The Elixir programming language: Reliable microservices made easy

Adam Kuffel
Lemonbeat GmbH, 44339 Dortmund
adam.kuffel@lemonbeat.com

Abstract

This paper describes the concepts and peculiarities of using the Elixir programming language for the development of reliable microservices. The author is working full-time with Elixir at Lemonbeat and is participating in the development of a distributed IoT solution based on Elixir in the backend. The goal of this paper is to share some knowledge regarding development of an application using Elixir.

1 Introduction

The Elixir programming language is a functional programming language which uses the actor model and runs on the Erlang VM. This allows applications to take the full advantages of the BEAM (Bogdans Erlang Abstract Machine) virtual machine. Elixir applications are compiled into Erlang bytecode and can use Erlang functions and modules within elixir code. [*]

The actor model was defined by Carl Hewitt in 1973 and describes software as a set of components that communicate with each other by exchanging messages. Using the actor model eases the development of distributed, concurrent and scalable systems by ensuring a loose coupling between the components. The developer is forced to develop the system as a collection of communicating processes which rely on messaging. [†]

Erlang was invented by Ericsson in 1987 and released as open source in 1998, it is still actively maintained by Ericsson. With its initial use-case for telecom switches, it is meant to be distributed, fault-tolerant, soft real-time and high available. The name OTP (Open telecom platform) was introduced in that time. Systems written in Erlang are meant to be always available and support hot-swapping of code to achieve updates without downtime. [‡]

[*] https://elixir-lang.org/
[†] http://worrydream.com/refs/Hewitt-ActorModel.pdf
[‡] http://erlang.org/download/armstrong_thesis_2003.pdf

Systems written in Erlang are powering high volume applications like the T-Mobile SMS and authentication services, the Facebook Chat service and the WhatsApp backend. Open source erlang applications like, e.g. CouchDB, Ejabberd and RabbitMQ are popular and actively used for mission critical components in backend applications. [§]

Erlang supports vertical scalability through native SMP support, modern computers multiple cores and need applications that can make use of them, Erlang supports this. [**]

2 Nodes, Applications and Processes

During runtime an Elixir application is just an Erlang Node, through the compilation of Elixir applications to Erlang Bytecode. A node is an instance of an Erlang virtual machine, which runs applications, that are a collection of running processes. Every application runs isolated and can call modules and function as if they were in the same application. [††]

Nodes can communicate with each other using the Erlang Distribution Protocol, a TCP protocol that provides low-level socket connections for exchanging messages. Distributing applications and processes is built into the platform and can be used to create applications that run on multiple physical computers.[‡‡]

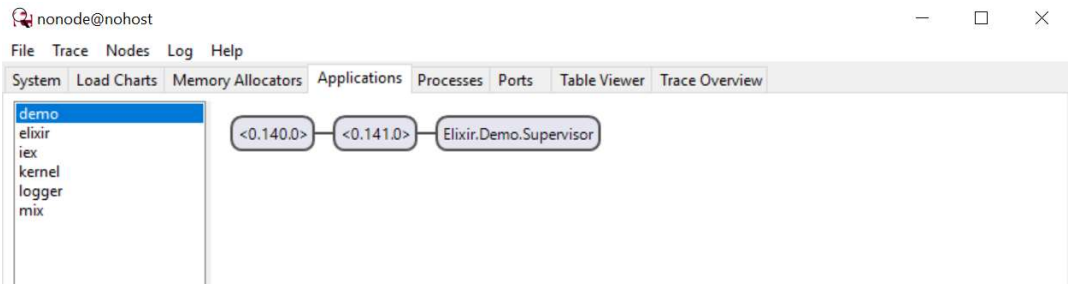

Figure 1 - Observer showing a running Node with applications on the left and processes on the right

As Applications are isolated they can be started and stopped separately. To ensure that dependent applications are started first, you can define them in the mix file of an elixir project. [§§]

Processes in Erlang are the most important concept in OTP, everything is a process, this allows concurrency and the distribution of work across all CPUs. Erlang processes are independent of operating system processes, they are lightweight, take microseconds to start and the Erlang VM supports up to 134 million of them. Processes use the actor concurrency model, this means each process is doing a specific task and processes communicate via messages. A process is identified by a PID, does not share information with other process and has its own mailbox for incoming messages. [***]

[§] (Armstrong, 2013) - https://gangrel.files.wordpress.com/2015/08/programming-erlang-2nd-edition.pdf
[**] (Herbert, 2013) - https://learnyousomeerlang.com/the-hitchhikers-guide-to-concurrency
[††] (Loder, 2016), p.94
[‡‡] (Tan Wei Hao, 2017), p.172
[§§] (Tan Wei Hao, 2017), p.42
[***] (Tan Wei Hao, 2017), p.39-40

2.1 Spawning processes

Spawning processes in Elixir can be done via the spawn function, the process does the work and removes itself when the desired function finishes. [†††]

