Extracted from:

From Ruby to Elixir

Unleash the Full Potential of Functional Programming

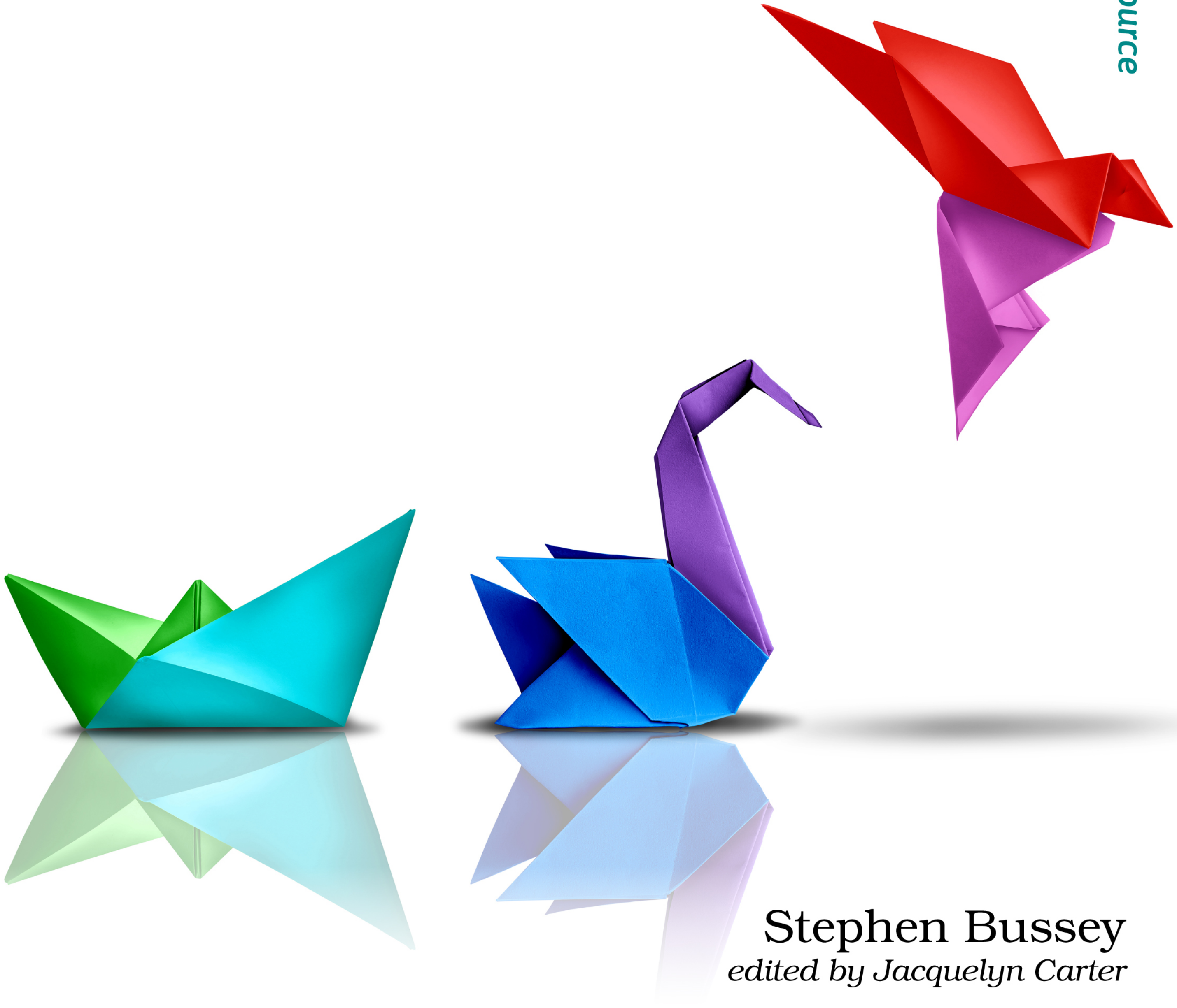
The Pragmatic Bookshelf
Raleigh, North Carolina

The Pragmatic Programmers

Your Elixir Source

From Ruby to Elixir

Unleash the Full Potential of Functional Programming



Stephen Bussey
edited by Jacquelyn Carter

From Ruby to Elixir

Unleash the Full Potential of Functional Programming

Stephen Bussey

The Pragmatic Bookshelf
Raleigh, North Carolina

For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

For sales, volume licensing, and support, please contact *support@pragprog.com*.

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

ISBN-13: 979-8-88865-031-8
Encoded using the finest acid-free high-entropy binary digits.
Book version: B1.0—May 31, 2023

Explore Elixir Processes

Processes are the foundation of concurrency in Elixir. They are small, easy to spawn, and you can run as many of them as you have memory for—in production tens to hundreds of thousands would be normal. You don't need to know much about the BEAM's process architecture to use processes, but the details really highlight how powerful they are.

