ELIXIR

SUCCINCTLY

BY EMANUELE DELBONO

SUCCINCTLY EBOOK SERIES

Syncfusion

www.dbooks.org

Elixir Succinctly

By

Emanuele DelBono

Foreword by Daniel Jebaraj



Copyright © 2019 by Syncfusion, Inc.

2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA
All rights reserved.

Important licensing information. Please read.

This book is available for free download from www.syncfusion.com on completion of a registration form.

If you obtained this book from any other source, please register and download a free copy from www.syncfusion.com.

This book is licensed for reading only if obtained from www.syncfusion.com.

This book is licensed strictly for personal or educational use.

Redistribution in any form is prohibited.

The authors and copyright holders provide absolutely no warranty for any information provided.

The authors and copyright holders shall not be liable for any claim, damages, or any other liability arising from, out of, or in connection with the information in this book.

Please do not use this book if the listed terms are unacceptable.

Use shall constitute acceptance of the terms listed.

SYNCFUSION, SUCCINCTLY, DELIVER INNOVATION WITH EASE, ESSENTIAL, and .NET ESSENTIALS are the registered trademarks of Syncfusion, Inc.

Technical Reviewer: James McCaffrey

Copy Editor: Courtney Wright

Acquisitions Coordinator: Tres Watkins, content development manager, Syncfusion, Inc.

Proofreader: Darren West, content producer, Syncfusion, Inc.

www.dbooks.org

Table of Contents

The Story Behind the *Succinctly* Series of Books **6**

About the Author **8**

Introduction **9**

A short history of Elixir 9

Why is Elixir so interesting today? 11

Chapter 1 The Language **13**

Basic types 13

Modules and functions 16

Import 18

Pattern matching 19

Recursion 22

Tail-call optimization 22

Helpful modules 23

List and Enum 23

Map 26

Control flow 28

Guards 29

Pipe operator 30

Type specifications 30

Behavior and protocols 33

Protocols 33

Behaviors 35

Macros 36

Chapter 2 The Platform **39**

Elixir applications....44
Mix.exs....46
Sample_app.ex....47
Sample_app_test.exs....47
GenServer....48
Supervisors....52
Observer....55
Chapter 3 Sample Application....62
The first test....63
Control the rover....67
Rover supervisor....70
Introducing Plug and Cowboy....73
Conclusion....84

www.dbooks.org

The Story Behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Face-book to help us spread the word about the *Succinctly* series!





www.dbooks.org

About the Author

Emanuele DelBono (@emadb) is a web developer based in Italy. He is one of the founders of CodicePlastico, a software house that builds web applications for its customers. He architects and develops web applications in Node.js and .NET.

Emanuele is also a speaker at various conferences about web development and agile practices. He plays an active role in Italian development communities such as Webdebs.org.

Introduction

A short history of Elixir

Elixir is a relatively new functional programming language that runs on the Erlang virtual machine. For a look into the history of this young language, we have to start in the 1980s.

It was 1982 when Ericsson, the leader in telecommunication systems at the time, was looking for a new programming language for its telephone switches and routers. Ericsson tried various languages, but none completely answered the requirements of availability, robustness, and concurrency.

The company decided to implement their own languages, and in 1986, they began developing Erlang, using Prolog. Over the next few years they developed a series of prototypes that would eventually become the foundation of what Erlang is today.

In 1990, the first version of Erlang (rewritten in C++) was used in a real project. It was a mobility server for Digital Enhanced Cordless Telecommunication (DECT) used in private offices. The project was successful, and was used to gather feedback for the next version of Erlang. This is when the Open Telecom Platform (OTP) framework was developed.

At that time, the core team was composed of three developers: Joe Armstrong, Robert Virding, and Mike William.

Erlang was used a lot in Ericsson until 1999, when they decided to ban it inside the company in favor of a more popular language. This decision was in some ways the trigger that permitted Erlang to reach outside of Ericsson and become a commonly used language.

Erlang became open source, and its peculiarities were suddenly appreciated. To understand why Erlang was (and still is) very appreciated, we have to remember that in 1998, Ericsson announced the AXD301 switch, which contains over a million lines of Erlang code, and was reported to achieve the nine "9"s availability, meaning that it was "down" for only 0.63 seconds in 20 years!

How could such level of availability be achieved?

Joe Armstrong talked a lot in his speeches about robustness, and he explained why Ericsson had to write its own language. The main driver was concurrency: if a language is not designed to cope with concurrency from the beginning, it's very difficult to add it later; whatever developers try to do it will create problems or be sub-optimal.

Erlang was designed with concurrency at its core: processes are truly independent, there are no penalties for massive concurrency, and distribution and concurrent behavior work the same on all operating systems.

With these principles, Erlang developed the maturity and stability it has today. These peculiarities are exactly what today's developers need to build applications with high availability and scalability.

www.dbooks.org

And here, Elixir enters the scene.

One problem with Erlang is the syntax of the language, which is very different from other programming languages we are using today. This is probably one of the factors that pushed Jose Valim, a Ruby developer, to investigate Erlang for designing Elixir.

In 2012, Jose Valim was working on a research project for Plataformatec, trying to improve the performance of Ruby on Rails in multi-core systems. However, Ruby was not designed to take full advantage of multicore systems. So Valim took a look at other platforms to better understand how they solved the concurrency problem, and eventually, he fell in love with the Erlang Virtual Machine.

As soon as Valim started using Erlang, he noticed that the language was missing some constructs available in other languages, and that the ecosystem wasn't very rich like Ruby or Nodejs (https://nodejs.org). That's when he decided to create Elixir.

The point was to design a modern functional programming language that runs on a battle-tested solid platform: The Erlang Virtual Machine.

Elixir has grown a lot in the past several years—we have version 1.8, a package manager, lots of documentation, and a growing community.

The core team of Elixir releases a new version about every six months. There is a mailing list where you can read about what decisions are being evaluated, and you can contribute with new ideas and proposals (and pull requests as well).

We said before that Erlang lacked a package manager. Elixir introduced Hex, a package manager that works for both Elixir and Erlang. At the time of writing, Hex has more than 8,000 packages, but it’s growing fast, and more packages are published every week.

You can find instructions for downloading and installing Elixir on the official Elixir website, as well as documentation and tutorials to help you get started.

To install Elixir on your computer, you have to install Erlang as a prerequisite, since Elixir uses the same virtual machine. The installation procedure depends on your operating system, and you can find detailed instructions here.

Once installed, you can see if everything works by typing **`iex`** from terminal. If it works, you should see something like this:

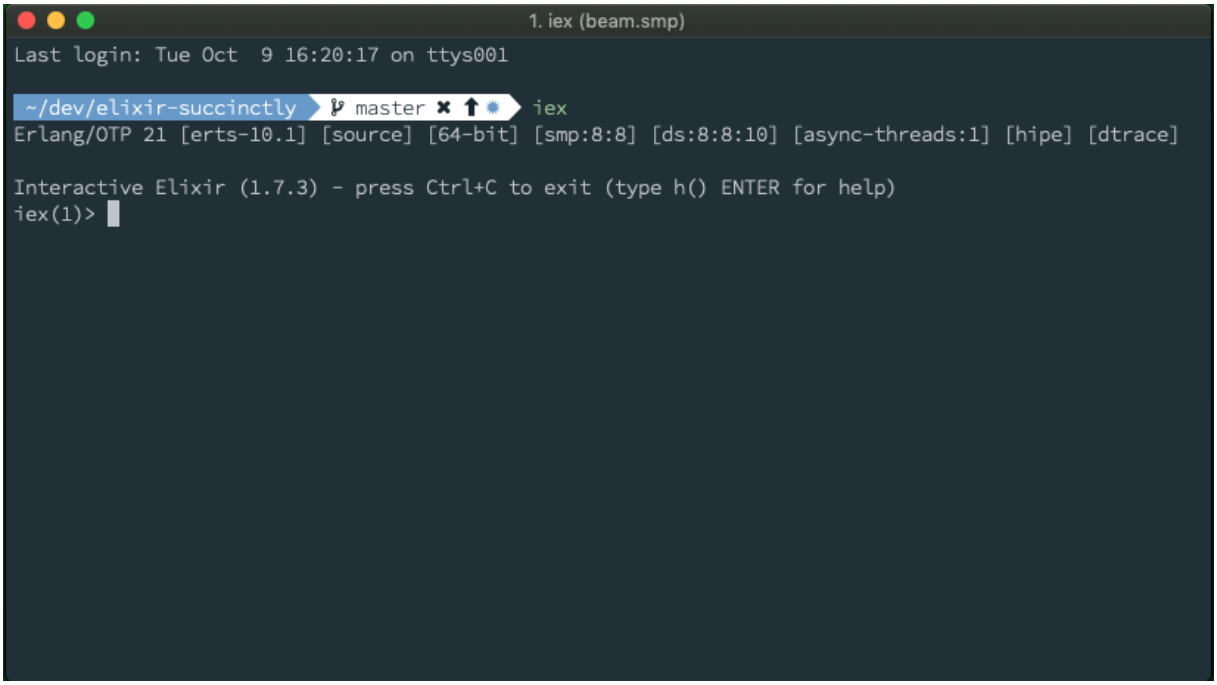


Figure 1 – iex REPL

Why is Elixir so interesting today?

Why are Elixir and Erlang so interesting for today's applications? And how can a platform developed more than 30 years ago—when the Internet had just been born—be useful today?

As said before, Erlang and the BEAM (the virtual machine that runs Erlang code) were designed with three ideas in mind: concurrency, fault tolerance, and distribution transparency.

Concurrency was needed by the domain: Ericsson is a company that builds telephone switches and routers, and one switch must be able to handle different phone calls concurrently. To be able to do this, every process must be independent from others, and must have its own private memory space. In Erlang every process is a sort of private space in which the code runs without sharing its data with other processes. To communicate with another process, it must send a message. The message arrives at the inbox of the destination process, and when the process is ready, it receives and processes it. This also means that inside a process, the code is always mono-thread/mono-process, and there's no need to sync, lock, or monitor the execution!

This is a very helpful simplification compared to Java or C#, in which concurrent or multi-thread programming is difficult to handle and debug.

www.dbooks.org

Another need of Ericsson's was to avoid communication interruption in their switches during calls, and if something accidentally goes wrong, they needed to restart the process as soon as possible. This is exactly what the Erlang VM does: in case of failure (exceptions), it kills the process and restarts it immediately, without affecting other processes. This feature is even more powerful, and gives the developers the ability to do hot updates to the application. That means you are able to change the code of your application in production without stopping it!

These processes are also easily distributable; the fact that a message is routed to a process that runs in the same virtual machine, or that is routed to a virtual machine that runs on another server, is completely transparent. Developers can start developing the application considering that all processes are running on the same machine, and then spread them to multiple machines (for example, for scalability) with little effort.

Consider the actual web applications: they usually are highly concurrent, and sometimes they need to be deployed on different machines for reaching performance needs. What programming platforms give the developer these facilities? Very few. One of the most-used programming languages for writing web applications is Node.js, which is a mono-thread programming language. For the developer, it's quite easy start with it (single thread apps are easy to understand), and it doesn't use all the power of your multi-core/multi-CPU server.

Today, most developers talk about microservices, but how hard is it to develop a solid microservice application? Writing a microservices application is hard because there isn't a single platform that developer can use to write all of the application. How do you make services communicate easily and reliably? Usually, third-party applications are needed (such as Message Bus, API, and brokers).

Elixir and Erlang, with their virtual machine, already have all the facilities that are needed to develop microservices applications inside—no need to install or configure other tools.

Another architecture that is gaining consensus in these years is the Actor Model. There are open-source frameworks for dealing with this architecture. The most famous is probably Akka for Java/Scala and Akka.NET for .NET. It works, and it's battle tested and well maintained, but what if you don't need a framework because the platform already has all the stuff built in? The Erlang Virtual Machine has all the facilities and components needed to build applications using the Actor Model pattern—battery included. It's a great feature for the same reason that concurrent-oriented programming languages should be built with these concepts from the beginning—adding them later is not a good option.

Today these features are mandatory in most applications; a web server is intrinsically concurrent and must deal with failures.

So even if it was built more than 30 years ago, the Erlang Virtual Machine is still one of the most interesting platforms for developing the applications that we need, and Elixir is a new functional programming language with a modern syntax (in some ways it resembles Ruby) that runs on one of the best virtual machines around.

Chapter 1 The Language

Now that Elixir is installed, we can start using it to explore its capabilities. Elixir is a functional language with immutable data structures and no concepts like objects, classes, methods, or other constructs of object-oriented programing (OOP). Everything is a function or a data structure that can be manipulated by a function. Functions are grouped in modules.

The easiest way to start is to type **iex** in a terminal. This command starts a REPL console in which you can start exploring the language.

Basic types

Elixir is dynamically typed, and the basic types are: numbers, Booleans, strings, atoms, lists, and tuples.

We can play with these types inside the REPL:

```
Erlang/OTP 21 [erts-10.1] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-
threads:1] [hipe] [dtrace]

Interactive Elixir (1.7.3) - press Ctrl+C to exit (type h() ENTER for
help)
iex(1)> 42
42
iex(2)>
nil
iex(3)>
```

Since Elixir is a functional language, every expression must return a value, so if you type **42**, the result of the expression is **42**. If you just type **Enter**, the result of the expression is **nil** (null value).

With numbers, we can do arithmetic:

```
iex(3)> 73 + 45
118
```

We can also evaluate logical operations:

www.dbooks.org

```
iex(4)> 42 == 44
false
iex(5)> true && false
false
```

Atoms are constant values, a sort of tag or label that never changes values. They can be compared to symbols in Ruby:

```
iex(6)> :an_atom
```

Atoms are preceded by a semicolon. The convention is to use snake_case.

Lists are just a series of values contained in square brackets:

```
iex(9)> [1, "hello", :foo_bar]
[1, "hello", :foo_bar]
```

Remember that, unlike with other programming languages (such as JavaScript), lists have no methods—they are just data structures. To work and manipulate lists, we must use the modules **`List`** or **`Enum`**.

The last basic type is the tuple:

```
iex(9)> {1, "hello", :foo_bar}
{1, "hello", :foo_bar}
```

As we can see, tuples are very similar to lists: the difference is that a list is stored as a linked list, while tuples are stored in a contiguous memory space, so updating a tuple is more expensive.

In Elixir tuples are often used as a return value. A classic way to return a value from a function is to use a tuple with two values: an atom (**`:ok`** or **`:error`**), and the value:

```
{:ok, 42}
{:error, "cannot calculate value"}
```

In addition to these basic types in Elixir, we can use more structured types, like maps. Maps are complex structures composed of keys and values:

```
iex(11)> %{name: "emanuele", country: "italy"}
%{country: "italy", name: "emanuele"}
```

Maps can also be nested to build more complex structures.

```
iex(12)> %{name: "emanuele", address: %{street: "via branze", city:
"brescia"}}
%{address: %{city: "brescia", street: "via branze"}, name: "emanuele"}
```

We'll look further into these types later in this chapter.

