Inspiration and some code taken from these books. (They’re great!)
Overview

1. Firehose-style introduction to Elixir
2. Actors (or “processes”)
3. Erlang/OTP - Elixir/OTP
   a. Supervisor
   b. GenServer
   c. Application
4. Elixir Macros
5. Slackbot
Elixir?

- Ruby syntax, compiled to Core Erlang to run on BEAM
- Functional (side-effects, no mutability)
- Modern toolset to develop Erlang programs
- All the Erlang stuff
  - Fault Tolerance
  - Scalability
  - Distribution
  - Responsiveness
  - Live code reloading
Tools

- The REPL
  *iex*
- The build-tool
  *mix*
- The package repository
  *hex* ([https://hex.pm/](https://hex.pm/))
- Editors
  *Emacs/IntelliJ/Atom/Sublime/Visual Studio Code/Vim*
https://github.com/m1dnight/hello_world

https://github.com/m1dnight/slackbot

Basic Elixir Syntax

- **Variables**
  > x = 5

- **Lambdas**
  > f = fn(x,y,z) -> x + y + z end
  > f.(1,2,3)

- **Function calls**
  > IO.puts("Hello, World")
  > IO.puts "Hello,World"

- **Modules**
  defmodule MyModule do
    def my_function() do
      5
    end
  end

- **Data types**
  - Atoms: :this_is_an_atom
  - Strings: “This is a string”
    ~s(“quoted string”)
  - Lists: [1,2,:foo]
  - Bools: true false :true :false
  - Null: nil :nil
  - Tuples: {:foo, “bar”, 1}
  - Maps: m = %{“key” => :value}
    m.foo or m[“foo”]
  - Bitstrings: <<1, 2, 3>>
    <<257::16>> 16 bits for that value
    == <<1, 1>>
Pattern Matching

• Basics
  > x = 1  # Check if 1 matches with pattern x
  > 1 = 1  # Check if 1 matches with 1
  > _ = 4  # Underscore wildcard

• Datastructures
  > %{:key => val} = %{:key => 5, :key => 6}  # Bind val to 5
  > [x,y,z | xs] = [1,2,3,4]
  > x = 5
  > {^x, y} = {5, 6}  # Unifies y with 6
  > {^x, y} = {6,7}  # Error

http://elixir-lang.org/getting-started/pattern-matching.html
def divide(_x, 0), do: {error, "division by zero"}

def divide(x, 1), do: {ok, x}

def divide(x,y) where is_integer(x) and is_integer(y) do:
  {ok, x / y}
end

def divide(x,y) do:
  {error, "can only divide integers"}
end
defmodule Greeting do
  @message "Hello 
  def greet(greetee) do
    IO.puts @message <> greetee
  end

  def insult(_insultee) do
    exit(:i_refuse)
  end
end
Basic Actor Operations

- Creating an actor with `spawn/1`
  
  ```elixir
  actor_id = spawn(fn() -> IO.puts "I’m another actor" end)
  ```

- Creating an actor with `spawn/3`
  
  ```elixir
  actor_id = spawn(IO, :puts, ["I’m another actor"])
  ```

- Sending a message to another actor with `send/2`
  
  ```elixir
  send(actor_id, {:this, "is", "a", "message"})
  ```
  
  - Asynchronous
  - Delivery is in order, guaranteed delivery to inbox (or you will find out otherwise if you use links)

- Receiving a message with `send`
  
  ```elixir
  receive do
    pattern -> exp
  end
  ```

http://elixir-lang.org/getting-started/processes.html
http://erlang.org/faq/academic.html
Inbox priorities

receive do
  {:second, _val} -> :first
after 0 ->
  receive do
    {:first, _val} -> :second
  end
end

