Extracted from:

# Programmer Passport: Elixir



The Pragmatic Bookshelf

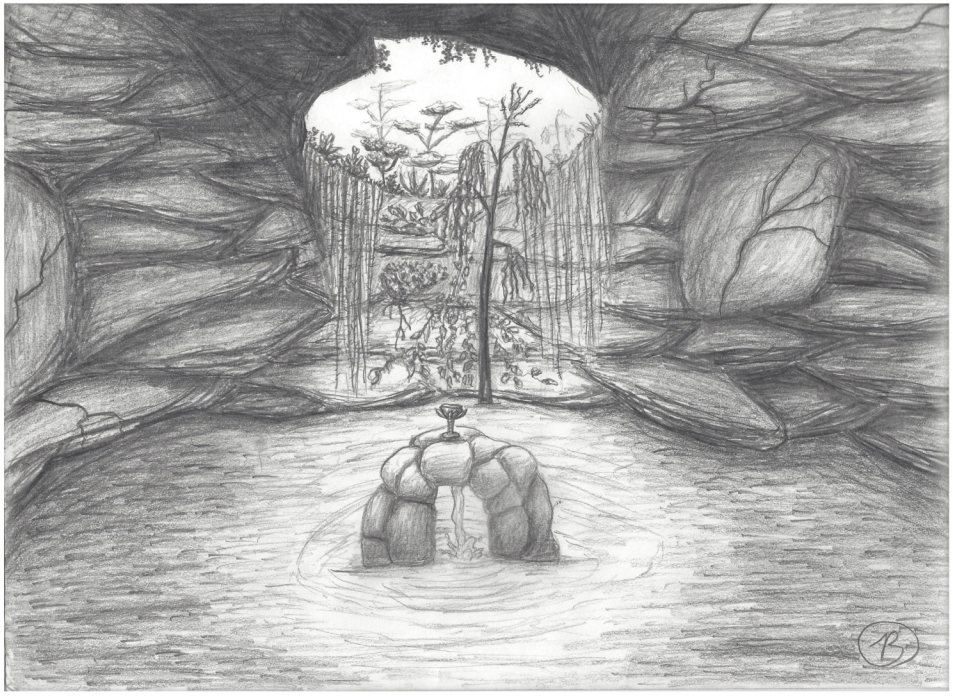
Raleigh, North Carolina

The Pragmatic Programmers

grox.io
Learning

# Programmer Passport

## Elixir



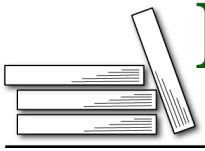
Bruce A. Tate

*Edited by Jacquelyn Carter*

# Programmer Passport: Elixir

Bruce Tate

The Pragmatic Bookshelf
Raleigh, North Carolina



Pragmatic Bookshelf



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

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Jacquelyn Carter
Copy Editor: Corina Lebegioara
Founders: Andy Hunt and Dave Thomas

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

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



ISBN-13: 978-1-68050-962-5
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—May 2022

Though you might not know it yet, every Elixir program runs in a process. So far, we've not had to write any code to start our own process. In this chapter, we'll change all of that. We'll look at the functions and primitives for spawning processes, sending messages, and building connections between them.

Let's plan our attack. First, we're going to walk through what Elixir processes are and how you can work with them. We'll explore them first in IEx.

Next, we'll use processes to build a key-value store using primitives. We'll talk about the pros and cons of our approach and also concepts like message passing and back pressure.

Finally, we'll give you the chance to try some of these concepts yourself. The chapter will be fairly intense.

Let's get started.

## Processes, Inboxes, and Pattern Matching

As you find in other languages, *processes* execute programs. Elixir processes are completely isolated from one another. Processes run concurrently to one another using *preemptive multitasking*. That means that Elixir is responsible for dividing up work, and processes don't share memory with one another. All communication between processes uses message passing.

Elixir processes aren't the same as operating system processes. They're extremely lightweight in terms of resources, including memory. Elixir *processes* are even more lightweight than operating system *threads*. It's common to have hundreds of thousands of them running concurrently.

The techniques and data structures you've seen so far might make some sense to you. When they use message passing and pattern matching together with tuples, many developers find that Elixir begins to resonate in earnest because working with processes in Elixir is often much easier than in other languages.

Processes and message passing are the primitives that serve to build the foundations of Elixir's OTP system, the library that Elixir uses to abstract concurrency, application lifecycles, message passing, supervision, and failover. Because it's such a central concept in other functional languages like Erlang and Scala, Programmer Passport will focus an entire book on OTP. This chapter will provide many of the primitives Elixir builds on for OTP.

