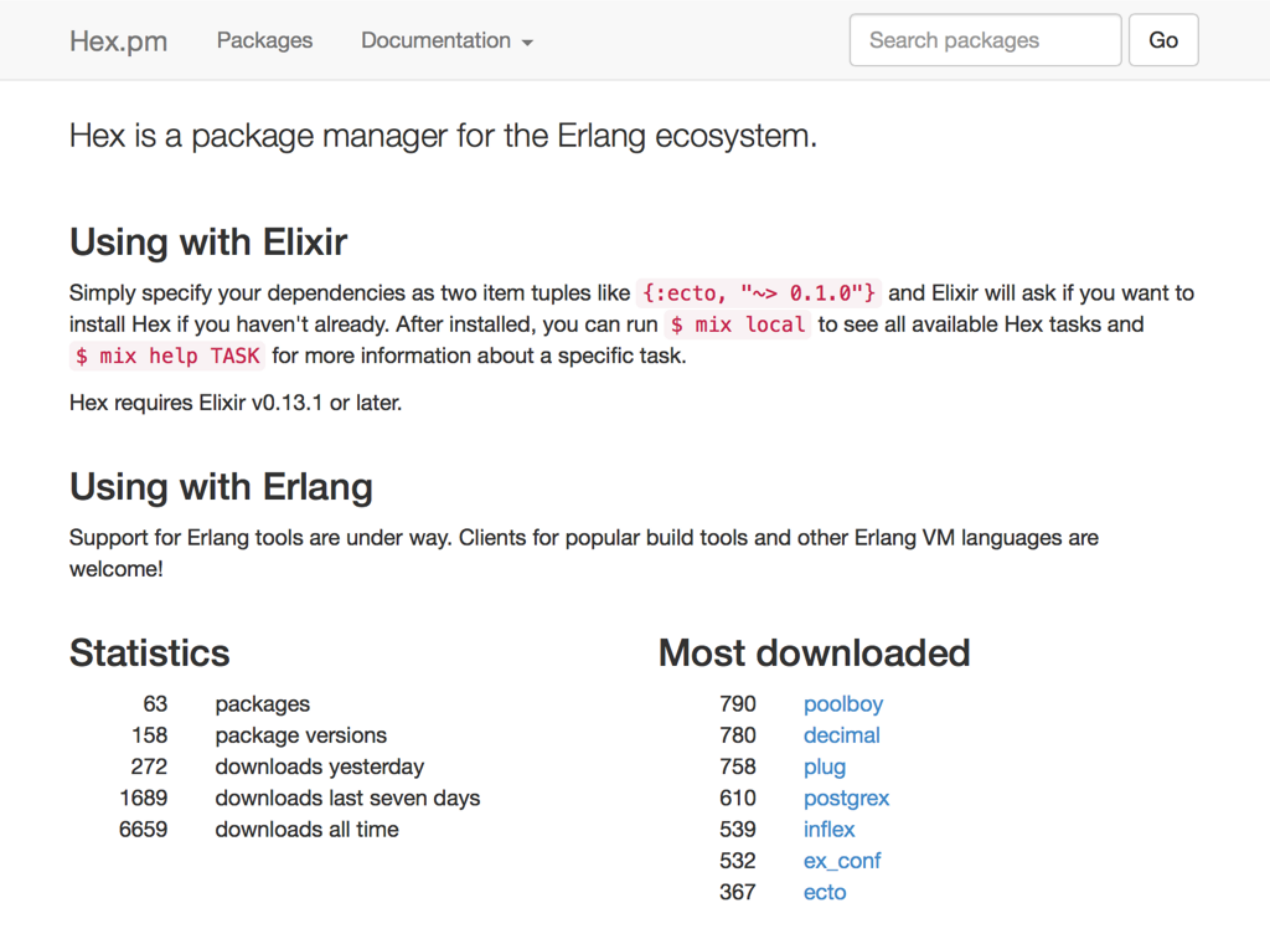
Extending Mix

- ecto (github.com/elixir-lang/ecto)
  - $ mix ecto.gen.migration
  - $ mix ecto.migrate
- exrm (github.com/bitwalker/exrm)
  - $ mix release

Umbrella projects

- apps/*
- Isolated applications
- Recursive tasks

Hex



Hex is a package manager for the Erlang ecosystem.

