Introduction to Elixir, Getting Elixir

- Local copy
  - http://172.16.0.20

- Elixir
  - elixir-lang.org

- Erlang
  - erlang.org

- Pre-built packages
  - erlang-solutions.com
An Introduction to Elixir
Painlessly Functional and Concurrent

Marc Sugiyama
Erlang Solutions, Inc
We do not have ONE webserver handling 2 millions sessions. We have 2 million webservers handling one session each.

Joe Armstrong
Erlang VM

Scheduler #1
run queue

Scheduler #2
run queue

...  

Scheduler #N  run queue

migration logic
Starting the System

• Start Elixir with **iex** on the unix shell or double-clicking the Elixir icon in Windows. This starts an Elixir shell.

• **h()** – Prints help message with list of shell helpers available

• **h(which)** – Prints help for the given module, function or macro

• Use the keys **CTRL + (n, p, f, b, y, a, e, ...)** to edit command lines as similarly to Emacs.
The Elixir Shell

$ iex
Erlang/OTP 17 [erts-6.4] [source] [64-bit] [smp:8:8] [async-threads:10]
hipe] [kernel-poll:false]

Interactive Elixir (1.0.5) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> c(“demo.ex”)
[Demo]
iex(2)> Demo.double(25)
50
iex(3)> Demo.times(4,3)
** (UndefinedFunctionError) undefined function: Demo.times/2
    Demo.times(4, 3)

iex(4)> 10 + 25
35
iex(5)>
## Data Types: integers and floats

<table>
<thead>
<tr>
<th>Number</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100_000_000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-234</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xAB10F</td>
<td></td>
<td>hexadecimal</td>
</tr>
<tr>
<td>0b1010</td>
<td></td>
<td>binary</td>
</tr>
<tr>
<td>?a</td>
<td></td>
<td>unicode values</td>
</tr>
<tr>
<td>?\n</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17.368</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-56.654</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12.34e-10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12_000.5_6e23</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Numbers in other bases:
  - 0xNNN for hexadecimal
  - 0bNNN for binary
- ?Char is used for unicode values
- Large integers are converted to bignums
- Floats are stored as a double
  - 64-bit representation
- Follows the IEEE 754 standard
Data Types: atoms

:january
:January
:fooBar
:hey_there
:well?
:yes!

:++
:--

:"a space"
:"Anything inside quotes{}#@ \n \012"
:"node@ramone.erlang.org"
:'this is "an" atom'
:"or "this" way"

- Atoms are constant literals
- Start with a letter or are surrounded by quotes
- ':' is not part of the atom
- Letters, digits and _ are allowed if the atom starts with a letter
- Operators are atoms
- Any character is allowed between '"'
Data Types: booleans

- true
- false
- 1 == 1.0
- 1 != 2
- 1 === 1
- 1 !== 1.0
- 1 < 2
- :a > :z
- :less < :more

- is_boolean(9+6)
- is_boolean(:true)
- not((1 < 3) and (2 == 2))
- not((1 < 3) or (2 == 2))
- not((1 < 3) xor (2 == 2))

- No separate boolean type
- true and false same as the atoms :true and :false
- Operators (and, or, xor, not) take and return true and false as if they were boolean types
- Operators short-circuit
Data Types: **truthy values**

- No separate type for nil
- **nil** same as the atom **:nil**
- Operators **&&** and **||** treat **nil** and **false** as false, everything else as true
- Short circuits
- Returns last value (not **true** or **false**)

```
nil
false

59 && :anything
59 && nil

nil || :anything
false || :anything
59 || :anything
```
Data Types: **tuples and lists**

- Tuples are used to denote datatypes with a fixed number of items
- Lists are used to store a variable number of items
- In a tuple each position has its own value and meaning
- Lists are often used for iteration

```erlang
{123, :bcd}
{:abc, {:def, 123}, :ghi}
{:person, "Joe", "Armstrong"}

[:january, :february, :march]
[:a,[:b,[:c,:d,:e],:f],:g]
[]
[{:person, "Joe", "Armstrong"},
 {:person, "Robert", "Virding"},
 {:person, "Mike", "Williams"}]

[?H,?e,?l,?l,?o]
'Hello'
```
Data Types: **lists**

- A recursive list definition consists of a head and a tail.
- Lists whose last tail term is `[]` are called:
  - **proper** lists or
  - **well formed** lists
- The tail may be any valid value, but...
- Most functions expect **proper lists**

\[ \text{List} = [\text{Element} \mid \text{List}] \text{ or } [] \]