We're going to cover the basics of processes in this section, but we'll also cover some of the interesting details of the process architecture. You'll learn how to spawn a process and pass messages to it, and you'll make an infinitely-running process that responds to incoming messages. We'll also go over error isolation, memory isolation, and garbage collection.

Spawn a Process

Elixir makes it very easy to start a new process. The spawn/1 functions takes a function and executes it inside of a new process. Let's do that in IEx:

```
iex> self()
#PID<0.109.0>

iex> pid = spawn(fn -> IO.puts("Hello from #{inspect self()}") end)
Hello from #PID<0.112.0>
#PID<0.112.0>

iex> Process.alive?(pid)
false
```

We pass a function into spawn/1 that prints out some information about the executing process. Your exact numbers will be different than mine, but notice that the spawned function prints out as 0.112.0 but the originating process is 0.109.0. This is called a process ID (PID) and is one of the core data types in Elixir. The difference in PIDs proves that the spawned function is actually executing inside of a different process.

This spawned process would be useful if we wanted to fire off some asynchronous code, but it's not really that useful right now. We need to be able to send messages into the process and receive responses from it in order to turn it into a useful tool. The receive function lets us do just that. And in order to send a message to the process, we'll use the send function.

Type this code into your IEx session:

```
iex> pid = spawn(fn ->
  receive do
    :hello -> IO.puts("Hello World")
    {:hello, name} -> IO.puts("Hello #{name}")
  end
end)

iex> Process.alive?(pid)
true

iex> send(pid, :hello)
Hello World
:hello

iex> Process.alive?(pid)
false
```

We were able to process the message :hello and see that the correct output was printed. If you try the example again with send(pid, {:hello, "Your Name"}), you will see that it responds with a different message. The receive function uses pattern matching to determine which code to run, just like you're already familiar with.

We aren't going to do the exercise here, but if you wanted to receive a response from the spawned server, you would send back to the originating process. This requires you to pass the PID of the current process as part of the message, and then to receive a response. That's pretty cumbersome, but you'll see how GenServer makes this very easy in the next main section.

Our process is no longer alive after a single message. Let's make it process messages forever.

Process Messages Forever

Recursion is very useful to create infinite loops. Usually an infinite loop would be a bad thing, but it's totally fine when used in a controlled way.

Create lib/examples/spawn/infinite.ex and add the following code:

elixir_examples/lib/examples/spawn/infinite.ex

```
defmodule Examples.Spawn.Infinite do
  def start do
    spawn(& loop/0)
  end

  defp loop do
    receive do
      {:add, a, b} ->
        IO.puts(a + b)
➤       loop()

      :memory ->
        {:memory, bytes} = Process.info(self(), :memory)
        IO.puts("I am using #{bytes} bytes")
➤       loop()
```

```
      :crash ->
        raise "I crashed"

      :bye ->
        IO.puts("Goodbye")
    end
  end
end
```

The start/0 function uses spawn/1 to kick off our looped process. The loop function is very simple, it is just a receive block with a variety of messages handled. For all messages except :bye, the loop/0 function is called as the last thing the function does. This creates a recursive loop that will handle messages forever. Let's try it out:

```
$ iex -S mix
iex> pid = Examples.Spawn.Infinite.start()

iex> send(pid, :memory)
I am using 2608 bytes

iex> send(pid, {:add, 1, 2})
3

iex> send(pid, {:add, 50, 50})
100
```

You could extend this exercise by turning loop/0 into loop/1 and keeping tracking of state for each message. If you did this, you would have a server that changes state based on messages it has received from the outside world. This is a pretty small change, but is still pretty cumbersome. Don't worry, GenServer will make this easy for us too.

Before we get into GenServer, let's look at some of the interesting details of how processes are implemented. These may seem unimportant at first, but they drastically shape the runtime characteristics of an Elixir application.

Error Isolation in Processes

There's an argument to be made that the genius of the BEAM is not its parallel execution ability, but rather its ability to isolate errors. Let's put that into perspective: if two requests come into a web server at the same time, and one of the requests crashes, then we would not expect the other request to also crash.

Let's spawn two processes and then crash one of them to create a basic demonstration:

```
$ iex -S mix
iex> p1 = Examples.Spawn.Infinite.start()
```

```
#PID<0.161.0>

iex> p2 = Examples.Spawn.Infinite.start()
#PID<0.163.0>

iex> send(p1, :crash)
[error] Process #PID<0.161.0> raised an exception
** (RuntimeError) I crashed

iex> [Process.alive?(p1), Process.alive?(p2)]
[false, true]
```

This is a simple example, but it serves to demonstrate that we did not have to do anything to isolate this error. In fact, it would not be possible to crash one of these processes from the other. Of course, an event like a database failure is going to cause errors all over an application, but that would be due to an external factor rather than an internal one.