Let's start to play with processes.

## Processes Have Pids and Message Queues

Regardless of whether an Elixir program is running in a test script, IEx, or mix task, it's running in a process. Let's open up IEx and start to play. Let's find the IEx process ID, like this:

```
iex> iex = self()
iex = #PID<0.103.0>
```

All processes are identified by a process ID, called a *pid*. Let's get the info for our pid:

```
iex> i
Term
  #PID<0.103.0>
Data type
  PID
Alive
  true
Name
  not registered
Links
  none
Message queue length
  0
Description
  Use Process.info/1 to get more info about this process
Reference modules
  Process, Node
Implemented protocols
  IEx.Info, Inspect
```

This list is interesting. In addition to the usual information, you can see a few extra bits of information. For example, the Alive section tells us our process is in fact alive.

The modules for working with a process are Process for processes on one system and Node for working with *distributed systems*. A distributed system divides a single system into slices, potentially across different networked computers.

Notice also the message queue length. Think of the message queue as a mailbox. We can send messages to our process, like this:

```
iex> send iex, :hello
:hello
iex> send iex, {:is, :anyone, :home}
{:is, :anyone, :home}
iex> i iex
Term
  #PID<0.103.0>
```

```
...
Message queue length
  2
...
```

Nice! We've sent a message to ourselves, so there are two messages in our own mailbox. Let's use the API to get information about this process:

```
iex> Process.info iex
[
  current_function: {Process, :info, 1},
  status: :running,
  message_queue_len: 2,
  ...
]
```

I've shortened this list, but you get the idea. Our process is running, and it has a message queue length of 2. In IEx, you can issue the command flush to empty the queue and print all of the messages, like this:

```
iex(14)> flush
:hello
{:is, :anyone, :home}
```

And we see the two messages we sent previously!

Typically, you don't receive messages with flush/0. That function is in the IEx.Helpers module. Let's look at a more realistic example.

## Get Matching Messages with receive

The receive control structure lets us receive a message matching a pattern. Let's build a function to send a message to IEx. A common pattern in Elixir is to shape messages into tuples. These messages often have some type of atom to match against, some kind of numeric code, and a string with an English message so we can easily tell what the message means. Let's build a function called deliver to send such a message:

```
iex> deliver = fn i -> send self(), {:message, i, "Message#{i}"} end
#Function<6.128620087/1 in :erl_eval.expr/5>
iex> Enum.each((1..6), deliver)
:ok
iex> Process.info(self())[:message_queue_len]
6
```

We build the deliver function to send a message in a three-tuple based on some integer to self(). The message is a three-tuple. Then, we deliver messages to each of the six integers in the range from 1 to 6. To make sure the messages got sent, we check the :message_queue_len using Process.info/1.

We already know how to flush all of the messages. Let's receive a message:

```
iex> receive do
...>   message ->
...>     IO.inspect(message)
...> end
{:message, 1, "Message1"}
{:message, 1, "Message1"}
```

We ask Elixir to receive a message and give it the message pattern to match. Elixir will return the first message in the queue that matches the pattern. We match the first message in the queue, print it out, and return it.

Let's look at receive in more detail. The expression we'll use is the receive message. It takes the form:

```
receive do
  pattern ->
    expression
  after
    timeout_in_milliseconds ->
      timeout_expression
end
```

Let's try another one. Let's match an explicit message, like this:

```
iex> receive do
...>   {:message, 4, text} ->
...>     text
...> end
"Message4"
iex> Process.info(self())[:message_queue_len]
4
```

We ask Elixir to match the message that matches the tuple {:message, 4, text}. Elixir skips the next two messages until it reaches the tuple {:message, 4, "Message4"} and binds the text variable to Message4. Four messages remain: 2, 3, 5, and 6. We've not received them yet.

Let's see how the timeout works.

```
iex> receive do
...>   {:message, 4, text} ->
...>     text
...> after
...>   1000 ->
...>     {:error, :no_message}
...> end
{:error, :no_message}
```

We ask for message 4. Finding none, Elixir waits one thousand milliseconds, or one second, and returns the value we provide. Using receive, we can match any pattern we choose. If there's no message yet in the message queue, Elixir will wait until there is a message. If you specify an optional after block, Elixir will raise a timeout error if no message is received.

receive is the way all processes interact with each other.

There's still an important tool you haven't seen yet. The next step in working with processes is starting them.