- `[:one, :two, :three]`
- `[:one, :two, :three \mid []]`
- `[:one, :two \mid [:three]]`
- `[:one, :two \mid [:three\mid []]]`
- `[:one\mid [:two \mid [:three \mid []]]]`
Data Types: **keyword property lists**

\[
\begin{align*}
\{\text{foo: 1, bar: [2, 3]}\} & = [\{\text{foo, 1}\}, \{\text{bar, [2, 3]}\}] \\
[17, 42, \text{foo: 1, bar: 2}] & = [17, 42, \{\text{foo, 1}\}, \{\text{bar, 2}\}] \\
\{\text{foo: 1, 17, 42, bar: 2}\} & \Rightarrow \text{syntax ERROR!} \\
[\{\text{foo, 1}\}, 17, 42, \{\text{bar, 2}\}] &
\end{align*}
\]

- Special syntax for **property lists**, lists of \{key, value\} tuples, where the key is an atom
Data Types: **strings and binaries**

- "Joe Armstrong"
- "Det här går bra"
- "Line 1\nLine 2"
  
  "Include #{value} expressions"

```elixir
t = {1,2,3}
"foo #{inspect(t)} bar"
```

- Can contain any Unicode codepoint
- Uses the binary data structure with UTF-8 encoding
- Interpolate Elixir string expressions with #{ ... }
Maps and Structs

```elixir
%{name: "Joe", phone: [1, 2, 3, 4]}
```

- Maps holds key/value pairs
- Keys and values are any Elixir data
- Structs are defined at compile time and keys are limited by the definition
Complex Data Structures

- Arbitrarily complex data structures are created by nesting
- Data structures do not need to be defined
- No explicit memory allocation or deallocation is needed
  - Allocated automatically
  - Deallocated by the *garbage collector* when no longer referenced.
Complex Data Structures

```erlang
[{{:person, "Joe", "Armstrong"},
  %{telephone_number: [3,5,9,7],
   shoe_size: 42,
   pets: [cat: "tubby", cat: "tiger"],
   children: [{"thomas", 5}, {"claire", 1}]}
},
 {{:person, "Mike", "Williams"},
  %{shoe_size: 41,
   likes: [:boats, :beer]}
}
]
```
Variables

- No need to declare
- Variables can start with a lowercase letter or `_`
- They may not contain any 'funny characters'
- `_` alone is a don't care variable
  - Its values are ignored and never bound
- Elixir does not have a static type system
- Types are determined at run time
Pattern Matching

Pattern = Expression

- Pattern matching is used for:
  - Assigning values to variables
  - Controlling the execution flow of programs (if, case, function heads)
  - Extracting values from compound data types
  - The pattern can contain variables which are bound when the matching succeeds

- The expression may not contain unbound variables

© 1999-2016 Erlang Solutions Ltd.
Pattern Matching: assigning

\[ a = 10 \]
Succeeds, binds \( a \) to 10

\[ \{b, c, d\} = \{10, :foo, :bar\} \]
Succeeds, binds \( b \) to 10, \( c \) to \( :foo \) and \( d \) to \( :bar \).

\[ \{e, e, :foo\} = \{:abc, :abc, :foo\} \]
Succeeds, binds \( e \) to \( :abc \).

\[ [h|t] = [1,2,3] \]
Succeeds, binds \( h \) to 1, \( t \) to [2,3].
Pattern Matching: controlling

A match must either succeed or fail

\{a, a, b\} = \{:abc, :def, :123\}
- fails

[a,b,c,d] = [1,2,3]
- fails

[a,b|c] = [1,2,3,4,5,6,7]
- succeeds, a = 1, a =2, c = [3,4,5,6,7]

[h|t] = []
- fails
Pattern Matching: extraction

\{a, _, [b|\_], \{b\}\} = \{:abc, 23, [22, :x], \{22\}\}
- Succeeds, a = abc, b = 22

\text{c} = 10, 
\{c, c, 13, d, \_\} = \{11, 11, 13, 12, 15\}
- Succeeds, d = 12, c = 11

\text{c} = 10, 
\{^c, ^c, 13, d, \_\} = \{11, 11, 13, 12, 15\}
- fails!

\%\{name: name\} = \%\{name: "Joe", age: 35\}
- Succeeds, name = "Joe"
Modules and Functions

- Elixir programs consist of functions that call each other
- Functions are defined within modules
- Modules files have the `.ex` suffix
- Multiple modules can be defined in one file
- Name Modules with `defmodule Alias`
- Module alias always starts with Uppercase Letter
defmodule Demo do
  ## This is a comment.
  ## Everything after '#' is ignored.

  def double(x) do
    times(x, 2)
  end

  defp times(x, n) do
    x * n
  end
end
Modules

- Exported functions may be called from other modules with a **fully qualified call**
  - `Module.fun(arg1, ...)`
- Local/private functions may only be called within the module
Functions