This example from the console shows how to start an interactive elixir shell and spawn a process that prints something on the screen:

```
~$ iex
Interactive Elixir (1.7.3) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> pid = spawn(fn -> IO.puts("Hello Process!") end)
Hello Process!
#PID<0.87.0>
```

This code snippet shows how to implement communication between processes:

```
~$ iex
iex(1)> owner = self()
iex(2)> child = spawn(fn -> send current, {self(), "Hello other Process!"} end)
iex(3)> receive do
...(3)> {sender, message} -> IO.puts("#{inspect sender} => #{inspect message}")
...(3)> end
#PID<0.87.0> => Hello other Process!
```

2.2 Spawning processes on another Node

A key feature of Erlang applications is the ability to distribute your application across multiple Nodes. To connect two nodes, they need to use the same Erlang Cookie, Erlang version and must be able to resolve their names.

These commands start two nodes, connect them and the beta node spawns a process on the alpha node.

```
# On the first server
~$ iex --sname alpha --cookie secret
iex(alpha@server1)1>

# On the second server
~$ iex --sname beta --cookie secret
iex(beta@server2)1> Node.connect(:"alpha@server1")
true
iex(beta@server2)2> Node.list
[:"alpha@server1"]
iex(beta@server2)3> Node.spawn(:"alpha@server1",fn()-> IO.puts("From server1") end)
#PID<10101.106.0>
From server1
```

[†††] https://hexdocs.pm/elixir/Process.html

Erlang distribution assumes that nodes run in the same network, the traffic is not encrypted by default. It is possible to configure Erlang Distribution via TLS to secure the communication between the nodes on potentially insecure networks. [‡‡‡]

3 Elixir's Erlang/OTP concepts

Elixir provides modules to use common Erlang/OTP behaviors comfortably, this section explains some basic concepts about common modules.

3.1 GenServer

It is possible to create processes using the Elixir Process module that handle the receiving of messages and use a loop to constantly react on new message in the process mailbox. But this concept is already part of the language and Elixir offers you a generic approach to handle process communication in a standardized way. [§§§]

The generic server behavior is implemented by the GenServer module and allows creating a client-server relationship between processes. It allows having callbacks functions for messages and it keeps a state. This enables to quickly develop processes that handle messages. [****]

This code snippet shows a basic GenServer implementation with a handle_call function which will be called when the GenServer receives a call. The state of the GenServer starts with an empty list when the GenServer process gets a {:add, item} message it adds the item to the list.

GenServers support call functions which wait for a result to return that result to the caller and cast functions which are asynchronous, thus return directly.

```
defmodule DataStore do

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

  def handle_cast({:add, item}, state) do
   new_state = [item | state]
   {:noreply, new_state}
  end

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

end
```

[‡‡‡] http://erlang.org/doc/apps/erts/erl_dist_protocol.html
[§§§] https://hexdocs.pm/elixir/GenServer.html
[****] https://blog.appsignal.com/2018/06/12/elixir-alchemy-deconstructing-genservers.html

Using a GenServer means that all processes of the node can communicate synchronously and send messages to add items to the list, not without worrying about locks, mutexes or other concurrency problems. The GenServer will process the messages from its mailbox and update its state.

```
iex(1)> {:ok, pid} = GenServer.start(DataStore, [1,2,3])
iex(2)> GenServer.cast(pid ,{:add, 4})
iex(3)> GenServer.call(pid, :list)
[1,2,3,4]
```

3.2 Supervisor

Elixir applications benefit from the fault-tolerance of the Erlang VM, the philosophy behind the fault-tolerance is: "Let it crash!". Crashing processes should not affect the application and should be restarted automatically with a new state. This can be done via process linking: starting a GenServer with GenServer.start means, if the GenServer goes down, the process that started it will not be affected.

To take down the process that started the child process, you can use GenServer.start_link, which essentially means that the processes are linked, a terminated child process will take down the parent process. Essentially this means you can create trees of processes which watch each other and react on dying processes. Another reaction to a crashing process is a restart, in case of failures the process will be restarted with a new state and the system will carry on. [††††]

The concept of processes being restarted on failure is also called supervision, the structures that describe, which processes should be restarted under certain circumstances, are also known as supervision trees.

Elixir offers a module to create supervisors for processes with just a couple of lines, the supervisor is started on application startup and watches its child process, if a process dies, it will be automatically restarted by the Supervisor. [‡‡‡‡]

```
iex(1)> children = [{DataStore, []}]
iex(2)> {:ok, pid} = Supervisor.start_link(children, strategy: :one_for_one)
```

Supervisors offer different strategies for restarting processes, :one_for_one replaces the terminated process only, :one_for_all restarts all processes in the supervisors. Supervisors can be nested to create supervision trees to control how processes restart in case of failures. [§§§§]

[††††] (Thomas, 2016), p.230-231
[‡‡‡‡] https://hexdocs.pm/elixir/Supervisor.html
[§§§§] (Thomas, 2016), p.232