> send(self(), {:first, 1})
> send(self(), {:second, 2})

http://elixir-lang.org/getting-started/processes.html
defmodule Greetingv2 do
  def loop() do
    receive do
      {:greet, greetee} ->
        IO.puts "Hello " <> greetee
        loop()
      {:insult, _insultee} ->
        exit(:i_refuse)
    end
  end
end

iex(1)> pid = spawn(Greetingv2, :loop, [])
#PID<0.108.0>
iex(2)> send(pid, {:greet, "XAOP"})
Hello XAOP
{:greet, "XAOP"}
iex(3)> send(pid, {:insult, "XAOP"})
{:insult, "XAOP"}
iex(4)> send(pid, {:greet, "XAOP"})
{:greet, "XAOP"}
iex(5)>
When a process dies, it emits a **special** message to all linked processes.

http://elixir-lang.org/getting-started/processes.html
Links and Monitors

When a process dies, it emits a special message to all linked processes.

```
> monitor(spawn(..))
```

[Diagram showing a process dying and emitting a message to its monitors.]

http://elixir-lang.org/getting-started/processes.html
defmodule Greetingv2 do
  def loop() do
    receive do
      {:greet, greetee} ->
        IO.puts "Hello " <> greetee
        loop()
      {:insult, _insultee} ->
        exit(:i_refuse)
    end
  end
end

iex(1)> pid = spawn_link(Greetingv2, :loop, [])
#PID<0.108.0>
iex(2)> send(pid, {:greet, "XAOP"})
Hello XAOP
{:greet, "XAOP"}
iex(9)> send(pid, {:insult, "XAOP"})
** (EXIT from #PID<0.106.0>):i_refuse

Interactive Elixir (1.4.2) - press Ctrl+C to exit
(type h() ENTER for help)
iex(1)>

Hello, World v2
Hello, World v2

defmodule Greetingv2 do
  def loop() do
    receive do
      {:greet, greetee} ->
        IO.puts "Hello " <> greetee
        loop()
      {:insult, _insultee} ->
        exit(:i_refuse)
    end
  end
end

iex(6)> pid = spawn_link(Greetingv2, :loop, [])
#PID<0.126.0>
iex(7)> Process.flag(:trap_exit, true)
false
iex(8)> send(pid, {:insult, "Elixir"})
{:insult, "Elixir"}
iex(9)> flush()
{:EXIT, #PID<0.126.0>, :i_refuse}
:ok
Elixir/OTP
Erlang/OTP?

- Erlang Open Telecom Platform
  - I.e., years worth of engineering from Ericsson!

- Abstractions over common behaviours
  - Supervisors: Make sure processes keep running
  - GenServer: Generic server interface
  - Custom behaviour interfaces
  - Application
  - ..

- Shapes your train of thought and implementation to obtain reusable parts
GenServer

- GenServer abstracts over the typical client-server architecture by providing a set of interface functions:
  - start
  - start_link
  - terminate/2
  - call/3
  - cast/2
  - info/2
  - terminate/2
  - code_change/3

- The interface functions are mostly optional. Logic can reside in a separate module or inside the GenServer.

https://hexdocs.pm/elixir/GenServer.html
defmodule Greeting.Server do
  use GenServer
  def start_link(args \ []]) do
    GenServer.start_link(__MODULE__, args, name: __MODULE__)
  end
  def init([]) do
    {:ok, []}
  end
  def handle_call({:greet, greetee}, _from, state) do
    IO.puts "Hello, " <> greetee
    {:reply, :ok, state}
  end
  def handle_call({:insult, insultee}, _from, state) do
    exit(:i_refuse)
  end
end
Hello, World - Server

```elixir
iex(2)> {:ok, pid} = GenServer.start_link(Greeting.Server, [])
{:ok, #PID<0.128.0>}
iex(3)> GenServer.call(pid, {:greet, "XAOP"})
Hello, XAOP
:ok
iex(4)> GenServer.call(pid, {:insult, "XAOP"})
** (EXIT from #PID<0.124.0>) :i_refuse
22:21:31.990 [error] GenServer #PID<0.128.0> terminating
** (stop) :i_refuse
... Last message: {:insult, "XAOP"}
State: []
```
Supervisor

- Supervisors supervise other processes. Used to build trees of processes and manages them by (re)starting them.
  - init/1

- Supervision trees allow you to contain failures in a tree.