- A function must be an atom
- The **arity** of a function is its number of arguments.

```erlang
Module.function(Arg1, Arg2, ..., ArgN)
function(Arg1, Arg2, ..., ArgN)
```
Functions

```erlang
def circumference(r) do
    # call Erlang math module
    2 * :math.pi() * r
end

def product(x, y) do
    x * y
end

def product(x, y, z) do
    x * y * z
end
```
Functions: Pattern matching

def area({:square, side}) do
    side * side
end
def area({:circle, radius}) do
    :math.pi() * radius * radius
end
def area({:triangle, a, b, c}) do
    s = (a + b + c) / 2
    :math.sqrt(s*(s-a)*(s-b)*(s-c))
end
Guards

```erlang
def number(num) when is_integer(num) do :integer end
def number(num) when is_float(num) do :float end
def number(_Other) do false end
```

- `is_number(x)`, `is_integer(x)`, `is_float(x)`
  - x is a number
- `is_atom(x)`, `is_pid(x)`, `is_tuple(x)`, `is_list(x)`
  - x is the specified datatype
- `length(list) == int, tuple_size(tuple) == size, x > y + z`
  - Some functions and mathematical applications can be applied in guards
- `x == y   x != y   x === y   x !== y`
  - x is (not) equal to y, x is exactly (not) equal to y (1==1.0 ✓, 1===1.0 ✗)
- `x <= y   x >= y`
  - x is less than or equal to y, x is greater than or equal to y
Elixir Syntax: calling function

- foo_function(:pets, [dogs: 35, cats: 47, budgies: 5])
- foo_function(:pets, dogs: 35, cats: 47, budgies: 5)
- foo_function :pets, dogs: 35, cats: 47, budgies: 5

- These are all the same
- foo_function gets two arguments, :pets and a property list
- You can omit parentheses around arguments in function calls
- If an argument looks like an element in a property list all remaining arguments become part of that property list
Elixir Syntax: do blocks

```elixir
def foo(n) do y = bar(x) ; x + y end

def foo(n), do: (y = bar(x) ; x + y)

def foo(n), do: n+5

• do...end is transformed into do: property list argument
• do...end is never separated with comma from previous argument
• do: is common in function clauses with a simple body
```

Need parentheses
Functions: examples

def factorial(0) do 1 end
def factorial(n) do
  n * factorial(n-1)
end

> factorial(3)
  (matches n = 3 in clause 2)
  == 3 * factorial(3-1)
  (matches clause 2)
  == 3 * 2 * factorial(2-1)
  (matches clause 2)
  == 3 * 2 * 1 * factorial(1-1)
  (matches clause 1)
  == 3 * 2 * 1 * 1
  == 6

• Pattern matching occurs in the function head
  – Unbound variables get bound after a successful pattern match
• Variables are local to each clause
• Variables are allocated and deallocated automatically
Recursion: **traversing lists**

- **def average(x) do**
  - sum(x) / len(x)
  end

- **def sum([h|t]) do**
  - h + sum(t)
  end
- **def sum([]) do 0 end**

- **def len([_|t]) do**
  - 1 + len(t)
  end
- **def len([]) do 0 end**

- Note the pattern of recursion is the same in both cases

- Taking a list and evaluating an element is a very common pattern
Recursion: using accumulators

```erlang
def average(x) do
    average(x, 0, 0)
end

def average([h|t], len, sum) do
    average(t, len+1, sum+h)
end

def average([], len, sum) do
    sum/len
end
```

- Only traverses the list once.
- Executes in constant space (tail recursive)
- `len` and `sum` play the role of accumulators
- `average([])` is not defined
- Evaluating `average([])` would cause a run time error.
Anonymous functions

Anonymous functions are data types encapsulating functional objects
They can be passed as arguments
They can be the return value of function calls
Note the ‘.’ after the variable name to call the function
Functions: capture operator

Using the capture operator &:

\[ f = &\text{function}/\text{Arity} \]
Binds the local function in the current module to \( f \)

\[ f = &\text{Module.function}/\text{Arity} \]
Binds the function exported in Module to \( f \)

\[ f = &\text{Map.fetch!}(%\{a: :abra\}, &1) \]
Binds a function of one argument to \( f \), \&1 is a placeholder

\[ f = &(&1 + &2) \]
Will bind a function of two arguments to \( f \)
Funs: closures

defmodule Fun do
  def bar(x) do
    fn(y) -> x + y end
  end
end

iex(1)> f = Fun.bar(10)
iex(2)> f.(10)
20
Funs: procedural abstraction

Before

```erlang
def double([h|t]) do
    [2*h | double(t)]
end
def double([]) do
    []
end

def bump([h|t]) do
    [h+1 | bump(t)]
end
def bump([]) do
    []
end
```

After

```erlang
def map([h|t], fun) do
    [fun.(h) | map(t, fun)]
end
def map([], _fun) do
    []
end
def double(l) do
    map(l, fn(x)-> 2*x end)
end
def bump(l) do
    map(l, &(&1+1))
end
```
Funs: higher order functions

Enum.all?(collection, predicate) -> true | false
  Returns true if the Predicate fun returns true for all elements in collection

Enum.filter(collection, predicate) -> collection
  Returns a collection with elements for which Predicate is true

Enum.each(collection, fun) -> :ok
  Applies Fun on every element in the collection.