Using with Elixir

Simply specify your dependencies as two item tuples like `{:ecto, "~> 0.1.0"}` and Elixir will ask if you want to install Hex if you haven't already. After installed, you can run `$ mix local` to see all available Hex tasks and `$ mix help TASK` for more information about a specific task.

Hex requires Elixir v0.13.1 or later.

Using with Erlang

Support for Erlang tools are under way. Clients for popular build tools and other Erlang VM languages are welcome!

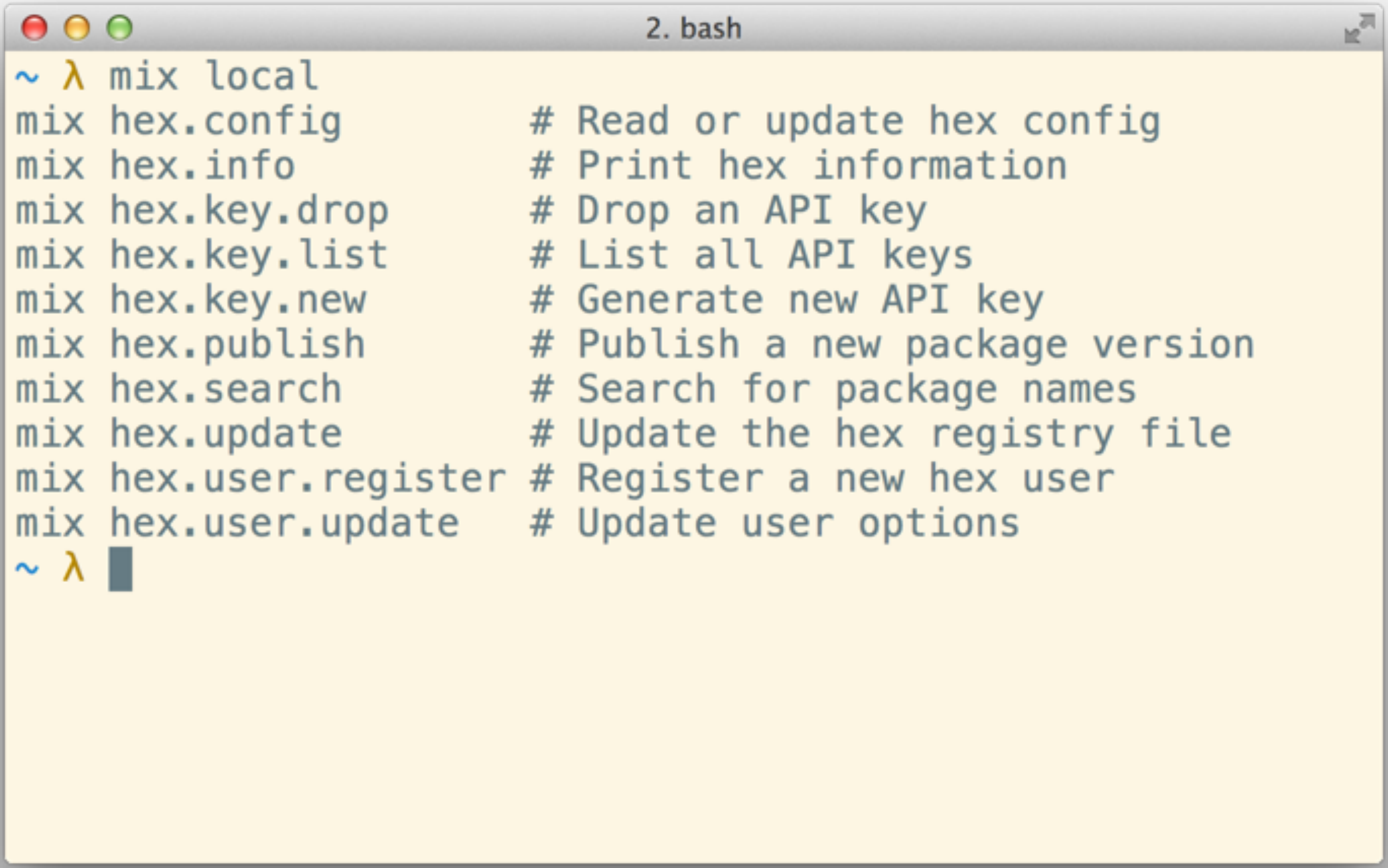
Statistics

63	packages
158	package versions
272	downloads yesterday
1689	downloads last seven days
6659	downloads all time

