Extracted from:

Programming Elixir

Functional |> Concurrent |> Pragmatic |> Fun

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Programming Elixir

Functional
|> Concurrent
|> Pragmatic
|> Fun

Dave Thomas

Foreword by
José Valim,
Creator of Elixir

edited by Lynn Beighley

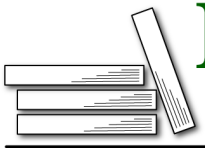
Programming Elixir

Functional |> Concurrent |> Pragmatic |> Fun

Dave Thomas

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

Lynn Beighley (editor)
Potomac Indexing, LLC (indexer)
Candace Cunningham (copyeditor)
Janet Furlow (producer)
Ellie Callahan (support)

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-937785-58-1
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—October, 2014

CHAPTER 15

Nodes—The Key to Distributing Services

There's nothing mysterious about a *node*. It is simply a running Erlang VM. Throughout this book we've been running our code on a node.

The Erlang VM, called *Beam*, is more than a simple interpreter. It's like its own little operating system running on top of your host operating system. It handles its own events, process scheduling, memory, naming services, and interprocess communication. In addition to all that, a node can connect to other nodes—in the same computer, across a LAN, or across the Internet—and provide many of the same services across these connections that it provides to the processes it hosts locally.

Naming Nodes

So far we haven't needed to give our node a name—we've had only one. If we ask Elixir what the current node is called, it'll give us a made-up name:

```
iex> Node.self
:nonode@nohost
```

We can set the name of a node when we start it. With iex, use either the `--name` or `--sname` option. The former sets a fully qualified name:

```
$ iex --name wibble@light-boy.local
iex(wibble@light-boy.local)> Node.self
:"wibble@light-boy.local"
```

The latter sets a short name.

The name that's returned is an atom—it's in quotes because it contains characters not allowed in a literal atom.

```
$ iex --sname wobble
iex(wobble@light-boy)> Node.self
:"wobble@light-boy"
```

Note that in both cases the iex prompt contains the node's name along with my machine's name (light-boy).

Now I want to show you what happens when we have two nodes running. The easiest way to do this is to open two terminal windows and run a node in each. To represent these windows in the book, I'll show them stacked vertically.

Let's run a node called node_one in the top window and node_two in the bottom one. We'll then use the Elixir Node module's list function to display a list of known nodes, then connect from one to the other.

Window #1

```
$ iex --sname node_one
iex(node_one@light-boy)>
```

Window #2

```
$ iex --sname node_two
iex(node_two@light-boy)> Node.list
[]
iex(node_two@light-boy)> Node.connect :"node_one@light-boy"
true
iex(node_two@light-boy)> Node.list
[:"node_one@light-boy"]
```

Initially, node_two doesn't know about any other nodes. But after we connect to node_one (notice that we pass an atom containing that node's name), the list shows the other node. And if we go back to node one, it will now know about node two.

```
iex(node_one@light-boy)> Node.list
[:"node_two@light-boy"]
```

Now that we have two nodes, we can try running some code. On node one, let's create an anonymous function that outputs the current node name.

```
iex(node_one@light-boy)> func = fn -> IO.inspect Node.self end
#Function<erl_eval.20.82930912>
```

We can run this with the spawn function.

```
iex(node_one@light-boy)> spawn(func)
#PID<0.59.0>
node_one@light-boy
```

But spawn also lets us specify a node name. The process will be spawned on that node.

```
iex(node_one@light-boy)> Node.spawn(:"node_one@light-boy", func)
#PID<0.57.0>
node_one@light-boy
iex(node_one@light-boy)> Node.spawn(:"node_two@light-boy", func)
#PID<7393.48.0>
node_two@light-boy
```

We're running on node one. When we tell spawn to run on node_one@light-boy, we see two lines of output. The first is the PID spawn returns, and the second line is the value of Node.self that the function writes.

The second spawn is where it gets interesting. We pass it the name of node two and the same function we used the first time. Again we get two lines of output. The first is the PID and the second is the node name. Notice the PID's contents. The first field in a PID is the node number. When running on a local node, it's zero. But here we're running on a remote node, so that field has a positive value (7393). Then look at the function's output. It reports that it is running on node two. I think that's pretty cool.

You may have been expecting the output from the second spawn to appear in the lower window. After all, the code runs on node two. But it was created on node one, so it inherits its process hierarchy from node one. Part of that hierarchy is something called the *group leader*, which (among other things) determines where IO.puts sends its output. So in a way, what we're seeing is doubly impressive. We start on node one, run a process on node two, and when the process outputs something, it appears back on node one.

Your Turn

➤ *Exercise: Nodes-1*

Set up two terminal windows, and go to a different directory in each. Then start up a named node in each. In one window, write a function that lists the contents of the current directory.

```
fun = fn -> IO.puts(Enum.join(File.ls!, ",")) end
```

Run it twice, once on each node.