Enum.map(collection, fun) -> collection
  Returns a list with the return value of Fun applied to all elements in collection.
Conditional Evaluation: if

test expression

if String.valid?(s) do
  IO.puts(“good string”)
else
  IO.puts(“bad string”)
end
Conditional Evaluation: cond

size = cond do
  String.length(S) < 5 -> small
  String.length(S) < 10 -> medium
  true -> big
end
Conditional Evaluation: case

```erlang
member = case Enum.member?(list, :foo) do
  true -> :ok
  false -> {:error, :unknown}
end
```
Defensive Programming

```erlang
def convert(day) do
  case day do
    :monday    -> 1
    :tuesday   -> 2
    :wednesday -> 3
    :thursday  -> 4
    :friday    -> 5
    :saturday  -> 6
    :sunday    -> 7
    other      -> {:error, :unknown_day}
  end
end
```

- Defensive programming: program in the convert function for the error case or ...
- ... let it fail here by deleting the `other` clause.
- This will raise an exception
- The caller will have to handle the error that they have caused.
Elixir Syntax: macros

defmodule MyMacros do
  defmacro unless(expr, body) do
    quote do: if( !unquote(expr), unquote(body) )
  end
end

require MyMacro
MyMacro.unless var, do: IO.puts "false"
MyMacro.unless var do IO.puts "false" end
MyMacro.unless(var) do IO.puts "false" end
Elixir Syntax: macros

MyMacro.unless var do IO.puts "false" end

defmacro unless(expr, body) do
  quote do: if( !unquote(expr), unquote(body) )
  end

  if( !var, do IO.puts "false" end )
Manual pages

- Manual pages for all the modules, both Elixir and Erlang, can be read online
  - [http://elixir-lang.org/docs](http://elixir-lang.org/docs)
  - [http://erlang.org/doc](http://erlang.org/doc)

- **Kernel** and **Process** modules are a good starting point
Why have a Process?

- Maintain a State
- Interact with the outside world
- Serialization
- Concurrency
- Parallelism
- Generate asynchronous events
- Handle asynchronous events
Creating Processes

- **Before**
  - Code executed by Process 1
  - **process identifier** is pid1
  - pid2 = spawn(M, f, as)

- **After**
  - A new process with pid2
  - pid2 is known only to pid1
  - pid2 runs M.f(as)
  - M.f/Arity must be exported

- **Convention:** we identify processes by their process ids (pids)
Creating Processes

- Spawning a process never fails*
- A process terminates
  - abnormally when run-time errors occur
  - normally when there is no more code to execute

```
spawn(Mod, func, args)
```

pid1 spawns

pid2
Message Passing

- Messages are sent using the expression `send(pid, msg)`
  - `msg` is any Elixir data
- Sending a message never fails*
- Messages sent to non-existing processes are discarded
- Messages are held in the receiving process's mailbox
Message Passing

```erlang
receive do
  {:reset, board} -> reset(board)
  {:do, signal}  -> process(signal)
end
```

**clauses**

**patterns**

**clause body**
Receiving Messages

- Messages can be matched and retrieved selectively
- Messages are retrieved when a message matches a receive clause
- The mailbox is scanned sequentially

```erlang
receive do
  :start -> ...
  :stop  -> ...
  {pid, :foo} ->
  {pid, :foo} ->
  ...
end
```

```
send(pid2, {self(), :foo})
```

- pid1
  - {pid1, :foo}

- pid2
Receiving Messages: **selective**

- The message `:foo` is received, followed by the message `:bar`
- This is irrespective of the order in which they arrived in the mailbox

```
pid1 receive do  % pid1 receiving
  :foo -> true
end

send(pid3, :foo)

pid2 receive do  % pid2 receiving
  :bar -> true
end

send(pid3, :bar)

pid3
```

```
```

© 1999-2016 Erlang Solutions Ltd.
Receiving Messages: non-selective

- The first message to arrive at the process `pid3` is retrieved first.
- The variable `msg` in the process `pid3` is bound to one of the atoms `:foo` or `:bar` depending on which message gets to the mailbox first.

```erlang
receive do
  msg -> true
end, ...
```

```erlang
send(pid3, :foo)
send(pid3, :bar)
```
Receiving Messages

• pidA sends a message to pidB containing its own pid
• pidB sends the pid in a message to pidC
• pidC replies directly to pidA

Erlang code:

```erlang
receive do
  {:transfer, pid} ->
  send(pidC, {:transfer, pid})
end
```
defmodule Echo do
  def go() do
    pid = spawn(Echo, :loop, [])
    send(pid, {self(), :hello})
    receive do
      {^pid, msg} ->
        IO.puts.inspect(msg)
    end
    send(pid, :stop)
  end
end

def loop() do
  receive do
    {from, msg} ->
      send(from, {self(), msg})
      loop()
    :stop -> :ok
  end
end
Registered Processes