Strings

The string type needs a special mention because strings in Elixir are binaries, meaning a list of bytes. Binaries in Elixir are built using this syntax:

```
iex(1)> list_of_bytes = <<97, 98, 99, 100>>
"abcd"
```

As we can see, the REPL shows that list of bytes as a string "**abcd**" since **97**, **98**, **99**, and **100** are the code points of the characters **a**, **b**, **c**, and **d**.

This means the following operation matches correctly and returns the right part:

```
iex(2)> <<97, 98, 99, 100>> = "abcd"
```

Elixir, like most programming languages, supports string interpolation using the Ruby syntax:

```
"#{:hello_atom} elixir"
```

Strings can be concatenated by using a special operator:

```
iex(3)> "hello" <> "elixir"
"helloelixir"
```

String is also a module (see next chapter) that contains a set of useful functions for working with strings.

www.dbooks.org

Modules and functions

I've already mentioned that Elixir is a functional language. The types presented so far can be used to build complex structures that represent our data, but to manipulate them, we need a set of functions, since we don't have objects, classes, etc.

Like many languages (such as Python and Perl), Elixir statements can be placed in a file and executed. Simply use any text editor, type Elixir statements, and save the file with a **.exs** file extension ("Elixir script"). Then, run the file from a shell by typing **elixir** followed by the file name. For example:

```
> elixir say_hello.exs
"hello Elixir"
```

To better organize code in real-world applications, Elixir functions can be defined along a module that acts as a sort of container (namespace) for the functions that it contains.

Let's see an example:

```
defmodule MyFunctions do
  # functions here
end
```

To create a module, we use the **defmodule** macro, whose name we will use to call the function that it contains.

```
defmodule MyFunctions do

  def sum(a, b) do
    a + b
  end

  def sub(a, b) do
    a - b
  end
end
```

Here we have defined a module (**MyFunctions**) that contains a couple of functions: **sum** and **sub**, each with two arguments (arity).



Note: Arity is an important aspect for Elixir functions. A function in Elixir is determined by its name and arity (the number of arguments it takes). That means that sum/2 is different from sum/3. One of the message errors that appears quite often is:

***** (UndefinedFunctionError) function MyFunctions.sum/3 is undefined or private. Did you mean one of * sum/2?***

The implementation of these functions is straightforward; it simply applies the operation onto the argument. Functions don't need an explicit return; the last expression is the return value.

To call these functions, we have to use the full name:

```
MyFunctions.sum(4, 7)
```



Note: The parentheses are not mandatory in Elixir; we can still call the function when omitting them: *`MyFunctions.sum 4, 7`.*

Functions are the building block of Elixir applications. Applications are subdivided into modules to group functions. The functions in a group can be called using the "full name" (**`ModuleName.function_name`**), but since this can be awkward, there are other macros that help.

Since Elixir is a functional programming language, functions are first-class citizens in the world of Elixir. This usually means that we can treat functions as values to pass around. For example, we can have a function that receives a function as argument.

```
def print_result(f) do
  IO.puts f.()
end
```



Note: We omitted the module for brevity.

When the function is very short (a one-line-function), a compressed syntax can be used to define the function:

```
def sum(a, b), do: a + b
```

This is the same as **`def`** with **`do end`**, but written with the special one-line syntax. With functional programming languages, the number of “mini-functions” is very high, and using this syntax to define them is quite common.

The **`print_result`** function receives a function as an argument. In its body, it executes the function (it is a function with no parameters) and **`puts`** the result to the terminal. (**`IO`** is the input/output core module. Consider **`IO.put`** like a **`console.log`** in JavaScript).

The **`.()`** is the way to call functions that are defined like values.

www.dbooks.org

How can we use this **print_result** function? To use it, we have to create a function and pass it to **print_result**. The idiomatic way to do this in Elixir is:

```
a = fn -> 42 end
print_result(a)
```

The **fn** keyword is used to define an anonymous function. The previous example shows a simple function that returns **42**. The arity of this function is **0**, and we match the function with the variable **a** so that we can pass it to the **print_result** function.

The **fn** syntax to define anonymous functions is quite useful for defining simple one-line functions that are needed inside the scope. With **fn** we can also define functions that receive parameters:

```
iex(1)> sum = fn (a, b) -> a + b end
#Function<12.128620087/2 in :erl_eval.expr/5>
iex(2)> sum.(4, 5)
9
```

After the function declaration, the REPL shows the returned value that is a **Function** type.

Import

In real-world applications, functions from one module are often in another module. To avoid the use of the full name, we can use the **import** macro:

```
defmodule MyFunctions do

  def sum(a, b) do
    a + b
  end

  def sub(a, b) do
    a - b
  end
end

defmodule DoSomeMath do
  import MyFunctions

  def add_and_subtract(a, b, c) do
    sub(sum(a, b), c)
  end
end
```

As you can see, we are using **sum** and **sub** without their full names. This is thanks to the **import** directive: we are saying to the compiler that all functions from module **MyFunctions** are available in the module **DoSomeMath**.

Imports can be more selective. Some modules are quite big, and sometimes we only need one or two functions. The **import** macro can be used with additional arguments that specify which function should be imported:

```
import MyFunctions, only: [sum: 2]
```

The atom only specifies that we want only **sum/2** functions.



Note: Functions are identified in Elixir by name/arity. In the previous example, we are saying that we want function `sum` with two arguments.

Pattern matching

One great (and possibly the best) feature of Elixir is pattern matching. We already said that Elixir is a functional language. In functional programming values are immutable, which means that this code should not work:

```
iex(1)> a = 1
1
iex(2)> a = 2
2
```

If you try **iex**, this code actually works, so it seems that variables are mutable. But not quite.

In Elixir, the **=** character is not an assignment operator, but a match operator. In the previous example, we are binding the variable **a** to the right side of **=**, the first time with **1**, the and second time (rebind) with **2**.

What really happens is:

1. The expression on the right side is evaluated (**1** or **2**)
2. The resulting value is matched against the left side pattern (**a**)
3. If it matches the variable on the left side, it is bounded to the right-side value

This means we can write the following code:

www.dbooks.org

```
iex(1)> a = 3
3
iex(2)> 3 = a
3
```

The second line is pattern matching in action; we know that **a** is bounded to **3** (the first statement), so **3 = a** matches, and the value on the right-hand side is returned.

It could happen that the values don't match:

```
iex(1)> a = 3
iex(2)> 7 = a
** (MatchError) no match of right hand side value: 3
```

In this case Elixir returns a match error, as we expected.

There is one last thing to know about pattern matching: the pin operator. In some cases, instead of rebounding a variable to a new value, we want to verify the match. The pin operator (**^**) can help in these cases:

```
iex(1)> a = 3
iex(2)> ^a = 3
iex(3)> ^a = 7
** (MatchError) no match of right hand side value: 7
```

Using the pin operator, we are not going to rebind the variable—we are checking if the actual bound matches against the right-hand side.

Pattern matching is powerful, and is used a lot in everyday programming with Elixir. One example is using it to avoid conditionals:

```
defmodule Bot do
  def greet("") do
    IO.puts "None to greet."
  end

  def greet(name) do
    IO.puts "Hello #{name}"
  end
end
```

The **Bot** module has two **greet** functions: the first uses pattern matching to match against an empty string, and the second is matched in the other cases. This means we can avoid checking the value of **name** when deciding what to print.

Pay attention to the order in which the functions are declared—they are evaluated from top to bottom.

Pattern matching can be used to decompose lists:

```
iex(1)> [head | tail] = [1, 2, 3, 4]
[1, 2, 3, 4]
iex(2)> head
1
iex(3)> tail
[2, 3, 4]
```

In this example, the right side is a list with four values, and the left side is a couple of variables that are going to be bound to the **`head`** (**`1`**) and the **`tail`** (**`2,3,4`**) of the list.

We can also match the single elements:

```
iex(4)> [a, b, c] = [1, 2, 3]
[1, 2, 3]
iex(5)> a
1
iex(6)> b
2
```

In this case it's important to match all the elements in the list. If we miss one, the match fails:

```
iex(4)> [a, b] = [1, 2, 3]
** (MatchError) no match of right hand side value: [1, 2, 3]
```

If we don't care about an element in the list, we can use the underscore character (_) as a match:

```
iex(15)> [a,b, _] = [1,2,3]
[1, 2, 3]
```

The **_** means that I don't care about the value and don't capture it, so **`3`** is not bounded to a variable.

Since it works on lists, it can work on hash maps, too:

www.dbooks.org

```
iex(16)> %{a: a, b: b} = %{a: 4, b: "hello", c: :foo}
%{a: 4, b: "hello", c: :foo}
iex(17)> a
4
iex(18)> b
"hello"
```

In this example, we are matching the keys **a** and **b** against the map on the right. The values of **a** and **b** are bounded in the variables **a** and **b**.

Recursion

Recursion is not strictly tied to Elixir; is a general programming concept, but it is a pattern that is used quite often in functional programming. Elixir is no exception, and pattern matching is another peculiarity that helps in writing recursive programs.

Consider the example of **Factorial**. Elixir does not have loop statements like **for** or **while**; there are some functions in the **Enum** module, but none is a real loop. Recursion is a way to emulate loops.

```
defmodule Factorial do
  def do_it(0) do
    1
  end
  def do_it(n) do
    n * do_it(n - 1)
  end
end
```

Programs written with recursion always have the same pattern:

1. Determine and implement the basic case
2. Determine and implement the general case

In **Factorial**, the basic case is when the number is **0**, in which the **Factorial** is **1**. The first **do_it** function does exactly that.

In general, we multiply **n** for the **Factorial** of **n-1**.

Tail-call optimization

Coming from non-functional languages, it could be a little scary writing recursive functions like **Factorial**. We might think that the stack will be filled with returning addresses.

Elixir can manage this issue using tail-call optimization: if the very last thing the function does is the recursive call, the runtime can optimize the stack without adding a new frame, using the result as the new parameter for the next call. So if the condition is satisfied, there's no need to get worried about stack overflows.

Is the previous **`Factorial`** function optimized for tail calls? No! Even if the recursive call is the last instruction, it's not the last thing the function does. The last thing the function does is to multiply **`n`** for the factorial of **`n-1`**. So, to make the **`Factorial`** function tail-optimized, we must do a little bit of refactoring:

```
defmodule Factorial do
  def do_it(n) do
    internal_do_it(n, 1)
  end

  defp internal_do_it(0, acc) do
    acc
  end

  defp internal_do_it(n, acc) do
    internal_do_it(n - 1, acc * n)
  end

end
```

Moving the multiplication inside the call to **`internal_do_it`** resolves the optimization problem, and the recursive call is now the last thing the function does.

Take special care when writing recursive functions, since the stack overflow problem can be a subtle error to find.

Helpful modules

The Elixir core library comes with lots of helpful modules that accomplish most of the basic functionalities needed. To use them, you can start with the documentation and read through the modules to find something suitable for your needs.

Since these modules are in the standard library, you don't need to explicitly import them.

List and Enum

`List` and **`Enum`** are two modules that contain functions for manipulating enumerable lists of things. Lists in Elixir are linked lists, and you can think of them as recursive structures, like this:

www.dbooks.org

```
iex(1)> [1 | [ 2 | [3 | [ 4 ]]]]
[1, 2, 3, 4]
```

List are composed of pairs (Lisp uses the same approach), and can be concatenated using the special operator **++**:

```
iex(3)> [1, 3, 5] ++ [2, 4]
[1, 3, 5, 2, 4]
```

Lists can also be subtracted:

```
iex(4)> [1, 2, 3, 4, 5] -- [1, 3, 5]
[2, 4]
```

We already see that we can pattern-match on a list to extract the head:

```
iex(5)> [head | tail] = [1, 2, 3, 4]
[1, 2, 3, 4]
iex(6)> head
1
iex(7)> tail
[2, 3, 4]
```

This operator is very useful when writing recursive functions. Consider the following function, which sums up the elements in the list.

```
defmodule ListUtils do
  def sum([]) do
    0
  end

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

There are two **sum** functions: the first is called with an empty list, and the sum of an empty list is **0**. The second is called when the list is not empty. The list is decomposed in **head** and **tail**, and a recursive call is executed to add the value of the **head** to the sum of the rest of the list.

Though simple, these couple of functions illustrate some powerful features of Elixir. You probably first thought to use a loop to implement the function, but thanks to recursion and pattern matching, the loop is not needed, and the code is very easy to read and understand.

Actually, this operation could be implemented using a function of the **Enum** module. The **Enum** module contains the **reduce** function that can be used to aggregate values from a list:

```
iex(5)> Enum.reduce([1, 2, 3, 4], fn x, acc -> x + acc end)
10
```

The **Enum.reduce** function receives the enumerable list to work with, and for every element of the list, it calls the function passed as the second parameter. This function receives the current value and an accumulator (the result of the previous iteration). So, to sum all the values, it just needs to add to the accumulator the current value.

Another useful and well-known function of the **Enum** module is the **map** function. The **map** function signature is similar to reduce, but instead of aggregating the list in a single value, it returns a transformed array:

```
iex(6)> Enum.map(["a", "b", "c"], fn x -> String.to_atom(x) end)

[:a, :b, :c]
```

Here we are transforming strings into atoms.

Another function of the **Enum** module is **filter**:

```
iex(7)>  Enum.filter([1, 2, 3, 4, 5], fn x -> x > 2 end)
[3, 4, 5]
```

All these functions can be called using a more compact syntax. For example, the **map** function:

```
iex(8)> Enum.map(["a", "b", "c"], &String.to_atom/1)
[:a, :b, :c]
```

The **&String.to_atom/1** is a way to specify which function has to be applied to the element of the list: the function **String.to_atom** with arity **1**. The use of this syntax is quite typical.

The **List** module contains functions that are more specific to linked list, like **flatten**, **fold**, **first**, **last**, and **delete**.

www.dbooks.org

Map

Maps are probably the second-most used structure for managing application data, since it is easily resembled to an object with fields and values.

Consider a map like this:

```
book = %{
  title: "Programming Elixir",
  author: %{
    first_name: "Dave",
    last_name: "Thomas"
  },
  year: 2018
}
```

It is easy to view this map as a POJO/POCO; in fact, we can access its field using the well-known syntax:

```
iex(2)> book[:title]
"Programming Elixir"
```

Actually, we cannot change the attribute of the hash map—remember that in functional programming, values are immutable:

```
iex(5)> book[:title] = "Programming Java"
** (CompileError) iex:5: cannot invoke remote function Access.get/2 inside
match
```

To change a key value in a map, we can use the **`put`** function:

```
iex(6)> Map.put(book, :title, "Programming Elixir >= 1.6")
%{
  author: %{first_name: "Dave", last_name: "Thomas"},
  title: "Programming Elixir >= 1.6",
  year: 2018
}
```

The **`Map.put`** function doesn't update the map, but it creates a new map with the modified key. Maps have a special syntax for this operation, and the previous **`put`** can be rewritten like this:

```
iex(7)> new_book = %{ book | title:  "Programming Elixir >= 1.6"}
%{
  author: %{first_name: "Dave", last_name: "Thomas"},
  title: "Programming Elixir >= 1.6",
  year: 2018
}
```

The short syntax takes the original map and a list of attributes to change:

```
new_map = %{old_map | attr1: value1, attr2: value2, ...}
```

To read a value from a map, we already see the **[]** operator. The **Map** module has a special function to get the value, the **fetch** function:

```
iex(7)> Map.fetch(book, :year)
{:ok, 2018}
```

Here, for the first time, we see a usual convention used in Elixir: the use of a tuple to return a value from a function. Instead of returning just **2018**, **fetch** returns a tuple with the "state" of the operation and the result. Can a **fetch** fail in some way?