Nodes, Cookies, and Security

Although this is cool, it might also ring some alarm bells. If you can run arbitrary code on any node, then anyone with a publicly accessible node has just handed over his machine to any random hacker.

But that's not the case. Before a node will let another connect, it checks that the remote node has permission. It does that by comparing that node's *cookie*

with its own cookie. A cookie is just an arbitrary string (ideally fairly long and very random). As an administrator of a distributed Elixir system, you need to create a cookie and then make sure all nodes use it.

If you are running the iex or elixir commands, you can pass in the cookie using the --cookie option.

```
$ iex --sname one --cookie chocolate-chip
iex(one@light-boy)> Node.get_cookie
:"chocolate-chip"
```

If we repeat our two-node experiment and explicitly set the cookie names to be different, what happens?

Window #1

```
$ iex --sname node_one --cookie cookie-one
iex(node_one@light-boy)> Node.connect :"node_two@light-boy"
false
```

Window #2

```
$ iex --sname node_two --cookie cookie-two
iex(node_two@light-boy)>
=ERROR REPORT==== 27-Apr-2013::21:27:43 ===
** Connection attempt from disallowed node 'node_one@light-boy' **
```

The node that attempts to connect receives false, indicating the connection was not made. And the node that it tried to connect to logs an error describing the attempt.

But why does it succeed when we don't specify a cookie? When Erlang starts, it looks for an .erlang.cookie file in your home directory. If that file doesn't exist, Erlang creates it and stores a random string in it. It uses that string as the cookie for any node the user starts. That way, all nodes you start on a particular machine are automatically given access to each other.

Be careful when connecting nodes over a public network—the cookie is transmitted in plain text.

Naming Your Processes

Although a PID is displayed as three numbers, it contains just two fields; the first number is the node ID and the next two numbers are the low and high bits of the process ID. When you run a process on your current node, its node ID will always be zero. However, when you export a PID to another node, the node ID is set to the number of the node on which the process lives.

That works well once a system is up and running and everything is knitted together. If you want to register a callback process on one node and an event-generating process on another, just give the callback PID to the generator.

But how can the callback find the generator in the first place? One way is for the generator to register its PID, giving it a name. The callback on the other node can look up the generator by name, using the PID that comes back to send messages to it.

Here's an example. Let's write a simple server that sends a notification about every 2 seconds. To receive the notification, a client has to register with the server. And we'll arrange things so that clients on different nodes can register.

While we're at it, we'll do a little packaging so that to start the server you run Ticker.start, and to start the client you run Client.start. We'll also add an API Ticker.register to register a client with the server.

Here's the server code:

nodes/ticker.ex

```
defmodule Ticker do

  @interval 2000   # 2 seconds
  @name     :ticker

  def start do
    pid = spawn(__MODULE__, :generator, [[]])
    :global.register_name(@name, pid)
  end

  def register(client_pid) do
    send :global.whereis_name(@name), { :register, client_pid }
  end

  def generator(clients) do
    receive do
      { :register, pid } ->
        IO.puts "registering #{inspect pid}"
        generator([pid|clients])
    after
      @interval ->
        IO.puts "tick"
        Enum.each clients, fn client ->
          send client, { :tick }
        end
        generator(clients)
    end
  end
end
```

We define a start function that spawns the server process. It then uses :global.register_name to register the PID of this server under the name :ticker.

Clients who want to register to receive ticks call the register function. This function sends a message to the Ticker server, asking it to add those clients to its list. Clients could have done this directly by sending the :register message to the server process. Instead, we give them an interface function that hides the registration details. This helps decouple the client from the server and gives us more flexibility to change things in the future.

Before we look at the actual tick process, let's stop to consider the start and register functions. These are not part of the tick process—they are simply chunks of code in the Ticker module. This means they can be called directly wherever we have the module loaded—no message passing required. This is a common pattern; we have a module that is responsible both for spawning a process and for providing the external interface to that process.

Back to the code. The last function, generator, is the spawned process. It waits for two events. When it gets a tuple containing :register and a PID, it adds the PID to the list of clients and recurses. Alternatively, it may time out after 2 seconds, in which case it sends a {:tick} message to all registered clients.

(This code has no error handling and no means of terminating the process. I just wanted to illustrate passing PIDs and messages between nodes.)

The client code is simple:

nodes/ticker.ex

```
defmodule Client do

  def start do
    pid = spawn(__MODULE__, :receiver, [])
    Ticker.register(pid)
  end

  def receiver do
    receive do
      { :tick } ->
        IO.puts "tock in client"
        receiver
    end
  end
end
```

It spawns a receiver to handle the incoming ticks, and passes the receiver's PID to the server as an argument to the register function. Again, it's worth noting that this function call is local—it runs on the same node as the client.