Process.register(pid, :alias)
Process.send(:alias, message)

- Registers the process `pid` with the name `:alias`
- Any process can send a message to a registered process
- The function `Process.registered/0` returns all registered process names
- The function `Process.whereis(:alias)` returns the pid of the process with the name `:alias`. 
Registered process

```erlang
defmodule Echo do
  def go() do
    pid = spawn(Echo,:loop,[])
    Process.register(:echo, pid)
  end
  def loop() do
    receive do
      {from, msg} ->
        send(from, {self(), msg})
        loop()
      :stop -> :ok
    end
  end
end
```

```
send(:echo, {self(), :hello})
```

```erlang
defmodule Echo do
  def go() do
    pid = spawn(Echo,:loop,[])
    Process.register(:echo, pid)
  end
  def loop() do
    receive do
      {from, msg} ->
        send(from, {self(),msg})
        loop()
      :stop -> :ok
    end
  end
end
```

![Diagram](image)
Message Passing

- Sending a message to an alias that isn’t registered causes the sending process to terminate with an `ArgumentError` error

```erlang
pid1 send(:is_not_registered, :hello)
```
Timeouts

- If the message \texttt{msg} is received within the time \texttt{timeOut}, \texttt{<expressions1>} is executed
- If not, \texttt{<expressions2>} is executed
- \texttt{timeOut} is an integer denoting the time in milliseconds or the atom \texttt{:infinity}

```erlang
receive do
  msg ->
    <expressions1>
  after timeOut ->
    <expressions2>
end
```
```erlang
def read(key) do
  send(:db,{self(),{:read,key}})
  receive do
    {:read, value} ->
      {:ok, value}
    {:error, reason} ->
      {:error, reason}
    after 1000 ->
      {:error, :timeout}
  end
end
```

**Timeouts**

- If the server takes more than a second to handle the request, a timeout is generated.
- Do not forget to handle messages received after a timeout.

© 1999-2016 Erlang Solutions Ltd.
task = Task.async(fn() -> DO_SOMETHING end)
DO_SOMETHING_ELSE
result = Task.await(task)

• Simple way to add parallelism
• Task.async/1 starts process to run a fun
• Retrieve the fun's return with Task.await/1
Parallelism

```erlang
def pbump() do
  1..1_000_000
  |> Enum.map(fn(v) -> Task.async(fn() -> bump(v) end) end)
  |> Enum.map(fn(t) -> Task.await(t) end)
end

defp bump(v) do
  v + 1
end
```

- Use Task to run funs in parallel to shorten wall clock time
But...

• Too much parallelism?
• Overhead of spawn and context switching
• Bottleneck is elsewhere
• Don't overload a downstream service
  – database connections
  – backend service
def psum() do
  1..1_000_000
  |> Enum.chunk(100_000, 100_000, [])
  |> Enum.map(fn(l) -> Task.async(fn() -> sum(l) end) end)
  |> Enum.map(fn(t) -> Task.await(t) end)
  |> sum
end

defp sum(l) do
  Enum.reduce(l, &(&1 + &2))
end

- Chunk list and start a task for each chunk
- Shortens wall clock time over single process but limits degree of parallelism
Client Server Models

- Processes can be used to implement client server solutions
- A server is usually responsible for providing a service or handling a resource
- Clients are the processes which use these resources
Client Server Models

- Clients make requests to the server through message passing
- Message passing is often hidden in functional interfaces
- A request is termed **synchronous** if the client needs a reply to the request
- A request is **asynchronous** if the client does not require a reply from the server
A Server Example

- The server is responsible for allocating and de-allocating frequencies on behalf of mobile phones.

Clients

:allocate

{:ok, frequency}

Server

:ok

{:deallocate, freq}
A Server Example

Client

{:request, pid, :allocate}

{:reply, {:error, :no_frequencies} or {:ok, frequency}}

{:request, pid, {:deallocate, frequency}}

{:reply, :ok}

Server
A Server Example

defmodule Frequency do

def start() do
  pid = spawn(Frequency, :init, [])
  Process.register(:frequency, pid)
end

def init() do
  frequencies = {get_frequencies(), []}
  loop(frequencies)
end

defp get_frequencies() do
  [10,11,12,13,14,15]
end

© 1999-2016 Erlang Solutions Ltd.
A Server Example

def stop() do call(:stop) end

def allocate() do call(:allocate) end

def deallocate(freq) do call({:deallocate, freq}) end

## We hide all message passing and the message protocol
## in functional interfaces.
defp call(message) do
  send(:frequency, {:request, self(), message})
  receive do
    {:reply, reply} -> reply
  end
end

defp reply(pid, message) do
  send(pid, {:reply, message})
end
A Server Example

```erlang
defp loop(frequencies) ->
  receive
    {:request, pid, :allocate} ->
      {newfrequencies, reply} =
        allocate(frequencies, pid)
      reply(pid, reply)
      loop(newfrequencies)
    {:request, pid, {:deallocate, freq}} ->
      newfrequencies =
        deallocate(frequencies, freq)
      reply(pid, :ok)
      loop(newfrequencies)
    {:request, pid, :stop} ->
      reply(pid, :ok)
  end
end
```