3.3 Task and Agents

A common mistake while creating Elixir application is using GenServer for everything. Sometimes the application needs to spawn a process to do a certain job, next time the application wants to make a state available for multiple processes. Using a GenServer provides both of these behaviors, but if only one of them is necessary Tasks and Agents offer either of those, internally they are implemented as GenServer with a certain focus. *****

Tasks can be used to make synchronous code asynchronous, they are an abstraction for processes and give developers more control about the results of code executed in a different process. †††††

Agents are processes which keep a state for sharing with other processes, they allow creating a process with an initial state, which can be retrieved and changed by all other components of the system. ‡‡‡‡‡

4 Built-in tools and libraries

Using Elixir allows developers to use all Erlang/OTP tools and libraries. This includes tools like e.g. Observer. Observer is a desktop application, which connects to a running Erlang node and allows to get system statistics, load charts, memory information, running processes, applications and process tracing. Observer is using Erlang Distribution for the communication with other nodes, thus it can also connect to production systems. §§§§§

The ErlangVM contains a built-in term storage system. ETS is able to store any Elixir/Erlang data structure in a key-value store that runs in memory and is organized in tables. An ETS table is kept as long as the node is running, this enables applications and processes to store data, which survives an application or process-crash. Using Elixir means ETS is available and can be used to store data that is available to all processes if desired. There is also an ETS implementation which is called DETS, which means the tables are stored on disk and can be persisted when a node is restarted. ******

Storing data in tables is a very common programming tasks, the built-in Erlang database is Mnesia. It is an implementation of DETS, which supports replication of your data across the whole cluster. Open-source products like, e.g. RabbitMQ use Mnesia to store messages and queues and replicate them across all RabbitMQ instances of the cluster. ††††††

For testing Elixir applications, the ExUnit Framework is part of every Elixir version, it allows assertions and supports test-driven-development. ‡‡‡‡‡‡

Applications that rely on creation of string-based markup, like web-applications, can use EEx which supports embedding and evaluating Elixir expressions in strings. EEx templates are compiled, this

***** (Thomas, 2016), p.255-256
††††† https://hexdocs.pm/elixir/Task.html
‡‡‡‡‡ https://hexdocs.pm/elixir/Agent.html
§§§§§ (Thomas, 2016), p. 179-182
****** http://erlang.org/doc/man/ets.html
†††††† https://learnyousomeerlang.com/mnesia
‡‡‡‡‡‡ https://hexdocs.pm/ex_unit/ExUnit.html

makes them fast. Popular web frameworks for Elixir use EEx templates for rendering web pages. §§§§§§

Managing dependencies and running different tasks is done by Mix, this build tool is used for creating, compiling and testing Elixir applications. It is extendable, so developers can create their own Mix tasks to automate common tasks or create templates. *******

The dependencies for Elixir are published via hex.pm. This package manager became so popular that it replaced the rebar build tool and other attempts in the Erlang community. Nowadays hex is the package manager for the Erlang ecosystem. †††††††

5 Conclusion

Compared with other languages Elixir is a relatively young language, it was started by José Valim in 2011 and is influenced by Erlang, Ruby and Clojure. It benefits from the maturity of the Erlang/VM and uses concepts that are well known to the Erlang programming language.

Erlang seems old, Elixir makes the concepts of Erlang easier to understand and adds a complete toolchain to create applications easily. Using Elixir is not a huge risk, as due to the fact that it relies on Erlang a solid platform is guaranteed. The Elixir community is very active, the language gained popularity through projects like the Phoenix webframework and Ecto for database access. Elixir treats errors in documentation as Bugs and developer happiness is one of the goals of the community.

The Phoenix Framework is a good starting point for creating Elixir applications, it can be compared to Ruby on Rails and has a built in WebSocket abstraction to create soft-realtime webapplications like Chats or Games. ‡‡‡‡‡‡‡

Thanks to Erlang, distribution, fault-tolerance and concurrency are a cornerstone of the language, the Actor concept simplifies creation of concurrent applications. It seems that distribution, fault-tolerance and concurrency are common problems in modern microservices architectures. Erlang had this covered in 1989. Using Elixir takes advantage of the knowledge Erlang has about running this kind of distributed workloads.

§§§§§§ https://hexdocs.pm/eex/EEx.html
******* https://hexdocs.pm/mix/Mix.html
††††††† https://hex.pm/
‡‡‡‡‡‡‡ https://phoenixframework.org/

References

Armstrong, J. (2013). *Programming Erlang - Software for a concurrent World.* Dallas, Texas, USA: The pragmatic programmers.

Herbert, F. (2013). *Learn You Some Erlang for Great Good!*

Loder, W. (2016). *Erlang and Elixir for Imperative Programmers.* apress.

Platformatec. (2019, 02 26). *elixir-lang.org*. Retrieved from https://elixir-lang.org/

Tan Wei Hao, B. (2017). *The little Elixir & OTP Guidebook.* Manning.

Thomas, D. (2016). *Programming Elixir.* The Pragmatic Programmers.