```
iex(8)> Map.fetch(book, :foo)
:error
```

This way of returning results as tuples is quite useful when used in conjunction with pattern matching.

```
iex(9)> {:ok, y} = Map.fetch(book, :year)
{:ok, 2018}
iex(10)> y
2018
```

We call **fetch**, pattern matching the result with the tuple **{:ok, y}**. If it matches, in **y** we will have the value **2018**.

In case of error, the match fails, and we can branch to better manage the error using a **case** statement (which will see later).

www.dbooks.org

```
iex(11)> {:ok, y} = Map.fetch(book, :foo)
** (MatchError) no match of right hand side value: :error
    (stdlib) erl_eval.erl:453: :erl_eval.expr/5
    (iex) lib/iex/evaluator.ex:249: IEx.Evaluator.handle_eval/5
    (iex) lib/iex/evaluator.ex:229: IEx.Evaluator.do_eval/3
    (iex) lib/iex/evaluator.ex:207: IEx.Evaluator.eval/3
    (iex) lib/iex/evaluator.ex:94: IEx.Evaluator.loop/1
    (iex) lib/iex/evaluator.ex:24: IEx.Evaluator.init/4
```

Control flow

We already saw that with pattern matching, we can avoid most conditional control flows, but there are cases in which an **`if`** is more convenient.

Elixir has some control flow statements, like **`if`**, **`unless`**, and **`case`**.

`if` has the classic structure of **`if…else`**:

```
if test_conditional do
  # true case
else
  # false case
end
```

Since everything in Elixir is an expression, and **`if`** is no exception, the **`if…else`** construct returns a value that we can assign to a variable for later use:

```
a = if test_conditional do
# ...
```

When there are more than two cases, we can use the **`case`** statement in conjunction with pattern matching to choose the correct option:

```
welcome_message = case get_language(user) do
  "IT" -> "Benvenuto #{user.name}"
  "ES" -> "Bienvenido #{user.name}"
  "DE" -> "Willkommen #{user.name}"
  _ -> "Welcome"
end
```

The last case is used when none of the previous cases match with the result. **case** is a sort of switch where the **_** case is the default. As with **if**, **case** returns a value; here, **welcome_message** can be used.

Guards

In addition to control flow statements, there are guards that can be applied to functions, meaning that the function will be called if the guard returns **true**:

```
defmodule Foo do
  def divide_by_10(value) when value > 0 do
    value / 10
  end
end
```

The **when** clause added on the function signature says that this function is available only if the passed value is greater than **0**. If we pass a value that is equal to **0**, we obtain a match error:

```
iex(4)> Foo.divide_by_10(0)
** (FunctionClauseError) no function clause matching in Foo.divide_by_10/1

    The following arguments were given to Foo.divide_by_10/1:

        # 1
        0

    iex:2: Foo.divide_by_10/1
```

Guards works with Boolean expressions, and even with a series of build-in functions, like: **is_string**, **is_atom**, **is_binary**, **is_list**, **is_map**.

```
defmodule Foo do
  def divide_by_10(value) when value > 0 and (is_float(value) or
is_integer(value)) do
    value / 10
  end
end
```

In this case, we are saying that the **divide_by_10** function can be used with numbers greater than **0**.

www.dbooks.org

Pipe operator

Elixir supports a special flow syntax to concatenate different functions.

Suppose, for example, that we need to filter a list to obtain only the values greater than **5**, and to these values we have to add **10**, sum all the values, and finally, print the result to the terminal.

The classic way to implement it could be:

```
iex(14)> IO.puts Enum.reduce(Enum.map(Enum.filter([1, 3, 5, 7, 8, 9], fn x
-> x > 5 end), fn x -> x + 10 end), fn acc, x -> acc + x end)
54
:ok
```

Not very readable, but in functional programming, it's quite easy to write code that composes different functions.

Elixir gives us the pipe operator **|>** that can compose functions in an easy way, so that the previous code becomes:

```
iex(15)> [1, 3, 5, 7, 8, 9] |> Enum.filter(fn x -> x > 5 end) |>
Enum.map(fn x -> x + 10 end) |> Enum.reduce(fn acc, x -> acc + x end) |>
IO.puts
```

The pipe operator gets the result of the previous computation and passes it as the first argument to the next one. So in the first step, the list is passed as the first argument to **Enum.filter**, the result is passed to the next, and so on.

This way, the code is more readable, especially if we write it like this:

```
[1, 3, 5, 7, 8, 9]
  |> Enum.filter(fn x -> x > 5 end)
  |> Enum.map(fn x -> x + 10 end)
  |> Enum.reduce(fn acc, x -> acc + x end)
  |> IO.puts
```

Type specifications

Elixir is a dynamic language, and it cannot check at compile time that a function is called with the right arguments in terms of number, and even in terms of types.

But Elixir has features called specs and types that are helpful in specifying modules' signatures and how a type is composed. The compiler ignores these specifications, but there are tools that can parse this information and tell us if everything matches.

These features are the **`@spec`** and **`@type`** macros.

```
defmodule Math do
  @spec sum(integer, integer) :: integer
  def sum(a, b) do
    a + b
  end
end
```

The **`@spec`** macro comes just before the function to document. In this case, it helps us understand that the **`sum`** function receives two integers, and returns an integer.

The **`integer`** is a built-in type; you can find additional types here.

Specs are also useful for functions that return different values:

```
defmodule Math do
  @spec div(integer, integer) :: {:ok, integer} | {:error, String.t }
  def div(a, b) do
    # ...
  end
end
```

In this example, the **`div`** returns a tuple: **`{ok, result}`** or **`{:string, "error message"}`**.

But since Elixir is a dynamic language, how can the specs help in finding errors? The compiler itself doesn't care about the specifications—we must use Dialyzer, an Erlang tool that analyzes the specs and identifies possible issues (mainly type mismatch and non-matched cases).

Dialyzer, which came from Erlang, is a command-line tool that analyzes the source code. To simplify the use of Dialyzer, the Elixir community has created a tool called Dialyxir that wraps the Erlang tool and integrates it with Elixir tools.

The **`spec`** macro is usually used in conjunction with the **`type`** and **`struct`** macros that are used to define new types:

www.dbooks.org

```
defmodule Customer do
  @type entity_id() :: integer()

  @type t :: %Customer{id: entity_id(), first_name: String.t, last_name:
String.t}
  defstruct id: 0, first_name: nil, last_name: nil
end

defmodule CustomerDao do
  @type reason :: String.t
  @spec get_customer(Customer.entity_id()) :: {:ok, Customer} | {:error,
reason}
  def get_customer(id) do
    # ...
    IO.puts "GETTING CUSTOMER"
  end
end
```

Let's take a closer look at this code sample, starting with **`@type entity_id() :: integer()`**. This is a simple type alias; we have defined the type **`entity_id`**, which is an integer. Why have a special type for an integer? Because **`entity_id`** is speaking from a documentation point of view, and is contextualized since it represents an identity for a customer (it could be a primary key or an ID number). We won't use **`entity_id`** in another context, like **`sum`** or **`div`**.

We have a new type **`t`** (name is just a convention) to specify the shape of a customer that has an ID: a **`first_name`** and a **`last_name`**. The syntax **`%Customer{ ... }`** is used to specify a type that is a structure (see the next line). We can think of it as a special HashMap or a record in other languages.

The **`struct`** is defined just after the **`typespec`**; it contains an **`id`**, a **`first_name`**, and a **`last_name`**. To this attribute, the **`defstruct`** macro also assigns default values.

This couple of lines define the shape of a new complex type: a **`Customer`** with its attributes. Again, we could have used a simple hash, but **`structs`** with **`type`** defines a better context, and the core result is more readable.

After the customer module in which there is any code, we open the **`CustomerDao`** module that uses the types defined previously.

The function **`get_customer`** receives an **`entity_id`** (an integer) and returns a tuple that contains an atom (**`:ok`**) and a **`Customer struct`**, or a tuple with the atom **`:error`** and a reason (**`String`**).

Adding all this metadata to our programs comes with a cost, but if we are able to start from the beginning, and the application size grows to a certain level, it's an investment with a high return in value, in terms of documentation and fewer bugs.

Behavior and protocols

Elixir is a functional programming language that supports a different paradigm than C# or Java, which are object-oriented programming (OOP) languages. One of the pillars of OOP is polymorphism. Polymorphism is probably the most important and powerful feature of OOP in terms of composition and code reuse. Functional programming languages can have polymorphism too, and Elixir uses behavior and protocols to build polymorphic programs.

Protocols

Protocols apply to data types, and give us a way to apply a function to a type.

For example, let's say that we want to define a protocol to specify that the types that will implement this protocol will be printable in CSV format:

```
defprotocol Printable do
  def to_csv(data)
end
```

The **`defprotocol`** macro opens the definition of a protocol; inside, we define one or more functions with its own arguments.

It is a sort of interface contract: we can say that every data type that is printable will have an implementation for the **`to_csv`** function.

The second part of a protocol is the implementation.

```
defimpl Printable, for: Map do
  def to_csv(map) do
    Map.keys(map)
      |> Enum.map(fn k -> map[k] end)
      |> Enum.join(",")
  end
end
```

We define the implementation using the **`defimpl`** macro, and we must specify the type for which we are writing the implementation (**`Map`** in this case). In practice, it is as if we are extending the **`map`** type with a new **`to_csv`** function.

In this implementation, we are extracting the keys from the map (**`:first_name, :last_name`**), and from these, we are getting the values using a **`map`** on the **`keys`** list. And finally, we are joining the list using a comma as a separator.

www.dbooks.org

```
iex(1)> c("./samples/protocols.exs")
[Printable.Map, Printable]
iex(2)> author = %{first_name: "Dave", last_name: "Thomas"}
%{first_name: "Dave", last_name: "Thomas"}
iex(3)> Printable.to_csv(author) # -> "Dave, Thomas"
"Dave,Thomas"
```



Note: If we save the protocol definition and protocol implementation in a script file (.exs), we can load it in the REPL using the `c` function (compile). This will let us use the module's function defined in the script directly in the REPL.

Can we implement the same protocol for other types? Sure—let's do it for a list.

```
defimpl Printable, for: List do
  def to_csv(list) do
    Enum.map(list, fn item -> Printable.to_csv(item) end)
  end
end
```

Here we are using the **`to_csv`** function that we have defined for the **`Map`**, since **`to_csv`** for a list is a list of **`to_csv`** for its elements.

```
iex(1)> c("./samples/protocols.exs")
[Printable.List, Printable.Map, Printable]
iex(2)> author1 = %{first_name: "Dave", last_name: "Thomas"}
%{first_name: "Dave", last_name: "Thomas"}
iex(3)> author2 = %{first_name: "Kent", last_name: "Beck"}
%{first_name: "Kent", last_name: "Beck"}
iex(4)> author3 = %{first_name: "Martin", last_name: "Fowler"}
%{first_name: "Martin", last_name: "Fowler"}
iex(5)> Printable.to_csv([author1, author2, author3])
["Dave,Thomas", "Kent,Beck", "Martin,Fowler"]
```

In the output, we have a list of CSV strings! But what happens if we try to apply the **`to_csv`** function to a list of numbers? Let's find out.

```
iex(1)> c("./samples/protocols.exs")
[Printable.List, Printable.Map, Printable]
iex(2)> Printable.to_csv([1,2,3])
** (Protocol.UndefinedError) protocol Printable not implemented for 1
    samples/protocols.exs:1: Printable.impl_for!/1
    samples/protocols.exs:2: Printable.to_csv/1
    (elixir) lib/enum.ex:1314: Enum."-map/2-lists^map/1-0-"/2
```

The error message is telling us that **Printable** is not implemented for numbers, and the runtime doesn't know what to do with **to_csv(1)**.

We can also add an implementation for **Integer** if we think that we are going to need it:

```
defimpl Printable, for: Integer do
  def to_csv(i) do
    to_string(i)
  end
end
```

```
iex(1)> c("./samples/protocols.exs")
[Printable.Integer, Printable.List, Printable.Map, Printable]
iex(2)> Printable.to_csv([1,2,3])
["1", "2", "3"]
```

Elixir has some protocols already implemented. One of the most popular is the **to_string** protocol, available for almost every type. **to_string** returns a string interpretation of the value.

Behaviors

The other interesting feature that resembles functional polymorphism is behaviors. Behaviors provide a way to define a set of functions that have to be implemented by a module (a contract) and ensure that a module implements all the functions in that set.

Interfaces? Sort of. We can define a behavior by using the **@callback** macro and specifying the signature of the function in terms of specs.

```
defmodule TalkingAnimal do
  @callback say(what :: String.t) :: { :ok }
end
```

We are defining an "interface" for a talking animal that is able to say something. To implement the behavior, we use another macro.

www.dbooks.org

```
defmodule Cat do
  @behaviour TalkingAnimal
  def say(str) do
    "miaooo"
  end
end

defmodule Dog do
  @behaviour TalkingAnimal
  def say(str) do
    "woff"
  end
end
```

This resembles the classic strategy pattern. In fact, we can use functions without knowing the real implementation.