Most downloaded

790	poolboy
780	decimal
758	plug
610	postgrex
539	inflex
532	ex_conf
367	ecto

Hex tasks



```
~ λ mix local
mix hex.config          # Read or update hex config
mix hex.info            # Print hex information
mix hex.key.drop        # Drop an API key
mix hex.key.list        # List all API keys
mix hex.key.new         # Generate new API key
mix hex.publish         # Publish a new package version
mix hex.search          # Search for package names
mix hex.update          # Update the hex registry file
mix hex.user.register # Register a new hex user
mix hex.user.update     # Update user options
~ λ
```

Dependency resolution

- Find the latest version that satisfies all requirements
- Use the lockfile
- Honour overrides

Future work

- Erlang support
- Installing executables

IEx

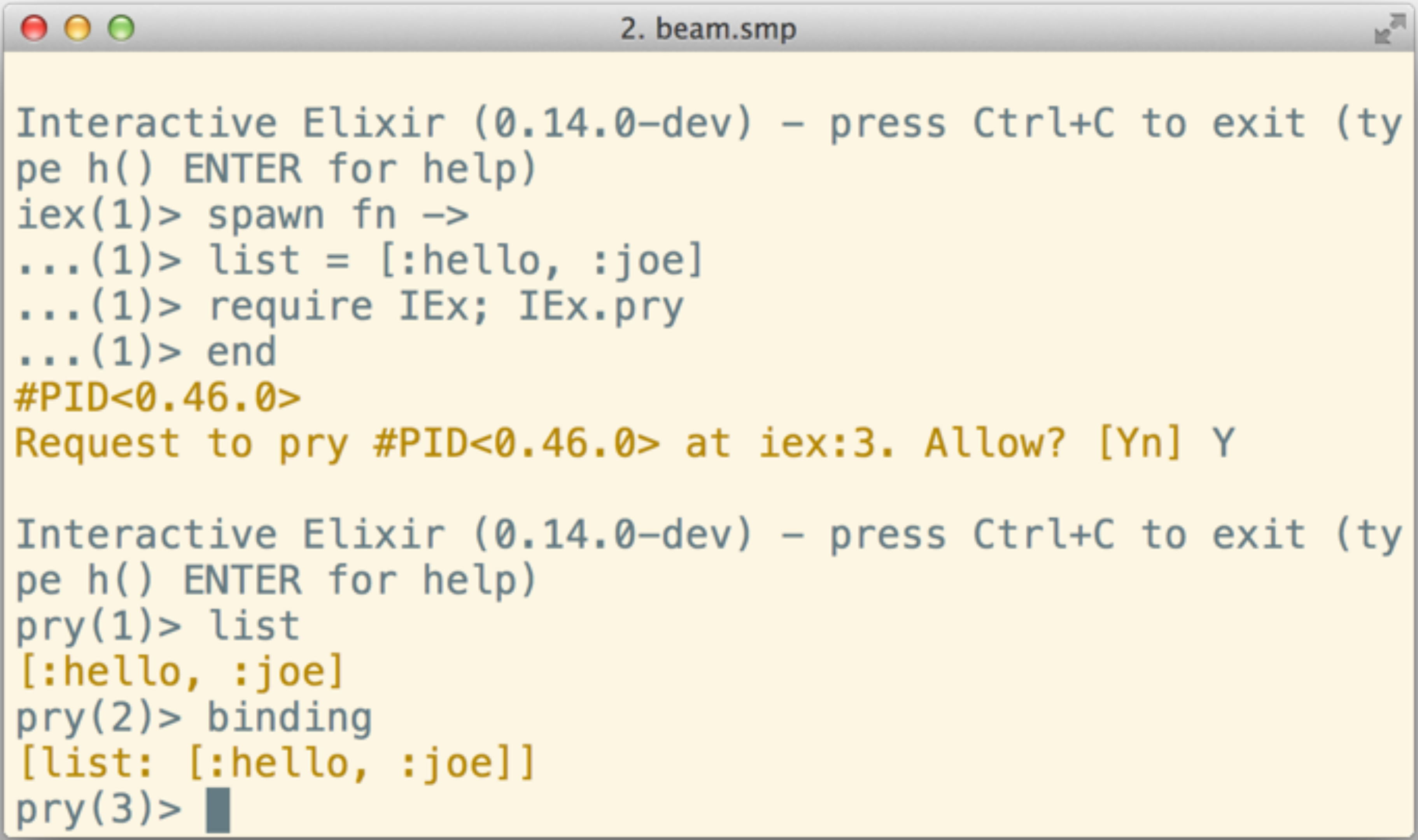
iex -S mix

- $ mix run
- Loads configs
- Loads and starts dependencies

pry

- Hook into a running process
- Inspired by ruby's pry

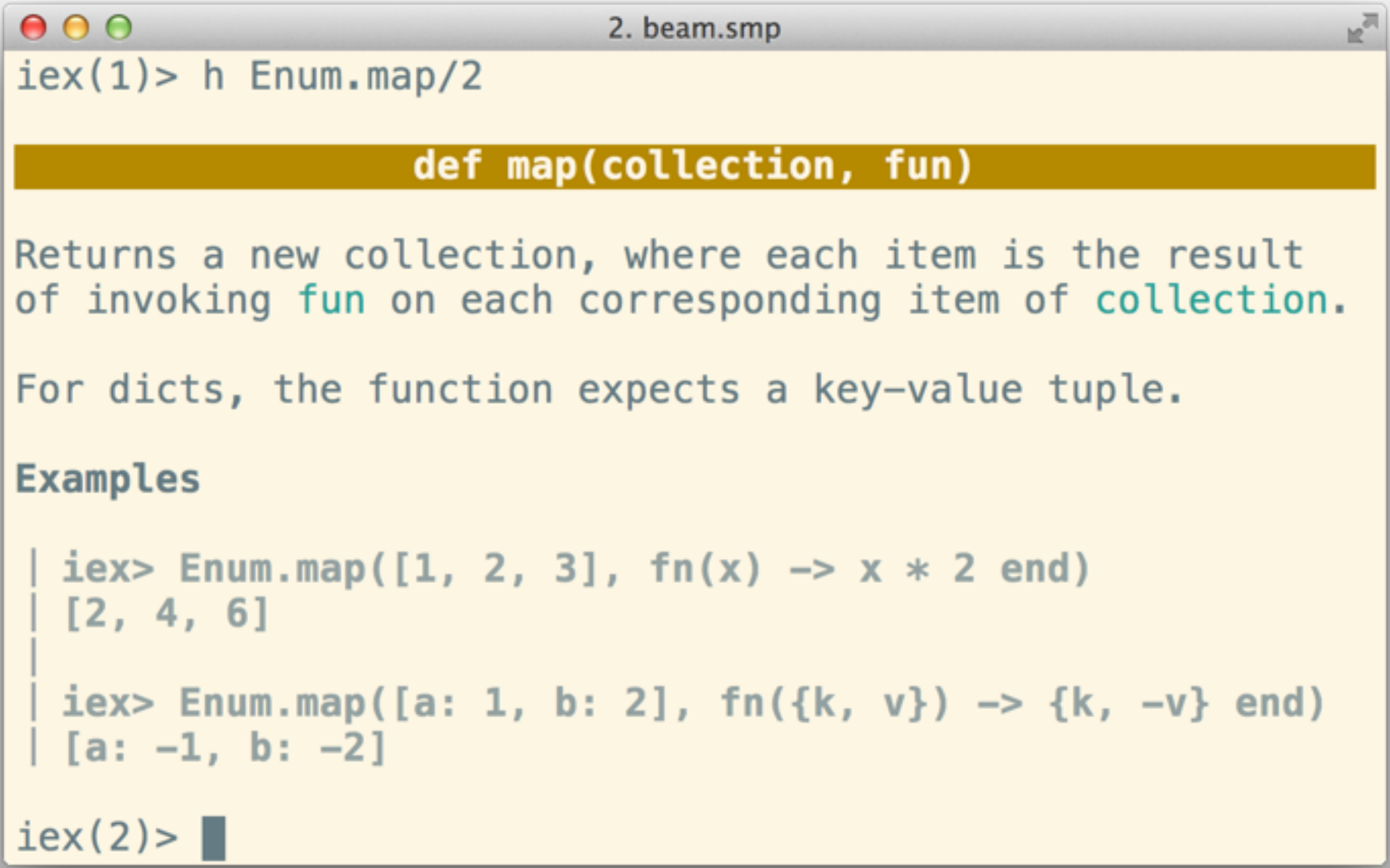
pry



2. beam.smp

```
Interactive Elixir (0.14.0-dev) - press Ctrl+C to exit (ty
pe h() ENTER for help)
iex(1)> spawn fn ->
...(1)> list = [:hello, :joe]
...(1)> require IEx; IEx.pry
...(1)> end
#PID<0.46.0>
Request to pry #PID<0.46.0> at iex:3. Allow? [Yn] Y

Interactive Elixir (0.14.0-dev) - press Ctrl+C to exit (ty
pe h() ENTER for help)
pry(1)> list
[:hello, :joe]