- Strategies
  - one_for_one
  - one_for_all
  - rest_for_one
  - simple_one_for_one
Hello, World - Supervisor

defmodule Greeting.Supervisor do
  use Supervisor

  def start_link(args \ []) do
    Supervisor.start_link(__MODULE__, [args])
  end

  def init(args) do
    children = [ [ worker(Greeting.Server, []) ] ]
    supervise(children, strategy: :one_for_one)
  end
end
Hello, World - Supervisor

iex(1)> self()
#PID<0.106.0>
iex(2)> Supervisor.start_link(Greeting.Supervisor, [])
{:ok, #PID<0.109.0>}
iex(3)> GenServer.call(Greeting.Server, {:greet, "XAOP"})
Hello, XAOP
:ok
iex(4)> GenServer.call(Greeting.Server, {:insult, "XAOP"})
** (exit) exited in: GenServer.call(Greeting.Server, {:insult, "XAOP"}, 5000)
iex(4)> GenServer.call(Greeting.Server, {:greet, "XAOP"})
Hello, XAOP
:ok
iex(5)> self()
#PID<0.106.0>
GenEvent

- One generic event management process
- Many event handlers
- Many event producers
- Callbacks
  - code_change/3
  - handle_call/2
  - handle_event/2
  - handle_info/2
  - init/1
  - terminate/2
# Define an Event Handler

defmodule LoggerHandler do

  use GenEvent

  # Callbacks

  def handle_event({:log, x}, messages) do
    {:ok, [x | messages]}
  end

  def handle_call(:messages, messages) do
    {:ok, Enum.reverse(messages), []}
  end

end

> {:ok, mgr} = GenEvent.start_link([])
> GenEvent.add_handler(pid, LoggerHandler, [])
> GenEvent.notify(pid, {(:log, 1)})
> GenEvent.call(pid, LoggerHandler, :messages)
Application

“A reusable component with specific functionality that can be started and stopped as a whole while being reusable in other projects.”

- Second coarsest grained component in Elixir/OTP
- Created using mix: `mix new greeting`
- Dependencies (Applications)
- Configuration files
- Wrapper around Supervisor
Hello, World - Application

$ mix new hello_world
* creating README.md
* creating .gitignore
* creating mix.exs
* creating config
* creating config/config.exs
* creating lib
* creating lib/hello_world.ex
* creating test
* creating test/test_helper.exs
* creating test/hello_world_test.exs

Your Mix project was created successfully.
Hello, World - Application

defmodule HelloWorld do
  use Application

  def start(_type, _args) do
    Supervisor.start_link(Greeting.Supervisor, [])
  end
end

● Returns {:ok, pid} where pid is the PID of the Supervision tree
● _type: Mode of operation. Does not matter unless distributed (failover/takeover)
● _args: Passed in via mix.exs (application state/parameters)
● Optional implementation of stop/1 for cleanup
Hello, World - Application Config

defmodule HelloWorld.Mixfile do
  use Mix.Project

  def project do
    [app: :hello_world,
     version: "0.1.0",
     elixir: " ~> 1.4",
     build_embedded: Mix.env == :prod,
     start_permanent: Mix.env == :prod,
     deps: deps()]
  end

  defp deps do
    []
  end

  defp application do
    [extra_applications: [:logger],
     mod: {HelloWorld, []}
    ]
  end

  ...

  ...

$ iex -S mix
Interactive Elixir (1.4.2) - press Ctrl+C to exit (type h() ENTER for help)
  iex(1)> GenServer.call(Greeting.Server, {:greet, "XAOP"})
Hello, XAOP
:ok
Hello, World - Failover