```
defmodule Factory do
  def get_animal() do
    # can get module from configuration file
    Cat
  end
end

animal = Factory.get_animal()
IO.inspect animal.say("hello") # "miaooo"
```

If the module is marked with the **`@behaviour`** macro but the function is not implemented, the compiler raises an error, **`undefined behaviour function`**, stating that it can't find the declared implementation.

Behaviors and protocols are two ways to define a sort of contract between modules or types. Always remember that Elixir is a dynamic language, and it can't be so strict like Java or C#. But with Dialyzer, specs, behaviors, and protocols can be quite helpful in defining and respecting contracts.

Macros

One of the most powerful features of Elixir are the macros. Macros in Elixir are language constructs used to write code that generate new code. You might be familiar with the concept of metaprogramming and abstract syntax trees; macros are what you need to do metaprogramming in Elixir.

It is a difficult topic, and in this chapter, we only see a soft introduction to macros. However, you most likely won’t need to write macros in your daily work with Elixir.

First of all, most of the Elixir constructs that we already used in our examples are macros: **`if`** is defined as a macro, **`def`** is a macro, and **`defmodule`** is a macro. Actually, Elixir is a language with very few keywords, and all the other keywords are defined as macros.

Macros, metaprogramming, and abstract syntax trees (AST) are all related. An AST is a representation of code, and in Elixir, an AST is represented as a tuple. To view an AST, we can use the instruction **`quote`**:

```
iex(1)> quote do: 4 + 5
{:+, [context: Elixir, import: Kernel], [4, 5]}
```

We get back a tuple that contains the function (**`:+`**), a context, and the two arguments **`[4,5]`**. This tuple represents the function that sums **`4`** to **`5`**. As a tuple, it is data, but it is also code because we can execute it:

```
iex(2)> Code.eval_quoted({:+, [context: Elixir, import: Kernel], [4, 5]})
{9, []}
```

Using the module **`Code`**, we can evaluate an AST and get back the result of the execution. This is the basic notion we need to understand AST. Now let's see how can we use an AST to create a macro.

Consider the following module. It represents a **`Logger`** module with just one function to log something to the terminal:

```
defmodule Logger do
  defmacro log(msg) do
    if is_log_enabled() do
      quote do
        IO.puts("> From log: #{unquote(msg)}")
      end
    end
  end
end
```

The **`defmacro`** is used to start the definition of a macro; it receives a message to be logged. The implementation checks the value of **`is_log_enabled`** function (suppose that this function will check a setting or an environment variable), and if that value is true, it returns the AST of the instruction **`IO.puts`**.

The **`unquote`** function is sort of the opposite of **`quote`**: since we are in a quoted context, to access the value of **`msg`**, we need to step out of the quoted context to read that value—**`unquote(msg)`** does exactly that.

What does this module do? This **`Logger`** logs the information only if the logging is enabled. If it is not enabled, it doesn't even generate the code necessary to log, meaning it does not affect the application's performance, since no code is generated.

www.dbooks.org

Macros and metaprogramming are difficult topics, and they are not the focus of this book. One of the main rules of writing macros is to not write them unless you really need to. They are useful for writing in a DSL or doing some magical stuff, but their introduction always comes at a cost.

Chapter 2 The Platform

In the previous chapter we learned how Elixir works, how to use its syntax, and how to write functions and small programs to do some basic stuff. But the real power of Elixir is the platform itself, based on the Erlang ecosystem.

In the introduction we said that one of the most used architectures in Erlang is the actor model, and that everything is a process. Let's start with the process.

Spawning a process in Elixir is very easy and very cheap. They are not real operating system processes, but processes of the virtual machine in which the Elixir application runs. This is what gives them a light footprint, and it's quite normal for a real-world application to spawn thousands of processes.

Let's begin by learning how to spawn a process to communicate with it. Consider this module:

```
defmodule HelloProcess do
  def say(name) do
    IO.puts "Hello #{name}"
  end
end
```

This is a basic "Hello World" example that we can execute just by calling **`HelloProcess.say("adam")`**, and it will print **`Hello adam`**. In this case, it runs in the same process of the caller:

```
iex(1)> c("hello_process.exs")
iex(2)> HelloProcess.say("adam")
"Hello adam"
iex(3)>
```

Here we are using the module as usual, but we can spawn it in a different process and call its functions:

```
iex(1)> c("hello_process.exs")
iex(2)> spawn(HelloProcess, :say, [“adam”])
Hello adam
#PID<0.124.0>
```

The **`spawn/3`** function runs the **`say`** function of the module **`HelloProcess`** in a different process. It prints **`Hello adam`** and returns a PID (process ID), in this case **`0.124.0`**. PIDs are a central part of the Erlang/Elixir platform because they are the identifiers for the processes.

A PID is composed of three parts: A.B.C.

www.dbooks.org

A is the node number. We have not talked about nodes yet; consider them the machine in which the process runs. **0** stands for the local machine, so all the PIDs that start with **0** are running on the local machine.

B is the first part of the process number, and C is the second part of the process number (usually **0**).

Everything in Elixir has a PID, even the REPL:

```
iex(1)> self
#PID<0.105.0>
```

The **self** returns the PID of the current process, in this case the REPL (**iex**).

We can use the PID to inspect a process status using the **Process** module.

```
iex(1)> Process.alive?(self)
true
```

We can try with our **HelloProcess** module:

```
iex(12)> pid = spawn(HelloProcess, :say, ["adam"])
Hello adam
#PID<0.133.0>
iex(13)> Process.alive?(pid)
false
```

As we can see, the process is dead after the execution. This happens because there is nothing that keeps the process alive—it simply puts the string on the console, and then terminates.

The **HelloProcess** module is not very useful; we need something that do some calculation that we can spawn to another process to keep the main process free.

Let's write it:

```
defmodule AsyncMath do
  def sum(a, b) do
    a + b
  end
end
```

This module is very simple, but we need it to start thinking about process communication. So we can start using this module:

```
iex(1)> c("async_math.exs")
[AsyncMath]
iex(2)> pid = spawn(AsyncMath, :sum, [1,3])
#PID<0.115.0>
iex(3)> Process.alive?(pid)
False
```

As we can see, it simply returns the PID of the spawned process. In addition, the process dies after the execution, so that we cannot obtain the result of the operation.

To make the two processes communicate, we need to introduce two new instructions: **receive** and **send**. **Receive** is a blocking operation that suspends the process waiting for new messages. Messages are the way process communicates: we can send a message to a process, and it can respond by sending a message.

We can refactor our module like this:

```
defmodule AsyncMath do
  def start() do
    receive do
      {:sum, [x, y], pid} ->
        send pid, {:result, x + y}
    end
  end
end
```

We have defined a **start** function that is the entry point for this module; we will use this function to spawn the process.

Inside the **start** function, we wait for a message using the **receive do** structure. Inside **receive**, we expect to receive a message (a tuple) with this format:

```
{:sum, [x, y], pid}
```

The format consists of an atom (**:sum**), an array with an argument for **sum**, and the **pid** of the sender. We pattern match on this and respond to the caller using a **send** instruction.

send/2 needs the **pid** of the process to send a message: a tuple with **:result**, and the result (**sum** of **x + y).**

If everything is set up correctly, we can load the new module and try it in the REPL:

www.dbooks.org

```
iex(1)> c("async_math.exs")
[AsyncMath]
iex(2)> pid = spawn(AsyncMath, :start, [])
#PID<0.151.0>
iex(3)> Process.alive?(pid)
true
iex(4)> send(pid, {:sum, [1, 3], self})
{:sum, [1, 3], #PID<0.105.0>}
iex(5)> Process.alive?(pid)
false
```

What have we done here? We loaded the **async_math** module and spawned the process using the **spawn** function with the **start** function of the module.

Now the module is alive because it is waiting for a message (**receive…do**). We send a message requesting the sum of **1** and **3**. The **send** function returns the sent message, but not the result. In addition to this, the process after the send is dead.

How can we get our result?

One thing I have not yet mentioned is that every process in Elixir has an inbox, a sort of queue in which all of its messages arrive. From that queue, the process dequeues one message at a time, processes it, and then goes to the next one. That's why I said that inside a process, the code is single thread/single process, because it works on a single message at a time.

This mechanism is also at the base of the actor model, in which each actor has a dedicated queue that stores the messages to process, and an actor works on a single time.

Going back to our example, the queue that stores the messages is the queue of the REPLS, since it is that process that asks for the sum of **1** and **3**. We can see what's inside the process queue by calling the function **flush**, which flushes the inbox and prints the messages to the console:

```
iex(6)> flush
{:result, 4}
```

Here is our expected result: **flush** prints the messages in the queue (in this case, just one). The message has the exact shape that we used to send the result.

Now that we have asked for a result, we can try to ask for another operation:

```
iex(6)> send(pid, {:sum, [5, 8], self})
iex(7)> flush
:ok
```

This time the inbox is empty: it seems like our request or the response to our request has gotten lost. The problem is that the **AsyncMath.start** function wait for the first message, but as soon as the first message is processed, it goes out of scope. The **receive do** macro does not loop to itself after a message is received.

To obtain the desired result, we must do a recursive call at the end of the **start** function:

```
defmodule AsyncMath do
  def start() do
    receive do
      {:sum, [x, y], pid} ->
        send pid, {:result, x + y}
    end
    start
  end
end
```

At the end of the **receive** block, we do a recursive call to **start** so that the process will go in a "waiting for message" mode.

With this change, we can call the **sum** operation anytime we want:

```
iex(1)> c("async_math.exs")
[AsyncMath]
iex(2)> pid = spawn(AsyncMath, :start, [])
#PID<0.126.0>
iex(3)> send(pid, {:sum, [5, 4], self})
{:sum, [5, 4], #PID<0.105.0>}
iex(4)> send(pid, {:sum, [3, 9], self})
{:sum, [3, 9], #PID<0.105.0>}
iex(5)> flush
{:result, 9}
{:result, 12}
:ok
iex(6)>
```

When we call the **flush** function, it prints out the two messages in the inbox with the two results. This happens because the recursive call to **start** keeps the process ready to receive new messages.

We have seen a lot of new concepts about Elixir and processes.

To recap:

- We created basic module that, when started, waits for a message and responds with another message
- We spawned that function to another process
- We sent a message using the send function
- We flushed the inbox of the REPL to view the result

www.dbooks.org

Is there a better way to capture the result? Yes, by using the same pattern of the **AsyncMath** module.

```
defmodule AsyncMath do
  def start() do
    receive do
      {:sum, [x, y], pid} ->
        send pid, {:result, x + y}
    end
    start()
  end
end

pid = spawn(AsyncMath, :start, [])
send pid, {:sum, [5, 6], self()}

receive do
  {:result, x} -> IO.puts x
end
```

We can put our program in waiting even after the execution of the **sum** operation—remember that the messages remain in the inbox queue, so we can process them after they arrive (unlike with events).

Now we have seen the basics of processes. We also have seen that even if it is cheap to spawn a process, in a real-world application it's not very feasible create processes and communicate with them using the low-level function that we have seen. We need something more structured and ready to use.

With Elixir and Erlang comes the OTP (Open Telecom Platform), a set of facilities and building blocks for real-world applications. Even though Telecom is in the name, it is not specific to telecommunications— it's more of a development environment for concurrent applications. OTP was built with Erlang (and in Erlang), but thanks to the complete interoperability between Erlang and Elixir, we can use all of the facilities of OTP in our Elixir program with no cost at all.

Elixir applications

Until now, we've worked with simple Elixir script files (.exs). These are useful in simple contexts, but not applicable in real-world applications.

When we installed Elixir, we also got **mix**, a command-line tool used to create and manipulate Elixir projects. We can consider **mix** as a sort of **npm** (from Node.js). From now on, we will use **mix** to create projects and manage projects.

Let's create a new project now:

```
~/dev> mix new sample_app
* creating README.md
* creating .formatter.exs
* creating .gitignore
* creating mix.exs
* creating config
* creating config/config.exs
* creating lib
* creating lib/sample_app.ex
* creating test
* creating test/test_helper.exs
* creating test/sample_app_test.exs

Your Mix project was created successfully.
You can use "mix" to compile it, test it, and more:

    cd sample_app
    mix test

Run "mix help" for more commands.
```

This CLI command creates a new folder named **`sample_app`** and puts some files and folders inside.

We'll now have a quick look at some of these files.

www.dbooks.org

Mix.exs

```
defmodule SampleApp.MixProject do
  use Mix.Project

  def project do
    [
      app: :sample_app,
      version: "0.1.0",
      elixir: "~> 1.8",
      start_permanent: Mix.env() == :prod,
      deps: deps()
    ]
  end

  # Run "mix help compile.app" to learn about applications.
  def application do
    [
      extra_applications: [:logger]
    ]
  end

  # Run "mix help deps" to learn about dependencies.
  defp deps do
    [
      # {:dep_from_hexpm, "~> 0.3.0"},
      # {:dep_from_git, git: "https://github.com/elixir-lang/my_dep.git",
tag: "0.1.0"},
    ]
  end
end
```

The **mix** file acts as a project file (a sort of package.json). It contains the app manifest, the application to launch, and the list of dependencies. Don't worry if everything is not clear right now—we will learn more as we go.

There are two important things to note here. First is the **deps** function that returns a list of external dependencies from which the application depends. Dependencies are specified as a tuple with the name of the package (in the form of an atom) and a string that represents the version.

The other important thing is the application function that we will use to specify with module should start at the beginning. In this template, there is only a **logger**.

Remember that comments start with a number sign (**#**) character.

Sample_app.ex

```
defmodule SampleApp do
  @moduledoc """
  Documentation for SampleApp.
  """

  @doc """
  Hello world.

  ## Examples

      iex> SampleApp.hello()
      :world

  """
  def hello do
    :world
  end
end
```

Sample_app.ex is the main file of this project, and by default consists only of a function that returns the atom **:world**. It's not useful; it's just a placeholder.

Sample_app_test.exs

```
defmodule SampleAppTest do
  use ExUnit.Case
  doctest SampleApp

  test "greets the world" do
    assert SampleApp.hello() == :world
  end
end
```

This is a template for a simple test of the function **SampleApp.hello**. It uses **ExUnit** as a test framework. To run the tests from the terminal, we must write **mix test**:

www.dbooks.org

```
~/dev> mix test
Compiling 1 file (.ex)
Generated sample_app app
..

Finished in 0.04 seconds
1 doctest, 1 test, 0 failures

Randomized with seed 876926
```

The other files are not important right now; we will look more closely at some of them in the upcoming chapters.

To start the application, we must use **`mix`** from the terminal:

```
~/dev> mix run
Compiling 1 file (.ex)
Generated sample_app app
```

Actually, the application does nothing.

GenServer

One of the most frequently used modules of OTP is the **`GenServer`** that represents a basic generic server, a process that lives by its own and is able to process messages and response to action.

`GenServer` is a behavior that we can decide to implement to adhere to the protocol. If we do it, we obtain a server process that can receive, process, and respond to messages.

`GenServer` has the following capabilities:

- Creating a server process
- Managing the server state
- Creating a server process
- Manage the server state
- Handling requests and sending responses
- Stopping the server
- Handling failures

It is a behavior, so the implementation details are up to us. **`GenServer`** lets us implement some functions (called callbacks) for customizing its details:

- **`init/1`** acts as a constructor, and is called when the server is started. The expected result is a tuple **`{:ok, state}`** that contains the initial state of the server.
- **`handle_call/3`** is the callback that is called when a message arrives to the server. This is a synchronous function, and the result is a tuple **`{:reply, response,`**

new_state} that contain the response to send to the caller, and the new state to apply to the server.

- **handle_cast/2** is the asynchronous callback, which means that it doesn't have to respond to the caller directly. The expected result is a tuple **{:noreply, new_state}** that contains the new state.
- **handle_info/2** receives all the messages that are note captured (by pattern matching) by the others' callbacks.
- **terminate/2** is called when server is about to exit, and is useful for cleanup.

Our task is to implement the needed callbacks to define the right behavior of our application.

Let's look at an example to see how **GenServer** works. We'll implement a sort of key/value in the memory store (consider it a minimal Redis database implementation).

The server will process two kind of messages:

- **{:set, key, value}** will store the value in memory assigning the key
- **{:get, key}**will return the value of the given key

```
defmodule MiniRedis do
  use GenServer

  def init(_) do
    {:ok, %{}}
  end

  def handle_call({:set, key, value}, _from, state) do
    {:reply, :ok, Map.merge(state, %{key => value})}
  end

  def handle_call({:get, key}, _from, state) do
    {:reply, Map.fetch(state, key), state}
  end
end
```

Here is a first basic implementation. The **init** function defines the initial state, which is basically an empty hash map.

The two **hande_call** functions handle the two types of messages that are important for the server: **set** and **get** a value. We are using the **handle_call** because we need a synchronous interface to interact with. The implementation of the two function is straightforward. Note that they reply with a tuple: an atom (**:reply)**, a response value **:ok** for the **set** operation, the value for the **get** operation, and the new **state**.

The new state (in the case of **get**) is equal to the current state, since **get** is idempotent.

To try this server, we can open the REPL and do something like this:

www.dbooks.org