- Frequency data in loop variable
- Messages are explicit
  - request
  - pid of sender
  - service required
- Loop again with updated frequency data
A Server Example

### The Internal help functions
### Used to allocate and deallocate frequencies.

defp allocate([], allocated, pid) do
  {{[], allocated}, {:error, :no_frequencies}}
end
defp allocate([freq|free], allocated, pid) do
  {free, [{freq, pid}|allocated]}, {:ok, freq}}
end
defp deallocate([freq|free], allocated, freq) do
  newallocated = :lists.keydelete(allocated, freq, 1)
  {[freq|free], newallocated}
end
def start(args) do
    spawn(Server, :init, [args]) end
def init(args) do
    state = initialize_state(args)
    loop(state)
end
def loop(state) do
    receive do
        {:handle, msg} ->
            newstate = handle(msg, state)
            loop(newstate)
        :stop -> terminate(state)
    end
end
def terminate(state) do clean_up(state) end
Finite State Machines

- Each state is represented as a tail recursive function.
- Each event is represented as an incoming message.
- Each state transition is achieved by calling the function denoting the new state.
Finite State Machines: example

def idle() do
  receive
    {a, :incoming} ->
      start_ringing()
      ringing(a)
    {a, :off_hook} ->
      start_tone()
      dial(a)
  end
end
Event Handlers

- A handler receives a specific type of event
  - Alarms
  - Equipment state changes
  - Errors

- When an event is received the handler applies one or more functions to the event

- Some or all of these actions can be enabled and disabled during run time
Event Handlers: example

- Alarm managers are implemented as event handlers
Links

- **Process.link/1** will create a bi-directional link between the process calling the BIF and the process `pidB`.

- **spawn_link/3** will yield the same result as calling `spawn/3` followed by `Process.link/1`, only that it will do it **atomically**.
**Links**

- **Exit Signals** are sent when processes terminate abnormally
- They are sent to all processes linked to the failing process
- The process receiving the signal exits
- The exiting process propagates a new signal to all of its linked processes

![Diagram showing links between processes pidA and pidB with signal {:EXIT, pidA, reason}]

© 1999-2016 Erlang Solutions Ltd.
When process \texttt{pidA} fails, the exit signals propagate to \texttt{pidB}.

From \texttt{pidB}, it propagates to \texttt{pidC}.
Exit Signals

- Processes can trap exit signals by calling `Process.flag(:trap_exit, true)`
- Exit signals are converted to messages: `{:EXIT, pid, reason}`
- Since the exit signal is trapped it does not propagate further
Exit Signals

- Process B marked with a double ring is trapping EXITs.
- If an error occurs in A or C, then they terminate.
- Process B will receive the \{:EXIT, pid, reason\} message.
- The process that did not terminate is not affected.
Built-in functions

- `exit(reason)` terminates the process
- `exit` generates an exit signal sent to linked processes
- `Process.exit(pid, reason)` sends an exit signal containing `reason` to the process `pid`
- If trapping exits, the signal is converted to an exit message
A Robust Server

defmodule Frequency do

def start() do
  pid = spawn(Frequency, :init, [])
  Process.register(pid, :frequency)
end

def init() do
  Process.flag(:trap_exit, true)
  frequencies = {get_frequencies(), []}
  loop(frequencies)
end

defp get_frequencies() do [10,11,12,13,14,15] end
defp allocate([], allocated, pid) do
  {{[], allocated}, {:error, :no_frequencies}}
end

defp allocate([freq|free], allocated, pid) do
  Process.link(pid)
  {{free, [{freq, pid}|allocated]}, {:ok, Freq}}
end

defp deallocate([free, allocated], freq) do
  {freq, pid} = List.keysearch(allocated, freq, 1)
  Process.unlink(pid)
  newallocated = List.keydelete(allocated, freq, 1)
  {{freq|free}, newallocated}
end
A Robust Server

def loop(frequencies) ->
    receive
        {:request, pid, :allocate} ->
            {newfrequencies, reply} = allocate(frequencies, pid)
            reply(pid, reply)
            loop(newfrequencies)
        ...
        {:EXIT, pid, reason} ->
            newfrequencies = exited(frequencies, pid)
            loop(newfrequencies)
        {:request, pid, :stop} -> reply(pid, :ok)
    end
end
A Robust Server

## Help functions used when a client crashes.

defp exited({free, allocated}, pid) do
  case List.keyfind(allocated, pid, 2) do
    {freq, pid} ->
      newallocated = List.keydelete(allocated, freq, 1)
      {[freq|free], newallocated}
    nil ->
      {free, allocated}
  end
end