pry(2)> binding
[list: [:hello, :joe]]
pry(3)>
```

First class docs



```
iex(1)> h Enum.map/2

                 def map(collection, fun)

Returns a new collection, where each item is the result
of invoking fun on each corresponding item of collection.

For dicts, the function expects a key-value tuple.

Examples

 | iex> Enum.map([1, 2, 3], fn(x) -> x * 2 end)
 | [2, 4, 6]
 |
 | iex> Enum.map([a: 1, b: 2], fn({k, v}) -> {k, -v} end)
 | [a: -1, b: -2]

iex(2)> 
```

ExUnit

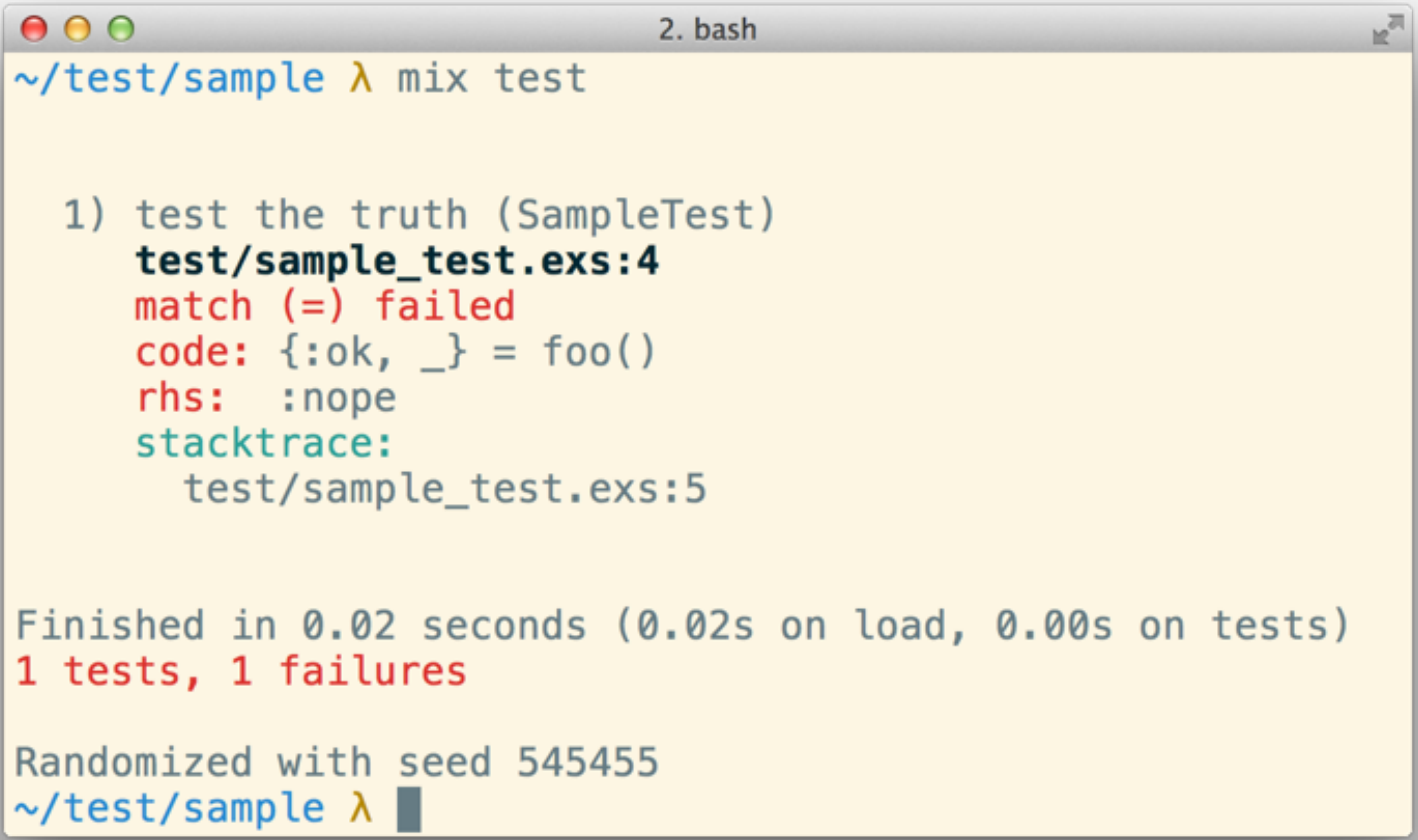
The assert macro

```elixir
defmodule SampleTest do
  use ExUnit.Case

  test "the truth" do
    assert {:ok, _} = foo()
  end

  defp foo do
    :nope
  end
end
```