However, inside the Ticker.register function, it locates the node containing the server and sends it a message. As our client's PID is sent to the server, it becomes an external PID, pointing back to the client's node.

The spawned client process simply loops, writing a cheery message to the console whenever it receives a tick message.

Let's run it. We'll start up our two nodes. We'll call Ticker.start on node one. Then we'll call Client.start on both node one and node two.

Window #1

```
nodes % iex --sname one
iex(one@light-boy)> c("ticker.ex")
[Client,Ticker]
iex(one@light-boy)> Node.connect :"two@light-boy"
true
iex(one@light-boy)> Ticker.start
:yes
tick
tick
iex(one@light-boy)> Client.start
registering #PID<0.59.0>
{:register,#PID<0.59.0>}
tick
tock in client
tick
tock in client
tick
tock in client
tick
tock in client
 :    :   :
```

Window #2

```
nodes % iex --sname two
iex(two@light-boy)> c("ticker.ex")
[Client,Ticker]
iex(two@light-boy)> Client.start
{:register,#PID<0.53.0>}
tock in client
tock in client
tock in client
 :    :   :
```

To stop this, you'll need to exit iex on both nodes.

When to Name Processes

When you name something, you are recording some global state. And as we all know, global state can be troublesome. What if two processes try to register the same name, for example?

The runtime has some tricks to help us. In particular, we can list the names our application will register in the app's mix.exs file. (We'll cover how when we look at packaging an application on page ?.) However, the general rule is to register your process names when your application starts.

Your Turn

➤ *Exercise: Nodes-2*

When I introduced the interval server, I said it sent a tick "about every 2 seconds." But in the receive loop, it has an explicit timeout of 2,000 ms. Why did I say "about" when it looks as if the time should be pretty accurate?

➤ *Exercise: Nodes-3*

Alter the code so that successive ticks are sent to each registered client (so the first goes to the first client, the second to the next client, and so on). Once the last client receives a tick, the process starts back at the first. The solution should deal with new clients being added at any time.

I/O, PIDs, and Nodes

Input and output in the Erlang VM are performed using I/O servers. These are simply Erlang processes that implement a low-level message interface. You never have to deal with this interface directly (which is a good thing, as it is complex). Instead, you use the various Elixir and Erlang I/O libraries and let them do the heavy lifting.

In Elixir you identify an open file or device by the PID of its I/O server. And these PIDs behave just like all other PIDs—you can, for example, send them between nodes.

If you look at the implementation of Elixir's IO.puts function, you'll see

```
def puts(device \\ group_leader(), item) do
    erl_dev = map_dev(device)
    :io.put_chars erl_dev, [to_iodata(item), ?\n]
end
```

(To see the source of an Elixir library module, view the online documentation at http://elixir-lang.org/docs/, navigate to the function in question, and click the *Source* link.)

The default device it uses is returned by the function :erlang.group_leader. (The group_leader function is imported from the :erlang module at the top of the IO module.) This will be the PID of an I/O server.

So, bring up two terminal windows and start a different named node in each. Connect to node one from node two, and register the PID returned by group_leader under a global name (we use :two).

Window #1

```
$ iex --sname one
iex(one@light-boy) >
```

Window #2

```
$ iex --sname two
iex(two@light-boy) > Node.connect(:"one@light-boy")
true
iex(two@light-boy) > :global.register_name(:two, :erlang.group_leader)
:yes
```

Note that once we've registered the PID, we can access it from the other node. And once we've done that, we can pass it to IO.puts; the output appears in the other terminal window.

Window #1

```
iex(one@light-boy) > two = :global.whereis_name :two
#PID<7419.30.0>
iex(one@light-boy) > IO.puts(two, "Hello")
:ok
iex(one@light-boy) > IO.puts(two, "World!")
:ok
```

Window #2

```
Hello
World
iex(two@light-boy) >
```

Your Turn

➤ *Exercise: Nodes-4*

The ticker process in this chapter is a central server that sends events to registered clients. Reimplement this as a ring of clients. A client sends a tick to the next client in the ring. After 2 seconds, *that* client sends a tick to *its* next client.

When thinking about how to add clients to the ring, remember to deal with the case where a client's receive loop times out just as you're adding a new process. What does this say about who has to be responsible for updating the links?

Nodes Are the Basis of Distribution

We've seen how we can create and interlink a number of Erlang virtual machines, potentially communicating across a network. This is important, both to allow your application to scale and to increase reliability. Running all your code on one machine is like having all your eggs in one basket. Unless you're writing a mobile omelet app, this is probably not a good idea.

It's easy to write concurrent applications with Elixir. But writing code that follows the happy path is a lot easier than writing bullet-proof, scalable, and hot-swappable world-beating apps. For that, you're going to need some help.

In the worlds of Elixir and Erlang, that help is called OTP, and it is the subject of the next few chapters.