The :EXIT message was sent before the server unlinked, but after it released the frequency
A Server Example

Client

{:request, pid, :allocate}

{:reply, {:error, :no_frequencies}} or {:ok, frequency}

{:request, pid, {:deallocate, frequency}}

{:EXIT, pid, reason}

{:reply, :ok}

Server
Open Telecom Platform
Open Telecom Platform

Applications & Libraries

System Design Principles

© 1999-2016 Erlang Solutions Ltd.
System Design Principles

A set of abstract principles and design rules
- They describe the software architecture of an Erlang System
- Allows you to write tools compatible with the structure
- Facilitate the understanding of the system

A set of generic behaviours
- Each behaviour is a framework containing generic code
- Solves a common problem
- Built-in support for debugging and software upgrade
- Generic error handling behind the scenes
- Facilitates understanding the sub blocks in the system
System Design Principles

- An application is a logical unit of processes and modules grouped together to perform a given task
- Application are trees of supervisors and their workers
- They can be implemented using generic behaviours
System Design Principles

Hardware / Operating System

User Defined OTP Applications

My App | My App | SNMP | Corba | Tools | Inets | My Code | C |

My App | My App | Mnesia | SASL | Std. Library | Kernel | Sourced | Components |

ERLANG

© 1999-2016 Erlang Solutions Ltd.
Standard Applications and Libraries

Basic Applications
- Erlang Runtime System, Kernel, Compiler, Standard Lib,
- System Architecture Support Library (SASL)

Database Applications
- Mnesia (Distributed relational database)
- ODBC (Interface for accessing SQL databases)

Operations and Maintenance Applications
- Operating System Monitor
- Simple Network Management Protocol
- OTP MIBs
Generic Servers

**Generic**
- Spawning the server
- Storing the loop data
- Sending requests to the server
- Sending replies to the client
- Receiving server replies
- Stopping the server

**Specific**
- Initialising the server state
- The loop data
- The client requests
- Handling client requests
- Contents of server reply
- Cleaning up
Generic Servers

- The **GenServer** module implements the client-server behaviour
  - It is part of the standard library application
- The library module contains all the generic code
- Non-generic code is placed in the callback module
A Generic Server: starting

defmodule Frequency do
  use GenServer

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

  def init(_args) do
    free = get_frequencies(),
    {:ok, {free, []}}
  end
end
A Generic Server: synchronous call

def allocate() do
  GenServer.call(:frequency, {:allocate, self()})
end

def handle_call({:allocate, pid}, _from, frequencies) do
  {newfrequencies, reply} = allocate(frequencies, pid)
  {:reply, reply, newfrequencies}
end
A Generic Server: asynchronous call

def deallocate(freq) do
    GenServer.cast(:frequency, {:deallocate, freq})
end

def handle_cast({:deallocate, freq}, frequencies) do
    newfrequencies = deallocate(frequencies, freq)
    {:noreply, newfrequencies}
end
def stop() do
  GenServer.cast(:frequency, :stop)
end

def handle_cast(:stop, frequencies) do
  {:stop, :normal, frequencies}
end

def terminate(_reason, _frequencies) do
  :ok
end
Mod.handle_info(message, data) -> {:noreply, newdata}
Mod.code_change(version, data, extra)

- The GenServer adds default callbacks
Generic Supervisors

**Generic**
- Spawning the supervisor
- Starting the children
- Monitoring the children
- Restarting the children
- Stopping the supervisor
- Cleaning up

**Specific**
- What children to start
- Specific child handling
  - Start, Restart
- Child dependencies
- Supervisor name
- Supervisor behaviours
Generic Supervisors

- Supervisors are implemented in the `Supervisor` module
A Supervisor Example

defmodule FreqSup do
  use Supervisor

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

  def init(_arg) do
    server = worker(Frequency, [])
    # Returns {:ok, supervisorspec}
    supervise([server], strategy: :one_for_one)
  end
end
Supervisor specification

- The supervisor specification is a tuple containing:
  - Supervisor non-generic information on the restart strategy
  - Child specifications for all static children
- Restart: :permanent, :transient, :temporary
- Strategy: :simple_one_for_one, :one_for_one, :one_for_all, :rest_for_one
Supervisor Strategies

one_for_one

one_for_all

rest_for_one
Restart Strategy

- permanent – always restarted
- temporary – never restarted
- transient – restart on abnormal exit
Putting it All Together

- Isolation
- Messages
- Process Linking
- Trap Exit
- Tail Call Optimization

- No defensive programming
- Robust, concurrent, parallel system
An Introduction to Elixir
Painlessly Functional and Concurrent

Marc Sugiyama
Erlang Solutions, Inc
Using mix
Building a Project

- Releases
- Directory Structure
- Using Templates
- Compiling, Testing
- Dependencies
- Building a Release
Release

Node

Dependent Apps

.app
.beam
.beam
.beam

.git
.rel
Application Tree

- **appname/**
  - **mix.exs**
  - **_build/**
  - **config/**
    - **config.exs**
  - **deps/**
  - **lib/**
  - **test/**