- Run same application on distributed system
- Only one server is actually active, rest is idle
- Other server takes over upon failure
- Predefined order of take-over
- If higher-priority server is back up, takes back control
Hello, World - Failover

def start_link(args \ [],) do
  GenServer.start_link(__MODULE__, args, name: {global, __MODULE__})
end
Hello, World - Failover

def start_link(args \ []]) do
    GenServer.start_link(__MODULE__, args, name: {:global, __MODULE__})
end
Hello, World - Failover Config

```
[{kernel,
  [{distributed, [{hello_world,
      0,
      [a@anorak, {b@anorak, c@anorak}]}]},
   {sync_nodes_mandatory, []},
   {sync_nodes_optional, [b@anorak, c@anorak]},
   {sync_nodes_timeout, 5000}
  ]}
].
```

http://erlang.org/doc/design_principles/distributed_applications.html
Hello, World - Failover Startup

- **Server**
  ```
  export name="a"
  iex --sname "$name"
  -pa _build/dev/lib/hello_world/ebin
  --app hello_world
  --cookie cookie
  --erl "-config config/$name"
  ```

- **Client**
  ```
  iex --cookie cookie --sname client -S mix run --no-start
  ```
Demo Time
Macros

- Quoting returns the AST representation
  > ast = quote do: 1 + 2
- ASTs are represented in Elixir terms
  > {:+, [context: Elixir, import: Kernel], [1,2]}
- Unquoting injects a term into a macro
  > quote do: unquote(5 * 2) + 1
  {:+, [context: Elixir, import: Kernel], [10, 1]}
- ASTs can be evaluated using Code.eval_ast/1
  > Code.eval_quoted(ast)
  {2, []}
- Macros can be expanded using the Macro module
  > Macro.expand_once(unless true, do: false, __ENV__)
Macro Hygiene

A macro can never capture variables in it’s expansion environment.

But if you really want to..

defmacro dirty(exp) do
  quote do
    var!(x) = unquote(exp)
  end
end

> x = 10
  10
> MyMacro.dirty(100)
  100
> x
  100
The unless macro

> unless boring() == false do
  sleep()
  true
end
> nil
The unless macro

defmodule MyMacros do
  defmacro unless(expression, do: body) do
    quote bind_quoted: [body: body, expression: expression] do
      if !expression do
        body
      end
    end
  end
end
The unless macro

> require MyMacro
> unless true, do: 5
>   nil
> unless false, do: 1 + 2
>   3
defmodule MathTest do
  use Assertion
  test "integers can be added and subtracted" do
    assert 1 + 1 == 2
    assert 2 + 3 == 5
    assert 5 - 5 == 10
  end
  test "integers can be multiplied and divided" do
    assert 5 * 5 == 25
    assert 10 / 2 == 5
  end
end
Simple Unit Tests

defmodule Assertion do
  defmacro __using__(_options) do quote do
    import unquote(__MODULE__)
    Module.register_attribute __MODULE__, :tests, accumulate: true @before_compile unquote(__MODULE__)
  end

  defmacro test(description, do: test_block) do
    test_func = String.to_atom(description)
    quote do
      @tests {unquote(test_func), unquote(description)}
      def unquote(test_func)(), do: unquote(test_block)
    end
  end
  # ...
end
The most reliable, stable, resilient, distributed, performant, concurrent, awesome Slack bot

https://github.com/m1dnight/slackbot
Dependencies

- Slack connection ([https://hex.pm/packages/slack](https://hex.pm/packages/slack))
- JSON parser ([https://hex.pm/packages/poison](https://hex.pm/packages/poison))
- Date/Time ([https://hex.pm/packages/timex](https://hex.pm/packages/timex))
- RSS Feeds ([https://hex.pm/packages/feeder_ex](https://hex.pm/packages/feeder_ex))
Architecture

All components are grouped in “Supervisor Trees”

iex>:observer.start()
The Slack Behaviour

- 3 callbacks
  - handle_connect/2
  - handle_event/3
  - handle_close/3

- Not a GenServer, not supervisable.
  - But we want to supervise it!
The Slack Behaviour

defmodule SlackLogic do
  use GenServer
  use Slack
  def handle_connect(slack, state) do
    SlackManager.notify(:connected)
    {:ok, state}
  end

  def handle_event(message = %{type: "message", text: text}, slack, state) do
    SlackManager.notify(message)
    {:ok, state}
  end

  def handle_close(_reason, _slack, state) do
    SlackManager.notify(:disconnected)
    {:ok, state}
  end
end
defmodule SlackManager do
  use GenServer

defmodule State do
end

def start_link(client,token) do
  GenServer.start_link(__MODULE__, [client,token], name: __MODULE__)
end

def init([client,token]) do
  {:ok, %State{client: client, token: token}}
end

def handle_cast({:notify, m}, state) do
  for handler <- state.handlers, do: send(handler, m)
  {:noreply, state}
end
...
def notify(m) do
  GenServer.cast(SlackManager, {:notify, m})
end