It's easy to take error isolation for granted. Clever programming from frameworks in languages that don't provide error isolation will make it feel like there's isolation, but a guarantee from the virtual machine runtime is another level of confidence.

Process Memory Architecture

Each process in Elixir has its own memory space. This consists of a heap and a stack that grow towards each other. Eventually, if they are unable to grow, the BEAM will allocate more memory to the process.

Processes start off with a fairly small amount of memory. On my computer, I see 2608 bytes taken up for a brand new process:

```
$ iex -S mix
iex> pid = Examples.Spawn.Infinite.start()
iex> Process.info(pid, :memory)
{:memory, 2608}
```

Process.info(pid) is a very useful source of information about any process that is actively running. It tells you things like heap size, stack size, reductions (which roughly equate to CPU usage), and the number of unprocessed messages. Here, we used Process.info/2 to return a focused version of the available data.

Data in Elixir is copied between processes. So if you send a message to a process, that memory will be duplicated and then passed as a message. This has benefits for small bits of data, but it would be a bit of waste to copy every single message between processes. Elixir has a little trick of its sleeve to optimize copies.

Elixir uses a binary heap[3] to globally store large (> 64 bytes) binary data. This binary heap is shared between processes and uses reference counting to determine when the memory can be cleaned up. Because the BEAM uses immutable data, you don't need to worry about this causing bugs in your application. The memory here is safe to use and can be referenced by multiple processes without fear.

The small memory size of processes is part of what makes them very easy to spawn and destroy. But, sometimes you will find yourself debugging a problem where too much memory is being used. So, let's cover how garbage collection works.

Garbage Collection

Garbage collection isn't fun, right? Actually, the BEAM's garbage collector is quite interesting. I wrote about this in fairly deep detail in *Real-Time Phoenix [Bus20]*, but we'll cover much less in this book.

Because each process in Elixir has its own memory heap and stack, each process performs its own garbage collection. The binary heap that was mentioned in the last section is globally shared, so there is a global collection process to handle it. However, it's relatively lightweight because the binary heap uses reference-counted binaries.

Each garbage collection process runs fast because it deals with less memory, and it happens on a cycle according to how much that process is churning data. If the process is very active, or is running out of memory, then the process will experience a collection cycle more often than a process that isn't doing much.

There is a hidden danger that you have to be cautious of, though. Long-lived processes can run into a situation where the amount of memory they are taking up is more than they need, but they won't undergo a garbage collection cycle because they aren't active enough to kick one off. In this situation, a process can take up more memory than it needs for a long period of time. Let's create an artificial example:

```
$ iex -S mix
iex> pid = Examples.Spawn.Infinite.start()
iex> Enum.each(1..1000, & send(pid, {:add, &1, &1}))

iex> send(pid, :memory)
I am using 62816 bytes
```

3. https://www.erlang.org/doc/apps/erts/garbagecollection#binary-heap

Your numbers may vary here, but you should expect to see that this number is higher than the 2.6 KB that it started out as. This doesn't immediately make sense because our process has no state, and it has processed all of its messages. So, why is it taking up thirty times more memory?

The issue here is that the process mailbox lives on the heap of the process. As we inundated it with messages, it had to allocate more memory to hold those messages. The garbage collection process only occurs based on running out of memory or processing a given number of reductions—neither of which are occurring.

We can manually trigger garbage collection with :erlang.garbage_collect(pid). Once you do this and query the process memory, you will see that it's back to its starting size.

```
iex> :erlang.garbage_collect(pid)
iex> send(pid, :memory)
I am using 2608 bytes
```

The number of long-lived processes is usually small enough that this doesn't matter. But, if it becomes a problem, then look at the ERL_FULLSWEEP_AFTER system variable and set it to a number like 20. This causes garbage collection to run more frequently—at the cost of a bit more CPU. This flag is enabled on every single production system I've worked on, and it has never caused problems for me.

There's another option to prevent memory bloat. You can put individual processes into a hibernation state. A hibernating process has its memory reduced as much as possible. But when the process receives a message, it will incur a cost to exit the hibernation state and handle the message. GenServer[4] has a hibernate_after option that will automatically enter hibernation when the GenServer is idle.

Both techniques are important to know about, but you likely won't need to use them for some time. Frameworks like Phoenix use hibernation with sane defaults so that you often don't need to think about it.

Now that you have the basics of processes down, let's take a look at how Elixir makes them easy with GenServer.

4. https://hexdocs.pm/elixir/GenServer.html