```
iex(1)> c("mini_redis.exs")
[MyServer]
iex(2)> {:ok, pid} = GenServer.start_link(MiniRedis, [])
{:ok, #PID<0.114.0>}
iex(3)> GenServer.call(pid, {:set, "greet", "ciao"})
:ok
iex(4)> GenServer.call(pid, {:get, "greet"})
{:ok, "ciao"}
```

Let's see what we've got here. Using the **c** function, we load the module **MiniRedis**. Then we start it using the **start_link** function of **GenServer** module. The **start_link** function receives the module to start as a server, and a list of parameters (empty in this case).

The result of **start** is the **:ok** atom and the **pid** of the process. We use the **pid** to use the server thought the call function that expect the message in the correct format. In this example, we call **set** and then **get** to check the result.

Even if everything works perfectly, it's not that easy to call a **GenServer** using a **pid** and know the exact format of the messages to send. That's why a **GenServer** usually has a public interface that the client can interact with.

We now modify the first implementation to add the public interface:

```
defmodule MiniRedis do
  use GenServer

  def init(_) do
    {:ok, %{}}
  end

  def start_link(opts \\ []) do
    GenServer.start_link(__MODULE__, [], opts)
  end

  def set(key, value) do
    GenServer.call(__MODULE__, {:set, key, value})
  end

  def get(key) do
    GenServer.call(__MODULE__, {:get, key})
  end

  def handle_call({:set, key, value}, _from, state) do
    {:reply, :ok, Map.merge(state, %{key => value})}
  end

  def handle_call({:get, key}, _from, state) do
    {:reply, Map.fetch(state, key), state}
  end
end
```

This version of our **MiniRedis** server has three more functions. The **start_link**, **set**, and **get** functions are part of the public interface. This definition comes from the fact that these three functions run in the client process, and not in the server process. Their implementation in fact call the server function using the **call** function of the **GenServer** module. The **call** function sends a message to the real server, and **handle_call** receives those messages.

It's also better to use the public interface to interact with the server because the client doesn't need to know the **pid** of the server process; it just calls **MiniRedis.put(…)**, which is the implementation of the client interface that manages the identification of the server to call.

In our example, we are using a simple technique to define a **GenServer** name: we are using the **__MODULE__** macro that expands to module name (**MiniRedis**) as the name of our server. Using the module name means that we can have just one instance of the server running, and in some cases, this is just fine.

Another option is to use the options of the **start_link** function to specify the server name:

www.dbooks.org

```
def start_link(opts \\ []) do
  GenServer.start_link(__MODULE__, [], name: :mini_redis)
end
```

Using an atom to specify a name is another option. The atom can also be passed as a parameter so that we can create different process of the same **GenServer**.

A more common option is the **Registry** module that stores the active servers' names and PIDs. We will see how to use **Registry** in the next chapter.

Supervisors

Another important component for writing Elixir (and Erlang) applications is the supervisor. Supervisors are special servers whose only task is to monitor another set of processes and decide what to do when they crash or die.

Suppose that our application is composed of a set of **GenServer**s that do some job. Since these **GenServer**s could crash, a supervisor is needed to monitor failures and (for example) restart the servers. That's the role of a supervisor.

To start using them, we can create a new project using **mix** and a special flag:

```
$ ~/dev > mix new sup_sample --sup
```

This command creates a new project that starts with a supervisor. The differences are in **mix.exs**, in which we have a specific application that should start, and the presence of **application.ex** as the entry point of the application:

```
defmodule SupSample.Application do
  # See https://hexdocs.pm/elixir/Application.html
  # for more information on OTP Applications
  @moduledoc false

  use Application

  def start(_type, _args) do
    # List all child processes to be supervised
    children = [
      # Starts a worker by calling: SupSample.Worker.start_link(arg)
      # {SupSample.Worker, arg},
    ]

    # See https://hexdocs.pm/elixir/Supervisor.html
    # for other strategies and supported options
    opts = [strategy: :one_for_one, name: SupSample.Supervisor]
    Supervisor.start_link(children, opts)
  end
end
```

Application.ex uses the **Application** module to create the context of the application. Applications in Elixir (and Erlang) are like libraries or packages in other languages, but since Elixir is a multi-process platform, every library that you use is actually an application (a process by itself).

The interesting part here is the **Supervisor.start_link** function. The **Application** module actually acts as a supervisor, and the processes it supervises are in the list of **children** specified in the **start_link** function. In the options we specify the restart strategy, which can be one of the following:

- **:one_for_one**: If a child process terminates, only that process is restarted
- **:one_for_all**: If a child process terminates, all other child processes are terminated, and then all child processes (including the terminated one) are restarted
- **:rest_for_one**: If a child process terminates, the terminated child process and the rest of the children started after it, are terminated and restarted

To see the supervisor in action, let's add a new module to this project. In the **/lib/sup_sample** folder, we create a new file called **my_server.ex** that contains this code:

www.dbooks.org

```
defmodule MyServer do
  use GenServer

  def init(_) do
   {:ok, []}
  end

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

  def ping do
    GenServer.call(__MODULE__, :ping)
  end

  def handle_call(:ping, _from, state) do
    {:reply, :pong, state}
  end
end
```

Here we have a simple **GenServer** that responds to a **ping** function with a **:pong**.

After this, we have to change the **application.ex** to specify which server to start:

```
defmodule SupSample.Application do
  use Application

  def start(_type, _args) do
    children = [
      {MyServer, []}
    ]
    opts = [strategy: :one_for_one, name: SupSample.Supervisor]
    Supervisor.start_link(children, opts)
  end
end
```

The **children** list contains the name of the module to start and the parameters.

To check that everything is wired correctly, we can use the REPL, specifying the application to start:

```
$ ~/dev > iex -S mix
Erlang/OTP 21 [erts-10.1] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-
threads:1] [hipe] [dtrace]

Compiling 1 file (.ex)
Interactive Elixir (1.7.3) - press Ctrl+C to exit (type h() ENTER for
help)
iex(1)>
```

The **-S** mix arguments tell the REPL to start the application using the specifications in the **mix.exs** file. In our case, it started the application supervisor and the server **MyServer**.

We can call it using the REPL:

```
iex(1)> MyServer.ping
:pong
```

Observer

Inside the REPL, we can use an Erlang application to inspect in the virtual machine and understand which processes are alive, and what they are doing. This application is called Observer. It is started with this instruction:

```
iex(2)> :observer.start
```

A new window opens:

www.dbooks.org

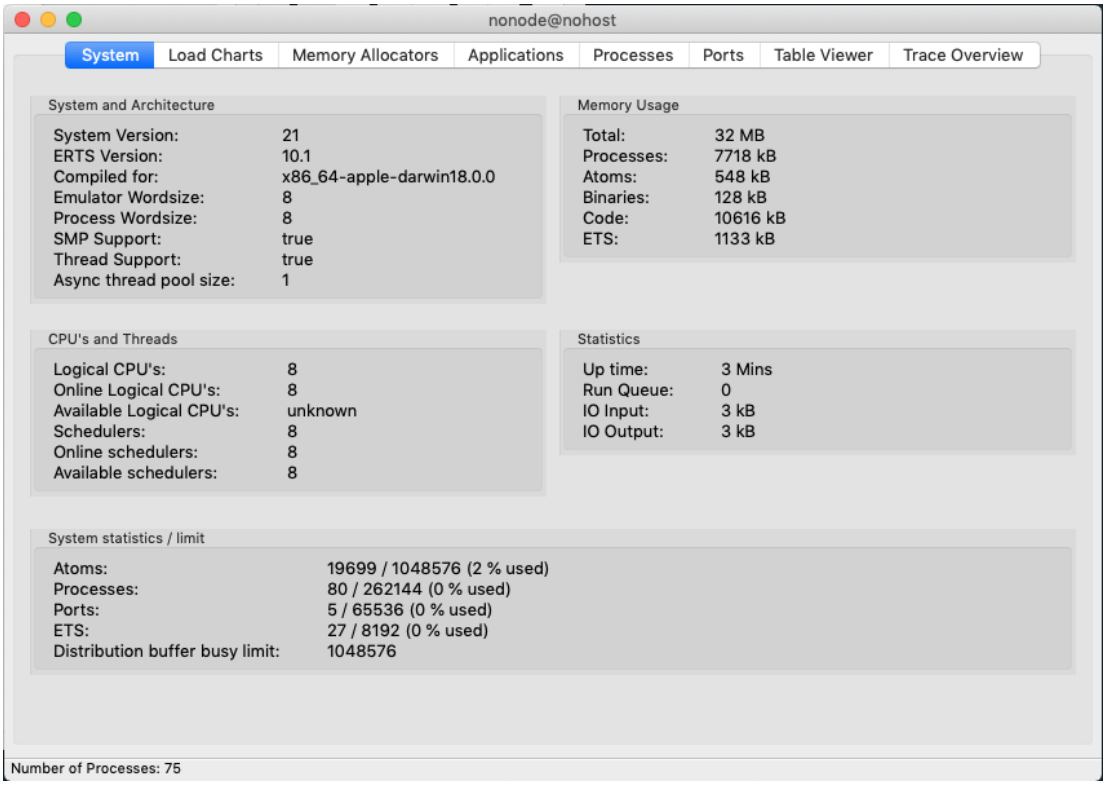


Figure 2 - Observer

It is a real desktop application with different tabs to inspect various sections of the current virtual machine. For now, we will investigate the Applications tab to see which applications are active, and to see how they are composed:

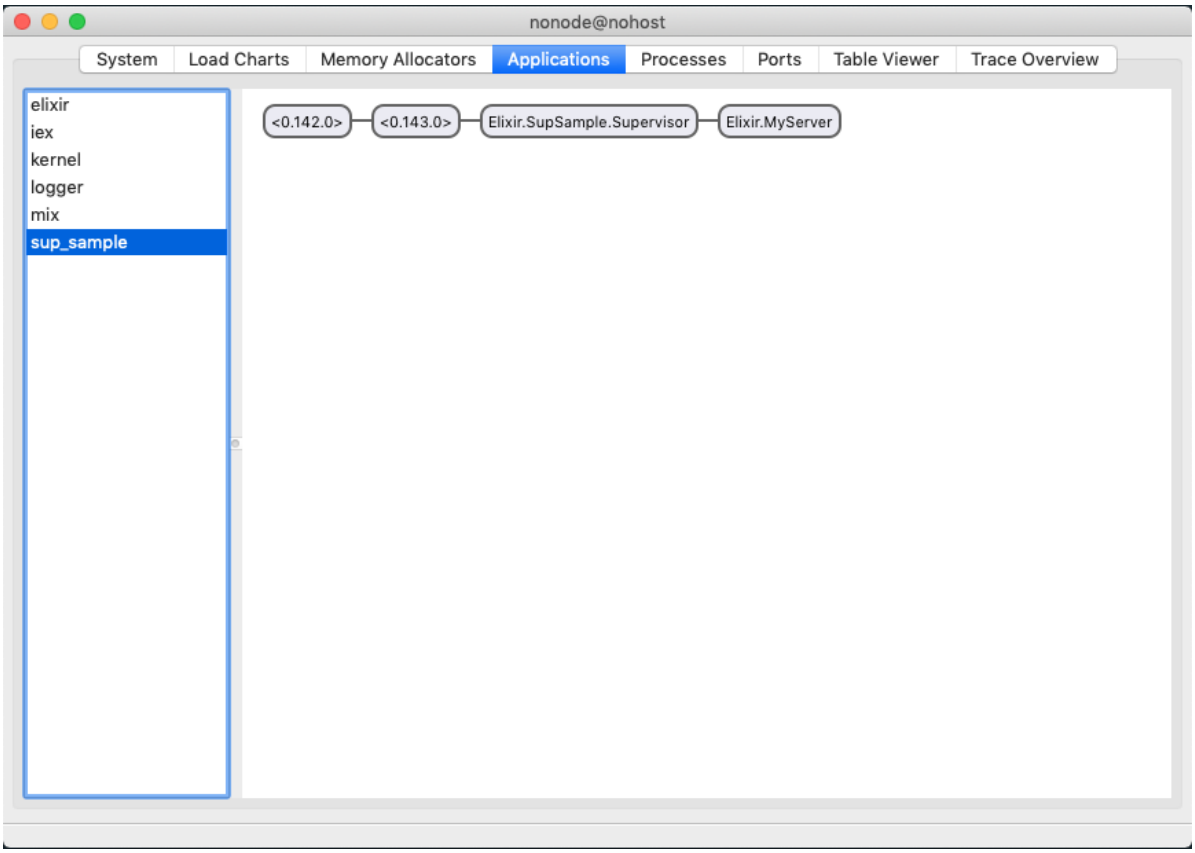


Figure 3 – Observer application tab

What we can see here is our application inside the REPL. The two important things here are the **`SupSample.Supervisor`** and **`MyServer`** that are linked together.

www.dbooks.org

If we select **`MyServer`**, we can see its PID in the status bar, which we can double-click to view more information about the process:

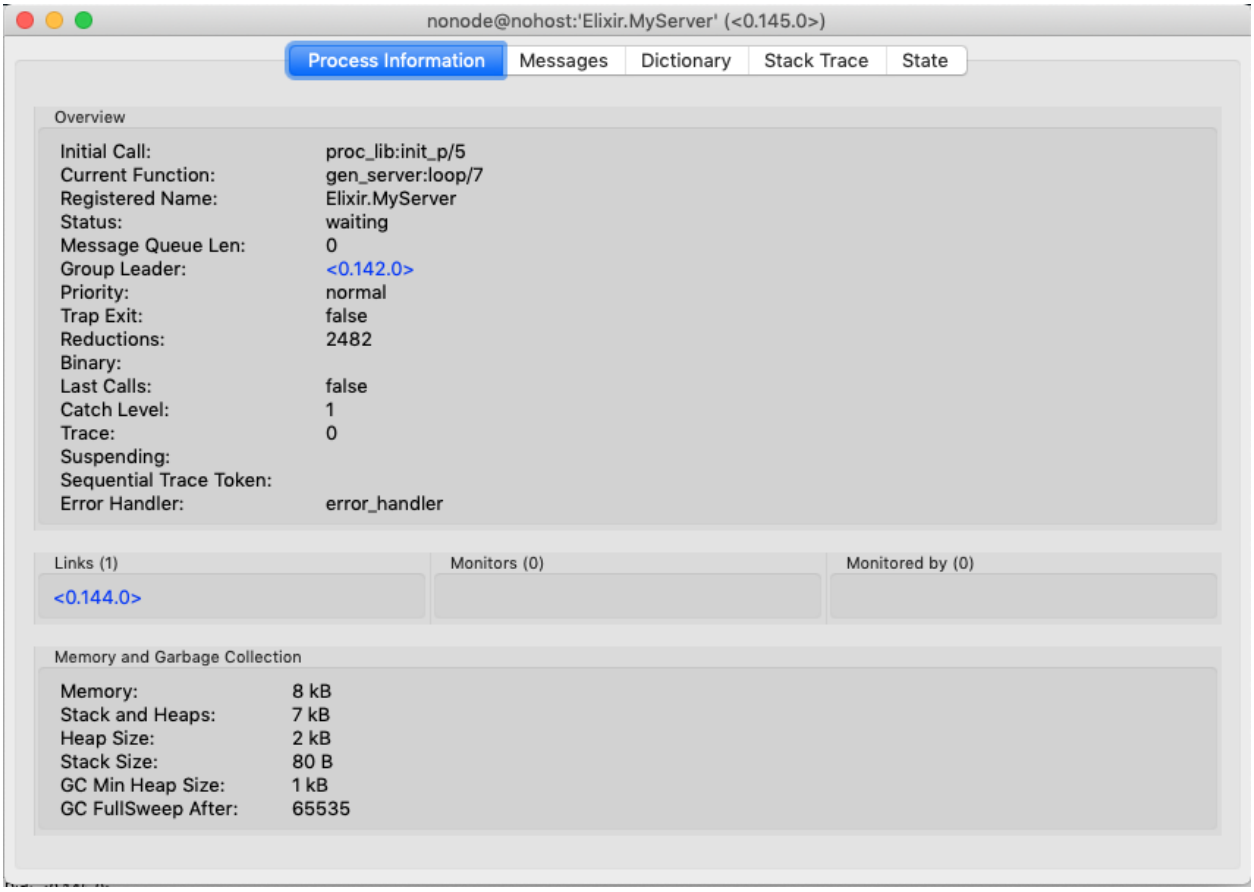


Figure 4 – Observer process details

Observer is a great tool for understanding how our application in configured, and how the processes are linked together. We will see more about Observer in the next chapter.

To see how the restart policy works, we will add a new function to the server to make it crash:

```
defmodule MyServer do
  use GenServer

  def init(_) do
   {:ok, []}
  end

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

  def ping do
    GenServer.call(__MODULE__, :ping)
  end

  def crash do
    GenServer.call(__MODULE__, :crash)
  end

  def handle_call(:ping, _from, state) do
    {:reply, :pong, state}
  end

  def handle_call(:crash, _from, state) do
    throw "Bang!"
    {:reply, :error, state}
  end
end
```

We simply add a new function and its callback to simulate a crash in the server. The **`crash`** function calls the **`handle_call`** callback that throws an error: **`Bang!`**.

As in other programming languages, if the error is not managed, it crashes the entire program. Let's see what happens here. Open the REPL and execute the **`crash`** function:

www.dbooks.org

```
$ ~/dev > iex -S mix
Erlang/OTP 21 [erts-10.1] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-
threads:1] [hipe] [dtrace]

Compiling 1 file (.ex)
Interactive Elixir (1.7.3) - press Ctrl+C to exit (type h() ENTER for
help)
iex(1)> GenServer.call(MyServer, :ping)
:pong
iex(2)> GenServer.call(MyServer, :crash)

16:49:43.466 [error] GenServer MyServer terminating
** (stop) bad return value: "Bang!"
Last message (from #PID<0.146.0>): :crash
State: []
Client #PID<0.146.0> is alive

    (stdlib) gen.erl:169: :gen.do_call/4
    (elixir) lib/gen_server.ex:921: GenServer.call/3
    (stdlib) erl_eval.erl:680: :erl_eval.do_apply/6
    (elixir) src/elixir.erl:265: :elixir.eval_forms/4
    (iex) lib/iex/evaluator.ex:249: IEx.Evaluator.handle_eval/5
    (iex) lib/iex/evaluator.ex:229: IEx.Evaluator.do_eval/3
    (iex) lib/iex/evaluator.ex:207: IEx.Evaluator.eval/3
    (iex) lib/iex/evaluator.ex:94: IEx.Evaluator.loop/1
** (exit) exited in: GenServer.call(MyServer, :crash, 5000)
    ** (EXIT) bad return value: "Bang!"
    (elixir) lib/gen_server.ex:924: GenServer.call/3
iex(2)> GenServer.call(MyServer, :ping)
:pong
iex(3)>
```

We started the REPL and called the **ping** function to check that **MyServer** is up and running. Then we called the **crash** function, and the error (in red) is shown with all the stack. But if we call the **ping** after the error, **:pong** is returned as if nothing has happened.

This is the magic of the supervisor and the restart policy. We configured the supervisor to restart the process if something bad happens, and it did exactly that.

In fact, the process after the crash is not the same; it's another process with another **pid** (we can inspect it using Observer), but from the point of view of the user, nothing has changed. This is one of the powerful features of Elixir (and Erlang). As a developer, you don't need to evaluate all the possible errors, catch all possible exceptions, and figure out what to do in every possible error case. The philosophy is "let it crash" and let the virtual machine do the work for you: restart the process as if nothing has happened.

There are a couple of things to consider:

- The state of the process is going to be reset at the initial state when the process restarts
- The messages in the inbox are lost: when the process crashes, the inbox crashes with the process

These two facts complicate things a bit…but not by much.

If your process is stateful and needs to restore the state in case of a crash, we can dump the state just before exiting the process. Elixir gives us a special callback that is called just before the process is about to exit. In this callback we have access to the state, and we can serialize it to a database or a file, or something that can be read as soon as the process restarts.

We can add this function to **`MyServer`** in our example:

```
def terminate(reason, state) do
  # dump the state
end
```

For the messages that are going to be lost in the inbox, we should write servers that are fast enough to keep the inbox empty as much as possible. Also consider the fact the supervisor can be hierarchical, and one supervisor can have other supervisors as children, and multiple servers, too. With this pattern we can scale to infinity, having big batteries of supervisors and servers to manage all the messages that arrives.

www.dbooks.org

Chapter 3 Sample Application

So far, we've had a quick look at most of the features of Elixir, we built some examples and played with the REPL to learn the language, and we wrote a sample application using **`mix`** and the supervisors.

The best way to learn a new language is to use it in a real-world application. In this chapter we will see how to build a complete web app, starting from scratch and using some libraries from the Elixir community.

Most of us probably know the Mars Rover Kata. It is an exercise that asks the developer to build a simulator of a simple rover that receives commands to move around the planet (Mars, in this case). The rover receives the commands in the form of characters and returns its position according to the execution of the commands.

The rules are:

You are given the initial starting point (x, y) of a rover and the direction (N, S, E, W) it is facing. The rover receives a character array of commands. Implement commands that move the rover forward/backward (f, b) and turn the rover left/right (l, r). Implement wrapping from one edge of the grid to another (planets are spheres, after all). Implement obstacle detection before each move to a new square. If a given sequence of commands encounters an obstacle, the rover moves up to the last possible point and reports the obstacle.

We start from this idea and expand it a little bit to create an online, multiplayer, real-time game!

What we want to create is a multiplayer game that runs in the browser. When a new user connects to the website and chooses the name for the rover, they can move it around the planet. The goal is to crash against another rover to destroy it, and the more rovers the user destroys, the more points they receive.

We will start architecting the application in terms of **`GenServer`**s and supervisors, and will start from the core rover component. Then we will add the web server to interact with the rover, and a web socket to notify the users.

The user interface is not part of the project; even in the repository of the project, we can see the full working example with the complete Javascript/HTML/CSS user interface.

Let's start create a new project called **`rovex`**:

```
$ ~/dev/> mix new rovex
```

As we already seen, this command creates the project structure needed to develop a full working project.

The application must be multiplayer, meaning we must have multiple live rovers on the planet, and for every new user that tries to connect and use the application, we need to create a new living rover.

In terms of Elixir, this means that every rover is a **GenServer** that is created as soon as a new user arrives, and it will receive the commands from this user.

These servers will have a private state: the position x and y, and the facing direction. This state can be changed by applying commands received from the user.

The first test

We'll start from the rover server, and since we are real developers, we start with a test. Let's edit the file **rover_test.exs** inside the folder **test**:

```
defmodule RoverTest do
  use ExUnit.Case

  test "get_state should return current state" do
    {:ok, _} = Rover.start_link({9, 9, :N, "rover0"})
    {:ok, state} = Rover.get_state("rover0")
    assert state == {9, 9, :N, 0}
  end
end
```

This is the first test we've seen in Elixir, but the syntax is the same as other testing tools, so it shouldn't scare you.

We include the **ExUnit.Case** library to be able to use its functions. We start the **Rover** server by calling the **start_link** function, passing the initial state, which is a tuple that contains two coordinates: the facing direction and the rover name.

```
{x, y, direction, name} = {9, 9, :N, "rover0"}
```

The test states that if we start a server with that initial state and we don't send commands, the state of the server is retrieved using the function **get_state,** and should not change. These tests are used to design the interface of the **Rover** module.

If we run the test now, we obtain an error, since the **Rover** module doesn't exist:

www.dbooks.org

```
$ mix test
Compiling 2 files (.ex)
Generated rovex app

== Compilation error in file test/rover_test.exs ==
** (CompileError) test/rover_test.exs:3: module Rover is not loaded and
could not be found
    (ex_unit) expanding macro: ExUnit.DocTest.doctest/1
    test/rover_test.exs:3: RoverTest (module)
    (elixir) lib/code.ex:767: Code.require_file/2
    (elixir) lib/kernel/parallel_compiler.ex:209: anonymous fn/4 in
Kernel.ParallelCompiler.spawn_workers/6
```

We have to make this test green. We add the **Rover** module:

```
defmodule Rover do
end
```

Run the test again:

```
$ mix test
Compiling 1 file (.ex)
Generated rovex app
..

  1) test get_state should return current state (RoverTest)
     test/rover_test.exs:5
     ** (UndefinedFunctionError) function Rover.start_link/1 is undefined
or private
     code: {:ok, _} = Rover.start_link({9,   9, :N, "rover0"})
     stacktrace:
       (rovex) Rover.start_link({9, 9, :N, "rover0"})
       test/rover_test.exs:6: (test)



Finished in 0.09 seconds
1 doctest, 2 tests, 1 failure

Randomized with seed 200712
```

Now it says that the **start_link** function does not exist. We must convert this module to a **GenServer** with its own callbacks:

```
defmodule Rover do
  use GenServer

  def start_link({x, y, d, name}) do
    GenServer.start_link(__MODULE__, {x, y, d, name}, name:
String.to_atom(name))
  end

  def init({x, y, d, name}) do
   {:ok, %{}}
  end

end
```

The **GenServer** has the **start_link** function that deconstruct the tuple that receive and calls the **Genserver.start_link** function to run the server.

The third parameter of the **start_link** function is the server name. For now, we convert the name passed (a string) into an atom. **GenServer** names can be an atom or a special tuple that we will see later.

In this call we are using the **RegistryHelper** module to have a unique name for the server. For now, consider the **RegistryHelper** a sort of map of all the active rovers, where the key of the map is the rover name. The **create_key** function returns a special tuple that can be used as the server name.

The **init** function should return the initial state; for now we have used an empty map.

Let's run the test again:

www.dbooks.org

```
$ mix test
Compiling 1 file (.ex)
..

  1) test get_state should return current state (RoverTest)
     test/rover_test.exs:5
     ** (UndefinedFunctionError) function RegistryHelper.create_key/1 is
undefined (module RegistryHelper is not available)
     code: {:ok, _} = Rover.start_link({9,   9, :N, "rover0"})
     stacktrace:
       RegistryHelper.create_key("rover0")
       (rovex) lib/rover.ex:7: Rover.start_link/1
       test/rover_test.exs:6: (test)



Finished in 0.03 seconds
1 doctest, 2 tests, 1 failure

Randomized with seed 222211
```

Now the problem is that the **get_state** function is missing. Let's change the server to add the **get_state** function and modify the **init** function to return the initial state:

```
defmodule Rover do
  use GenServer

  defstruct [:x, :y, :direction, :name]

  def start_link({x, y, d, name}) do
    GenServer.start_link(__MODULE__, {x, y, d, name}, name:
String.to_atom(name))
  end

  def init({x, y, d, name}) do
    {:ok, %Rover{x: x, y: y, direction: d, name: name }}
  end

  def get_state(name) do
    GenServer.call(String.to_atom(name), :get_state)
  end

  def handle_call(:get_state, _from, state) do
    {:reply, {:ok, {state.x, state.y, state.direction}}, state}
  end

end
```

There are a few new things here. First of all, we defined a **`struct`** to contains the state of the server; using a **`struct`** is a better way to manage the type and to enforce the shape of the state.

The **`init`** function now returns a **`struct`** with the values it receives from the caller, so the **`state`** is correctly initialized and passed to the callbacks that need the state.

The **`get_state`** function receives the name of the rover whose state is requested. Remember that we are in a context where multiple instances of this rover are live, and since Elixir is a functional language, the way to specify which rover is asked is to pass the name to the functions.

The **`get_state`** simply sends a call to the server, and **`handle_call`** is called. As we've already seen, the **`handle_call`** callback receives the state of the server, and it can use this state to create the response:

```
{:reply, {:ok, {state.x, state.y, state.direction}}, state}
```

Now the first test should be green:

```
$ mix test
...

Finished in 0.04 seconds
1 doctest, 2 tests, 0 failures

Randomized with seed 246068
```

We have now created a module that runs the rover process and is able to return its state. It's not much, but it's a start.

Control the rover

Now that we've created a rover, let's add more features, like movement and rotation. We'll start with the movement, adding a **`go_forward`** function to move the rover. We are opting for high-level function; something else will manage the conversion from letters to actions (such as **`F`** for moving forward, and **`L`** to turn left).

The test could be:

www.dbooks.org

```
test "handle_cast :go_forward should return updated state" do
  {:noreply, state} = Rover.handle_cast(:go_forward, %Rover{x: 1, y: 3,
direction: :N})
  assert state.x == 1
  assert state.y == 4
  assert state.direction == :N
end
```

Now that we know our rover is a **GenServer**, we can directly test the server function. This gives us the ability to write simple unit tests that don't need a real running server to be executed.

The previous test directly calls the **handle_cast** (the asynchronous version of **handle_call**), passing the action to be called (**:go_forward**) and the current state (second parameter). The result of this call should be the new state (position).

To make it pass, we can just add the **handle_cast** function, but for completeness, we will also add the public function **go_forward**.

```
# inside rover.ex
# [...previous code]

@world_width 100
@world_height 100


def go_forward(name) do
  GenServer.cast(RegistryHelper.create_key(name), :go_forward)
end

def handle_cast(:go_forward, state) do
  new_state = case state.direction do
    :N -> %Rover{ state | x: state.x, y: mod(state.y + 1, @world_height) }
    :S -> %Rover{ state | x: state.x, y: mod(state.y - 1, @world_height) }
    :E -> %Rover{ state | x: mod(state.x + 1, @world_width), y: state.y }
    :W -> %Rover{ state | x: mod(state.x - 1, @world_width), y: state.y }
  end

 {:noreply, new_state}
end
```

The implementation is quite easy, thanks to pattern matching. Based on the rover direction, we decide how to move the rover. The case switchs over the direction of the rover and, depending on the direction, builds the new state.

In the previous code, **@world_height** and **@world_width** are constants, and their values are declared at the beginning of the snippet. The **mod** function returns the module, and it is used to consider the wrap around the planet.

The use of pipe operators in structs and maps is a shortcut to merge the right side with the left side. The following operation returns a new **Rover** structure that brings **state** and changes the **x** and **y** to the new specified values.