- **mix.exs** – module with callback functions describing the project and application configuration
- **_build** – build artifacts
- **config.exs** – sets the application's environment
- **deps** – dependent applications
- **lib** – source code
- **test** – test code
Umbrella Application Tree

- `appname1` is an application tree
- Umbrella projects split source among more than one local application
- For larger projects
- Cannot use exrm (Elixir release tool)
mix.exs

def project do
  [app: :app_name,
   version: "0.0.1",
   elixir: "~> 1.2",
   build_embedded: Mix.env == :prod,
   startPermanent: Mix.env == :prod,
   deps: deps]
end

• mix project settings
• sets the release's version
• deps/0 is a local function returning the list of repository dependencies
mix.exs

def application do
  [applications: [:logger],
   description: 'this is my app',
   mod: {AppName, []}] end

• applications – list of dependent applications
• description – human description (as a list string)
• mod – application behaviour's callback module; only present if the application has processes
Starting a Project

$ mix new dirnamen --module ModuleName

• Creates a directory dirname for the module
• Use for starting a project that is a library with no processes
• Same as: mix dirname --app module_name
• Usually dirname and appname are the same
Starting a Project

$ mix new --sup dirname --app app_name

- Creates a directory dirname for the application
- Use for starting a project that has processes
- Creates a template for the top level supervisor
- Usually dirname and app_name are the same
Starting a Project

$ mix new dirname --umbrella app_name

- Creates a directory dirname for the umbrella application
- Usually dirname and app_name are the same
- Use mix to create applications under the umbrella's apps directory
Dependencies

defp deps do
  [
    {:lager, github: "basho/lager", tag: "3.1.0"}
  ]
end

• Modify mix.exs
• Adds lager as a dependency
$ mix deps.get

- Recursively fetches dependencies
- Locks version in mix.lock
  - unlock version with mix deps.unlock before updating deps
Development Build

$ mix compile

- dev is the default build mode
- Compiles .ex to .beam files in _build/dev
- dev has symbolic links back to lib for easier debugging
- Erlang VM keeps running even if top level supervisor crashes
Production Build

$ MIX_ENV=prod mix compile

- Production build
- Objects in _build/prod
- Builds are self contained to the _build directory (no symbolic links)
- Erlang VM crashes if top level supervisor crashes
Starting a Shell

$ iex -S mix
Erlang/OTP 18 [erts-7.2.1] [source] [64-bit] [smp:8:8]
[async-threads:10] [hipe] [kernel-poll:false]

Interactive Elixir (1.2.1) – press Ctrl+C to exit (type h() ENTER for help)
iex(1)>

- Run from top level directory of the project
- Adds project's ebin directories to Erlang VM's search path
Running Tests

$ mix test

- Runs tests in the tests directory
Building a Release

```elixir
defp deps do
  [ {:exrm, "~> 1.0.3"} ]
end
```

- Adds Elixir Release Manager as a dependency
- Uses hex to locate repository
- Version follows SemVer 2.0 schema
- `~>` allows for incremental releases
  - same as `>= 1.0.3` and `< 1.0.4`
  - Notation documented with Elixir Version module
Building a dev Release

$ mix compile
$ mix release --dev

- release is in rel/app_name
- symbolic links back to source tree
Building a Production Release

$ MIX_ENV=prod mix compile
$ MIX_ENV=prod mix release

• rel/app_name is a self contained directory for the release
• Copy to target host and run
• Does not work with umbrella applications
Release Tree

- `rel/`
  - `app_name/`
    - `bin/`
      - `erts-7.2.1/`
    - `lib/`
      - `app-vsn/`
        - `ebin/`
          - `include/`
        - `priv/`
      - `releases/`
        - `RELEASEES`
    - `0.0.1/`

- `bin` – app_name control script
- `erts-vsn` – Erlang VM
- `lib/app-vsn` – compiled applications tagged with application version
- `RELEASES` – .rel file
- `0.0.1` – release version
  - start scripts
  - node and app config files
Running the Release

$ rel/apple_name/bin/app_name console

[...]  
Erlang/OTP 18 [erts-7.2.1] ...  
Interactive Elixir (1.2.1) ...  
iex(app_name@MacBook-Pro)1>

- rel/app_name/bin/app_name is the control script
- console – start with an interactive console
- start – start headless
- stop – stop release
- attach – attach a shell to a running release
- ping – check if running
sys.config, vm.args, relx.config

rel/
  sys.config
  vm.args
  relx.config

- Optional configuration files
- sys.config – Erlang term file with application environment variables. Overrides environments provided by applications.
- vm.args – command line arguments used when starting Erlang Beam
- relx.config – custom relx configuration
Getting Help

$ mix help
$ mix help command

- mix has built in help text
- online tutorial
- exrm
  - https://exrm.readme.io/