The assert macro

```
quote do assert {:ok, _} = foo() end

{:assert, [], [
  {:=, [], [
    {:ok, {:_, [], nil}},
    {:foo, [], []}
  ]}
]}
```

Beautiful failures



```
~/test/sample λ mix test


  1) test the truth (SampleTest)
     test/sample_test.exs:4
     match (=) failed
     code: {:ok, _} = foo()
     rhs:  :nope
     stacktrace:
       test/sample_test.exs:5



Finished in 0.02 seconds (0.02s on load, 0.00s on tests)
1 tests, 1 failures

Randomized with seed 545455
~/test/sample λ
```

Tags & Filters

```elixir
defmodule SampleTest do
  use ExUnit.Case

  @tag :integration
  test "the truth" do
    # call some expensive service
  end
end
```

Tags & Filters

- —only / —include / —exclude
- $ mix test test/sample_test.exs:5

Doctests

```
@doc """
Returns a new collection, where each item is the result
of invoking `fun` on each corresponding item of `collection`.

For dicts, the function expects a key-value tuple.

## Examples

    iex> Enum.map([1, 2, 3], fn(x) -> x * 2 end)
    [2, 4, 6]

    iex> Enum.map([a: 1, b: 2], fn({k, v}) -> {k, -v} end)
    [a: -1, b: -2]

"""
@spec map(t, (element -> any)) :: list
def map(collection, fun) do
```

Standard library

Stream

- Composable, lazy collections
- Implements Enumerable protocol

Read file by line

```
def read(filename) do
  read_device(File.open!(filename))
end

def read_device(device) do
  case IO.read(device) do
    :eof -> :ok
    line ->
      operation(line)
      read_device(device)
  end
end
```

Streaming IO

```
File.stream!(filename)
|> Enum.each(&operation/1)
```

Streaming GenEvent

```elixir
stream = GenEvent.stream(pid)

# Take the next 10 events
Enum.take(stream, 10)

# Print all remaining events
for event <- stream do
  IO.inspect event
end
```

Extending OTP

- OTP's great
- No high-level abstractions
- What exists in other languages?

Agent

- Abstraction around state
- Inspired by Clojure
- Builds on GenServer

Agent

```elixir
defmodule Cache do
  def start_link do
    Agent.start_link(fn -> HashDict.new end)
  end

  def put(pid, key, value) do
    Agent.update(pid, &Dict.put(&1, key, value))
  end

  def get(pid, key) do
    Agent.get(pid, &Dict.get(&1, key))
  end
end
```

Agent

```
-module(sample).
-export([start_link/0, put/3, get/2]).

-define(Agent, 'Elixir.Agent').
-define(HashDict, 'Elixir.HashDict').

start_link() ->
    ?Agent:start_link(fun() -> ?HashDict:new() end).

put(Pid, Key, Value) ->
  ?Agent:update(Pid, fun(Dict) ->
    ?HashDict:put(Dict, Key, Value)
  end).

get(Pid, Key) ->
    ?Agent:get(Pid, fun(Dict) -> ?HashDict:get(Dict, Key) end).
```

Task

- Asynchronous tasks
- Small, single action

Task.async & Task.await

```
task = Task.async(&do_some_work/1)
res  = do_some_other_work()
res + Task.await(task)
```

?

@emjii