```
%Rover{ state | x: state.x, y: mod(state.y + 1, @world_height) }
```

The **go_backward** operation is very similar to this one, and we omit the code for brevity.

The rotation functions are also quite similar; we'll use the same pattern.

Starting from the test:

```
test "handle_cast :rotate_left should return updated state (N)" do
  {:noreply, state} = Rover.handle_cast(:rotate_left, %Rover{x: 1, y: 3,
direction: :N})
  assert state.x == 1
  assert state.y == 3
  assert state.direction == :W
end
```

If the robot direction is north (**N**) and we rotate left, the new direction must be west (**W**), while the coordinates should be the same.

Now we know how to implement this:

```
def rotate_left(name) do
  GenServer.cast(RegistryHelper.create_key(name), :rotate_left)
end

def handle_cast(:rotate_left, state) do
  new_state =
    case state.direction do
      :N -> %Rover{state | direction: :W}
      :S -> %Rover{state | direction: :E}
      :E -> %Rover{state | direction: :N}
      :W -> %Rover{state | direction: :S}
    end
  {:noreply, new_state}
end
```

What changes after a rotation is just the direction.

We now have all the logic needed to move the rover around the planet.

www.dbooks.org

Rover supervisor

The application requires a new robot to be created when a new user starts to play. The role of creating new rovers is the task for a supervisor.

We already see that supervisor acts as a life controller for a **GenServer**, and we saw that we must specify the list of children to create when the supervisor starts.

This case is a little bit different. We don't know how many **GenServer**s the supervisor will have to create at the beginning, since rovers come to life by request and die when a crash occurs. The component to do such things is a **DynamicSupervisor**, a supervisor that doesn't have a predefined list of children. Instead, children can be added or removed when it is necessary.

This means that this supervisor starts empty (with no children), and when a new user connects to the application, the supervisor will be called to create a new rover:

```
defmodule RoverSupervisor do
  use DynamicSupervisor

  def start_link(_) do
    DynamicSupervisor.start_link(__MODULE__, [], name: __MODULE__)
  end

  def init(_) do
    DynamicSupervisor.init(strategy: :one_for_one)
  end

  def create_rover(name, x, y, direction) do
    DynamicSupervisor.start_child(__MODULE__, {Rover, {x, y, direction,
name}})
  end

  def kill(rover) do
    pid = RegistryHelper.get_pid(rover.name)
    DynamicSupervisor.terminate_child(__MODULE__, pid)
  end
end
```

The two interesting functions here are **create_rover** and **kill**. The first is used to create a new rover and add it to the list. This function calls the **start_child** on the base **DynamicSupervisor**, passing the reference to the supervisor itself (the **__MODULE__** macro) and the specifications to define the new process: the **Rover** module and the initial state (x,y, direction and name).

The second function is **kill**, and it is used to terminate a process. This function will be called when a crash occurs. It gets the **pid** of the process (using the **Registry**) and calls **terminate_child** to kill the process.

With just these few modules, we have what we need to start the rovers.

Open the REPL using this command:

```
$ > iex -S mix
```

This command starts the REPL and loads the application defined in the **mix** file so that we can use the modules defined in the app.

We can run the supervisor and create new rovers:

```
Erlang/OTP 21 [erts-10.1] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-
threads:1] [hipe] [dtrace]

Compiling 1 file (.ex)
Generated rovex app
Interactive Elixir (1.7.3) - press Ctrl+C to exit (type h() ENTER for
help)
iex(1)> RoverSupervisor.start_link([])
{:ok, #PID<0.146.0>}
iex(2)> RoverSupervisor.create_rover("foo", 1, 2, :N)
{:ok, #PID<0.148.0>}
iex(3)> RoverSupervisor.create_rover("bar", 1, 2, :N)
{:ok, #PID<0.5127.0>}
iex(4)> Rover.go_forwar("foo")
:ok
iex(4)> Rover.get_state("foo")
{:ok, {1, 3, :N}}
```

We must change the **mix** file to inform the runtime, which is the application that it has to start.

Let's add this line to **mix.exs**:

```
def application do
    [
      extra_applications: [:logger],
      mod: {Rover.Application, []}
    ]
end
```

The application function returns the list of applications to execute and the main module to start. For now, we just return the **Rover.Application** module (which we'll see shortly) and the default **logger**.

www.dbooks.org

The **Application** module is this:

```
defmodule Rover.Application do
  use Application

  def start(_type, _args) do

    children = [
      Supervisor.child_spec({Registry, [keys: :unique, name:
Rover.Registry]}, id: :rover_registry),
      Supervisor.child_spec({RoverSupervisor, []}, id: RoverSupervisor),
    ]

    opts = [strategy: :one_for_one, name: Application.Supervisor]
    Supervisor.start_link(children, opts)
  end
end
```

The **application.ex** module is the entry point for your application. It is a supervisor, and starts the children defined in the list. For now, we just have a couple of processes: the **Registry** that stores the running rovers, and the **RoverSupervisor** used to create and kill rovers.

When we start the REPL with this file in place, the application module is executed automatically and **Registry** and **RoverSupervisor** start at the beginning.

Now we can start creating the new rover.

If we run Observer, we can start viewing our three processes:

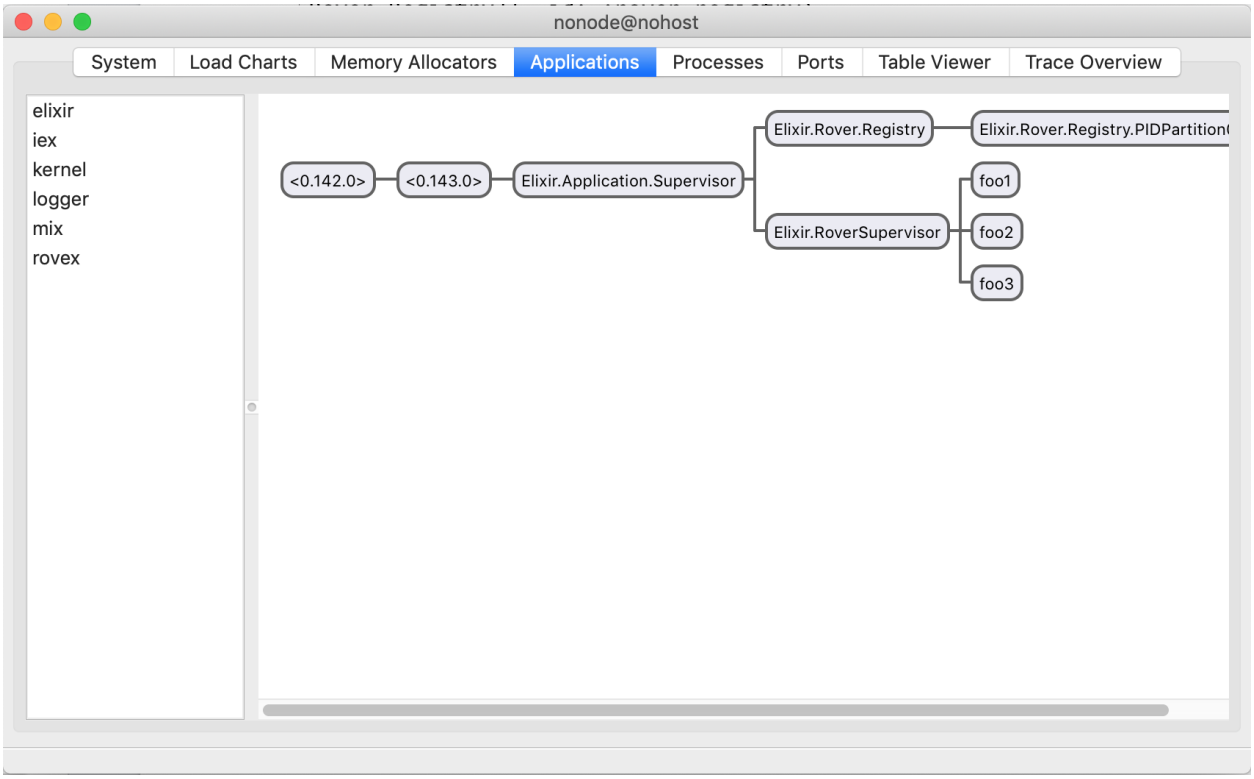


Figure 5 – Rovex processes

Notice the three rovers created using the **`create_rover`** function in the REPL, the two supervisors, and the **`Application`** supervisor.

The basic blocks of our application are now ready, and we can start adding some API to interact with the rover using the HTTP protocol.

This is when Plug comes in.

Introducing Plug and Cowboy

According to the Plug website, Plug is defined as:

1. *A specification for composable modules between web applications*
2. *Connection adapters for different web servers in the Erlang VM*

In practice we can consider it a sort of library used to write web API. Plug is not a fully web-based framework; it needs a web server to run (usually Cowboy), and is very simple to set up and use.

www.dbooks.org

The web server Cowboy is an implementation (in Erlang) of the HTTP protocol, and is used by Plug as the lower layer.

To understand how it works, we will start using it in our application to accept commands via HTTP. First of all, we add the dependencies to our **`mix`** file:

```
 {:plug, "~> 1.5"},
 {:cowboy, "~> 1.0"},
```

Then in the same file, we add two extra applications: **`:cowboy`** and **`:plug`**. We already see that Elixir applications are composed of different applications, and Cowboy with Plug are two of them. They are not libraries in the common sense (like a NPM package); they run on their own, with their own processes.

Finally, we have to add to our list of main supervisor children the Plug/Cowboy supervisor that starts and listens to HTTP requests. It takes a few parameters: the protocol (**`:http`** in this case), the module that will receive the requests, a set of Cowboy options (empty array), and the port to listen to (**`3000`**):

```
defmodule Rover.Application do
  use Application

  def start(_type, _args) do

    children = [
      Supervisor.child_spec({Registry, [keys: :unique, name:
Rover.Registry]}, id: :rover_registry),
      Supervisor.child_spec({RoverSupervisor, []}, id: RoverSupervisor),
      Plug.Adapters.Cowboy.child_spec(:http, Rover.Web.Router, [], port:
3000),
    ]

    opts = [strategy: :one_for_one, name: Application.Supervisor]
    Supervisor.start_link(children, opts)
  end

end
```

This is what we need to configure the application to start a web server on port **`3000`** and start waiting for an incoming connection.

Now we need to add a module to manage these connections.

```
defmodule Rover.Web.Router do
  use Plug.Router

  plug(Plug.Parsers, parsers: [:json], json_decoder: Poison)
  plug(:match)
  plug(:dispatch)

  get "/ping" do
    send_resp(conn, 200, Poison.encode!(%{message: "pong"}))
  end

  match(_) do
    send_resp(conn, 404, "")
  end
end
```

This is a minimal implementation of a web module. First of all, it defines a couple of plugs. (Plugs are a kind of middleware, in node terms.)

The first plug is used to parse JSON content, and is based on the Poison library. What it does is simply transform JSON content into Elixir maps so that we can use them in the request handlers.

The second plug is used to determine which route (and then which function) matches the requested route. The last plug is used to tell **`plug`** to dispatch the request to the specified function.

Our handler **`get`** matches against the route **`/ping`** and the HTTP method **`GET`**, and what it does is to send the response **`{"message": "pong"}`** with status code **`200`**. To convert the response into JSON, it uses the **`encode`** function of the Poison library.

The **`match`** function at the end of the file is used to respond **`404 Not Found`** to every other request.

We can see if everything works by running the server using **`mix run --no-halt`** and using curl or another HTTP client to open the URL http://localhost:3000/ping and verify the response.

If everything is correct, we continue by adding some more route handlers. We will start with the route that creates a new rover. It will be a **`POST`** to the route **`/rover`**:

www.dbooks.org

```
post "/rover" do
  rover_name = conn.body_params["name"]
  x = conn.body_params["x"]
  y = conn.body_params["y"]
  d = String.to_atom(conn.body_params["d"])

  case RoverSupervisor.create_rover(name, x, y, d) do
    {:ok, _} -> send_resp(conn, 201, encode(%{message: "created rover
#{rover_name}"}))
    {:error, {:already_started, _}} -> send_resp(conn, 400,
encode(%{message: "rover already exists"}))
    _ -> send_resp(conn, 500, encode(%{message: "generic error"}))
  end
end
```

Let's analyze the code. The **post** function declares a new route that respond to **/rover**. Then it reads data from the body request: **name, x, y, d**. The request is supposed to be in JSON, and **conn.body_params** is a sort of map of the posted JSON.

This is possible because at the beginning of the module, we add a plug module to parse the application/JSON request. Poison is the library used for parsing and generating JSON responses.

When the data needed to create a new rover is available, all we have to do is to call **RoverSupervisor.create_rover** to start the server, and based on the result, we build a correct response: **201** if the rover is correctly created, and an error in other cases.

If the result of **create_rover** is okay, the rover process is alive and ready to receive commands.

Now we can implement the route that receive commands:

```
post "/command" do
  rover_name = conn.body_params["name"]
  command = String.to_atom(conn.body_params["command"])
  Rover.send_command(rover_name, command)
  send_resp(conn, 204, encode(%{}))
end
```

As with the previous route, it is a **post** method. In the body, the client posts the rover name and the command (**F**, **B**, **L**, **R**). We send this command directly to the living rover.

Note: We are using a helper function on the rover to simplify the client, and using pattern matching to identify which real command we need to call.