Assume existence of a process named “SlackManager”
Storage

- Plain text files with Erlang terms
- Actor representing state
- Separate supervision tree

```erlang
defmodule Brain.Karma do
  use GenServer
  def increment(subject, amount \ 1) do
    GenServer.call __MODULE__, {:change, subject, amount}
  end
  def decrement(subject, amount \ -1) do
    GenServer.call __MODULE__, {:change, subject, amount}
  end
  def get(subject) do
    GenServer.call __MODULE__, {:get, subject}
  end
end
```
def handle_call({:get, subject}, _from, state) do
  ^subject, current_karma = List.keyfind(state, subject, 0, {subject, 0})
  {:reply, current_karma, state}
end

def handle_call({:change, subject, amount}, _from, state) do
  ^subject, current_karma = List.keyfind(state, subject, 0, {subject, 0})
  new_karma = current_karma + amount
  new_state = List.keystore(state, subject, 0, {subject, new_karma})
  content = new_state
  |> Enum.map(&[:io_lib.print(&1) | ".\n"])
  |> IO.iodata_to_binary
  File.write(data_backup_file(), content)
  {:reply, new_karma, new_state}
end
def increment(subject, amount \ 1) do
  GenServer.call __MODULE__, {:change, subject, amount}
end

def decrement(subject, amount \ -1) do
  GenServer.call __MODULE__, {:change, subject, amount}
end

def get(subject) do
  GenServer.call __MODULE__, {:get, subject}
end
Plugins (and also a first macro!)

- Define a custom behaviour to reduce boilerplate code

```elixir
defmodule Plugin do
  # Each Plugin should implement an on_message function.
  @callback on_message(message :: term, channel :: term) :: any

  # Optional callback to execute when the plugin starts.
  @callback initialize() :: any

  ...
```
Plugins (and also a first macro!)

- Define a custom behaviour to reduce boilerplate code

```elixir
defmacro __using__(_) do
  quote location: :keep do
    @behaviour Plugin
    use GenServer
    def init(args) do
      SlackManager.add_handler self() { :ok, args }
    end
    def handle_info(message = %{type: "message", text: text}, state) do
      reply = on_message(text, message.channel)
      ...
    end
  end
end
```
Plugins (and also a first macro!)

- Define a custom behaviour to reduce boilerplate code

```python
def initialize(), do: :ok
defoverridable initialize: 0
```
A Plugin

defmodule Bot.ChuckNorris do
  use Plugin
  @url 'http://api.icndb.com/jokes/random'
  def on_message(<<"joke?"::utf8, _::bitstring>>, _channel) do
    j = joke()
    case j do
      {:error, e} -> IO.puts "Error getting joke #{e}"
                   {:noreply}
      {:ok, text} -> {:ok, "#{text}"}
    end
  end
  end
  #Catch-all
  def on_message(_m, _channel) do
    {:noreply}
  end
end
Thanks!

- Elixir is built on a battle-tested virtual machine
- Pattern matching, reduces a lot of code
- BEAM offers you, with little to no effort:
  - Distribution
  - Scalability
  - Reliability
  - More-ities
- OTP offers you modularity by construction (if you follow the unwritten rules)
- Slackbot is awesome
Some ideas..

- Interconnect all Slackbots and obtain world domination
- Use the Erlang ETS table
- Use macros to remove even more boilerplate from Plugins
- Use a database for the bot’s brain
  https://hexdocs.pm/ecto/Ecto.html
- Let it proxy cleverbot
  http://www.cleverbot.com/api
- Patches welcome!