```
# in Rover.ex

def send_command(name, :F) do
  Rover.go_forward(name)
end

def send_command(name, :B) do
  Rover.go_backward(name)
end

def send_command(name, :L) do
  Rover.rotate_left(name)
end

def send_command(name, :R) do
  Rover.rotate_right(name)
end
```

We now have a real application that can control rovers using HTTP commands. Nice, but not yet very useful.

At the beginning of this chapter, we said that we would going to build a multiplayer game where players should destroy other players' rovers by crashing against them.

Now every player can create a new rover and can move it, but there is not yet the possibility to know where the other rovers are and to crash against them.

To implement this feature, we start by creating a new type of **`GenServer`** that has the task of keeping track of currently created rovers and their positions. Every time a rover changes its position, this new **`GenServer`** checks for rovers overlapping, and if necessary, kills the rover.

Here is the source code of this **`GenServer`**, which we will call **`WorldMap`**:

www.dbooks.org

```
defmodule WorldMap do
  use GenServer

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

  def init([]) do
    {:ok, %{rovers: []}}
  end

  def update_rover(name, x, y) do
    GenServer.call(__MODULE__, {:update_rover, name, x, y})
  end

  defp update_rover_list(rovers, name, x, y) do
    case Enum.find_index(rovers, fn r -> r.name == name end) do
      nil -> rovers ++ [%{name: name, x: x, y: y}]
      index -> List.replace_at(rovers, index, %{name: name, x: x, y: y})
    end
  end

  def handle_call({:update_rover, name, x, y}, _from, state) do
    rover_list = update_rover_list(state.rovers, name, x, y)

    case Enum.find(rover_list, fn r -> r.name != name && r.x == x && r.y
== y end) do
      nil ->
        {:reply, :ok, %{state | rovers: rover_list}}
      rover_to_kill ->
        RoverSupervisor.kill(rover_to_kill)
        new_rovers = List.delete(rover_list, rover_to_kill)
        {:reply, :ok, %{state | rovers: new_rovers}}
    end
  end
end
```

The implementation is quite easy now; during the **`init`** phase, it prepares the rover list to an empty list. The only public function is **`update_rover`**, which every rover should call when its state changes. This function checks the **`WorldMap`** state if the rover is present. In this case it updates its position; otherwise, it adds the rover to the list.

Then it checks for collision if finding other rovers in the same position. If it finds one, it kills the rover using the **`RoverSupervisor.kill`** function, and removes the rover from the state.

We already saw the implementation of the **`kill`** function, which now has a real meaning in this context.

The **WorldMap GenServer** must be started when the application starts, so we have to go back to our application file and add the declaration of **WorldMap** to the children:

```
children = [
  Supervisor.child_spec({Registry, [keys: :unique, name: Rover.Registry]},
id: :rover_registry),
  Supervisor.child_spec({Registry, [keys: :duplicate, name:
Socket.Registry]}, id: :socket_registry),
  Plug.Adapters.Cowboy.child_spec(:http, Rover.Web.Router, [], port:
3000),
  Supervisor.child_spec({RoverSupervisor, []}, id: RoverSupervisor),
  Supervisor.child_spec({WorldMap, []}, []),
]
```

We also need to change the rover GenServer code to notify **WorldMap** every time something changes. It is straightforward to implement; we just need to add this line to the **handle_cast** function for **:go_forward** and **:go_backword**:

```
WorldMap.update_rover(state.name, state.x, state.y)
```

Now everything is more connected; rovers can destroy other rovers and they will be killed accordingly, but we still haven't notified players.

At this point, players don't know if their rover is about to die or to destroy another rover because communication are one-way only: players can control the rovers, but they are not notified about what happening on the server.

The best way to do this is to implement a WebSocket channel to notify the clients. Plug and Cowboy are already configured; we just need to add a module to manage the web socket communication, and then we will be ready to use it for notifying clients.

```
children = [
  # ...
  Plug.Adapters.Cowboy.child_spec(:http, Rover.Web.Router, [], port: 3000,
dispatch: dispatch()),
  # ...
]
defp dispatch do
  [
    {:_,
     [
       {"/ws", Rover.Web.WsServer, []},
       {:_, Plug.Adapters.Cowboy.Handler, {Rover.Web.Router, []}}
     ]}
  ]
end
```


www.dbooks.org

First, we need to configure Cowboy to use WebSockets. Back in the application file where we are starting the Plug/Cowboy process we need to change the configuration: The dispatch option is where we are telling cowboy that a new handler exists and it is in the **`WsServer`** module.

The **`WsServer`** module is the module that manage the WebSocket connections:

```
defmodule Rover.Web.WsServer do
  @behaviour :cowboy_websocket_handler
  @timeout 5 * 60_000
  @registration_key "ws_server"

  def init(_, _req, _opts) do
    {:upgrade, :protocol, :cowboy_websocket}
  end

  def send_message_to_client(_rover, message) do
    Rover.Application.dispatch("#{@registration_key}", message)
  end

  def websocket_init(_type, req, _opts) do
    #{rover, _} = :cowboy_req.qs_val(<<"rover">>, req)
    {:ok, _} = Registry.register(Socket.Registry, "#{@registration_key}",
[])
    state = %{}
    {:ok, req, state, @timeout}
  end

  def websocket_terminate(_reason, _req, _state) do
    :ok
  end
end
```

This piece of code is quite dense—Elixir doesn't have yet a library that manage web sockets, so Cowboy from Erlang must be used.

The two main functions here are **`websocket_init`**, which adds in the registry, the client connection using the rover name in the query string to identify the rover, and **`send_message_to_client`**, a function used to notify the connected clients.

This last function is used inside the rover process to notify the clients about the rover's position:

```
Rover.Web.WsServer.send_message_to_client(state.name, state)
```

The **`name`** is used to identify the sender, and the **`state`** will be used by the client to update the position.

This is all we need to create a WebSocket connection that notifies clients about changes to the rovers.

The UI of the application is out of scope of this book (since it will be written in Javascript and HTML), but if you want to see the full application you can find it on my GitHub at this address: GitHub.

There is also a completely functional version of this application, where we can play with our friends, available here. The backend code is exactly what you find on GitHub.

For a recap of the architecture, have a look at the following figure:

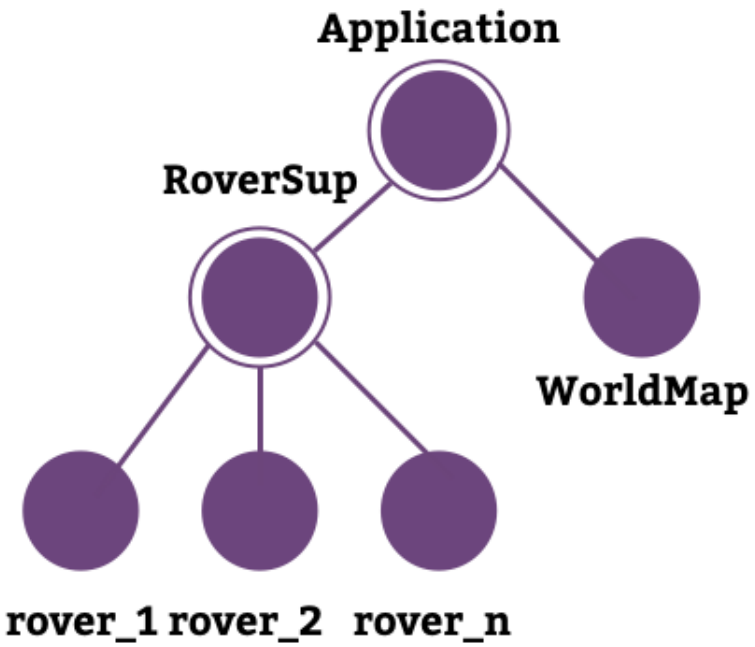


Figure 6 – Application servers and supervisors

The Application is the main supervisor that runs the **`WorldMap`** GenServer, and a second-level supervisor that supervises the rover processes. There can be many rover processes—on my machine (a MacBook pro with 16MB RAM) I simulated a world with more than 2,000 rovers, and everything worked like a charm, without any problems.

Rover communicates with **`WorldMap`**, sending status updates so that **`WorldMap`** is able to check for collisions and keep its map aligned with reality.

www.dbooks.org

We omitted the schema from infrastructure but Cowboy, Plug, and the registries are just another set of processes or supervisors.

We can look at them using Observer. Start the application using the command **`iex -S mix`**, and use the **`RoverSupervisor`** to create some rovers:

```
iex(1) > RoverSupervisor.create_rover("one", 1, 2, :N)
{:ok, #PID<0.3993.0>}
iex(2) > RoverSupervisor.create_rover("two", 4, 9, :N)
{:ok, #PID<0.3998.0>}
iex(3) > RoverSupervisor.create_rover("three", 24, 73, :N)
{:ok, #PID<0.4000.0>}
iex(4) > :observer.start
```

Using the Applications tab, we will see something like this:

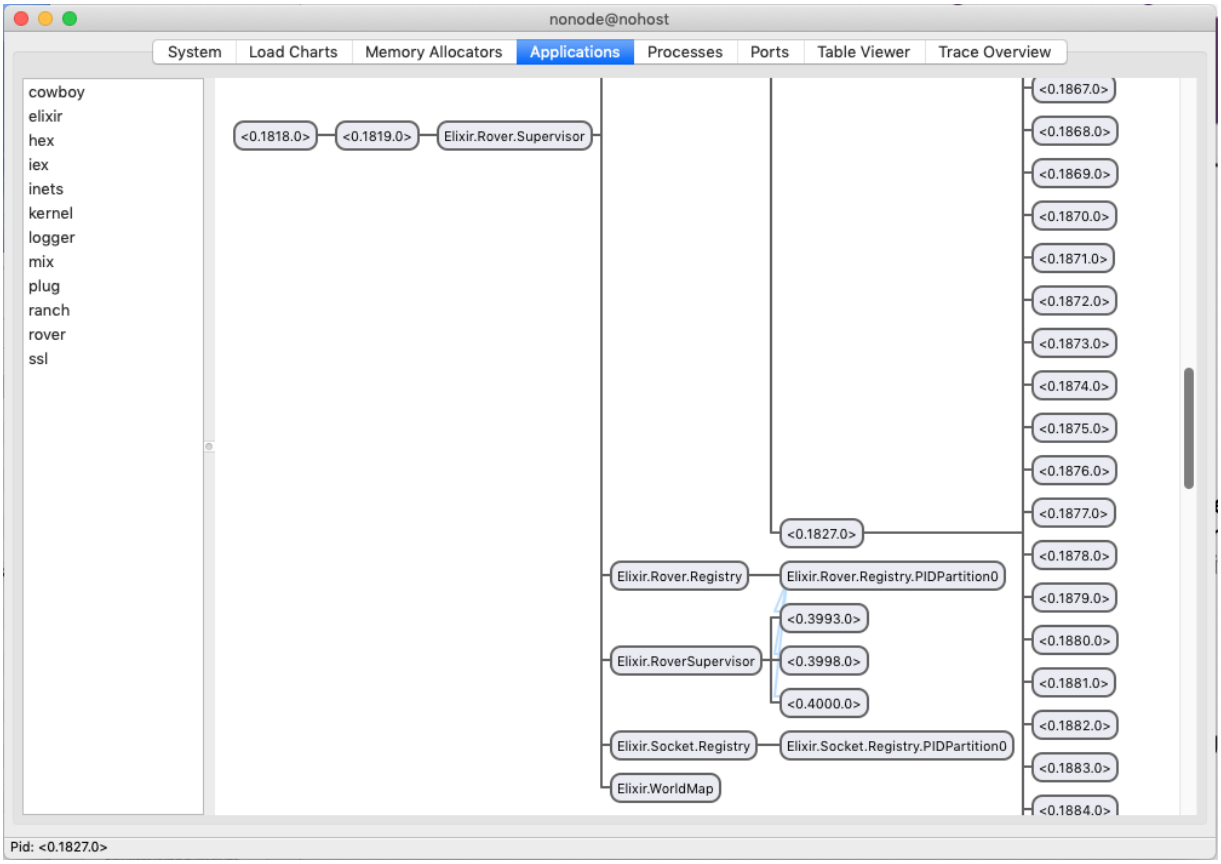


Figure 7 – Rovex process tree

We can see **`Elixir.RoverSupervisor`** with three children: the three rovers that we have created from the REPL.

There are the two registries: one for the rovers, and one for the web socket connection. The rover's registry is connected to the rover's instances with blue lines.

There is also the **`WorldMap`** that we can inspect double clicking on it to view its state:

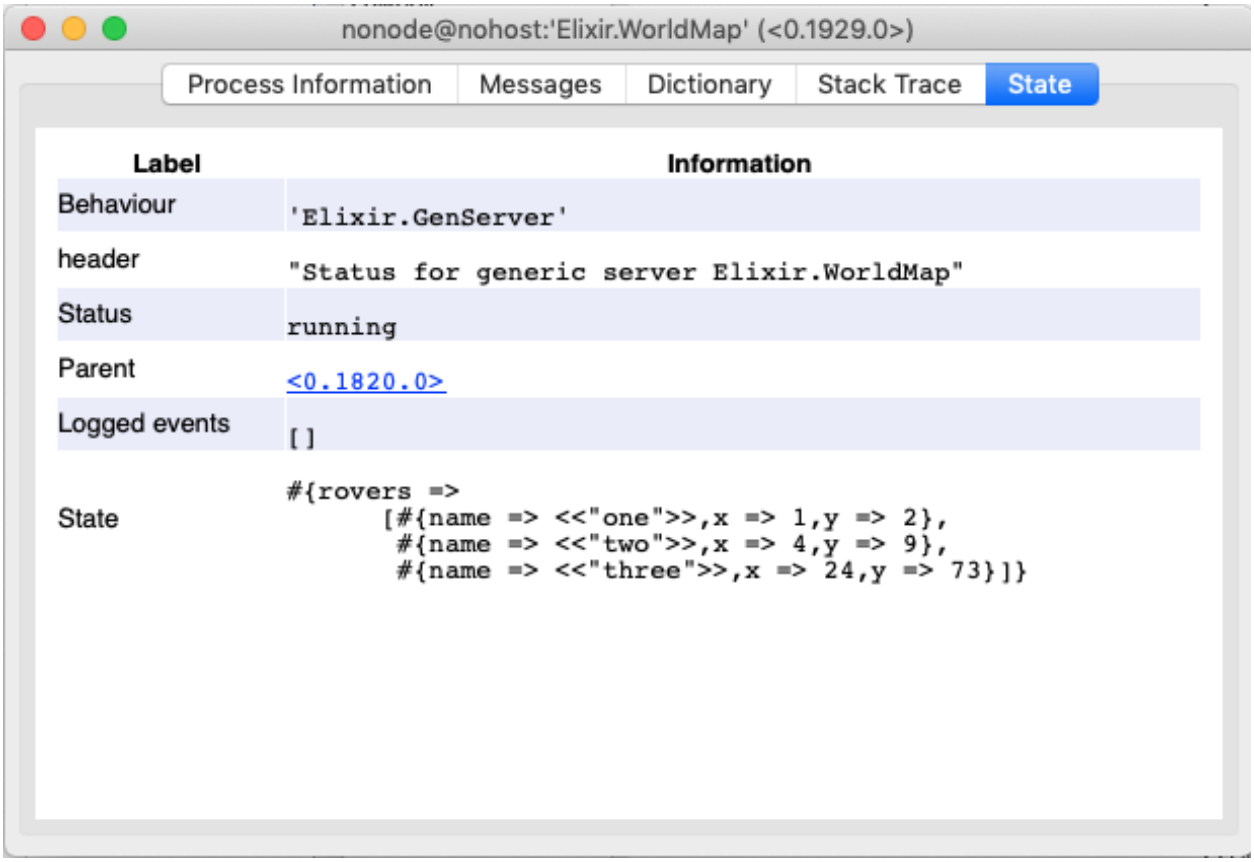


Figure 8 – Rovex WorldMap state

All other processes on the right (the long list) are processes owned by Cowboy, and most of them are HTTP connections ready to be used from the clients.

Our logical structure exactly matches the physical one shown by Observer.

www.dbooks.org

Conclusion

We have seen most of the features of Elixir and how to use them to build a real application. This book is not exhaustive, but surely gives you an idea of how Elixir works, and a look into the peculiarities of the language and the platform.

Even if Elixir was born eight years ago, it is still quite young (and the ecosystem is young) . I consider this a great moment to invest in Elixir: its youngness is what we need to grow a better platform for highly scalable web applications and services.

Having read this book, the best thing that you could do is to start creating a real project with Elixir to get more confidence with the language and the tools.

The community is very inclusive, you can find all the help you need on the Elixir Forum to resolve any issues you encounter during the first weeks.