Elixir IN ACTION

Saša Jurić

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Elixir in Action
Version 01

For more information on this and other Manning titles go to
www.manning.com

brief contents

FM1 About this book

This book has a couple of goals, listed here in the order of importance:

1. Building highly available systems
2. Taking advantage of the Erlang virtual machine to do it
3. Using the Elixir language as an interface to the EVM

As you can see, the book has a very practical focus - it aims to teach you how to write highly available systems, using arguably the best currently available tools for the job: Erlang and Elixir. Therefore, learning about Erlang and Elixir is more like a secondary goal. Of course, to leverage both technologies properly, you need to know a great deal about them. But whatever is being taught here is directly or indirectly related to the primary goal: how to build highly available systems.

After finishing the book, you will have enough knowledge to start developing production ready systems on your own. Of course, there will be many additional topics, tools, and techniques, you'll have to research for yourself. There simply isn't enough space in the book to treat them all. However, you'll gain the knowledge that is broad enough to get you started.

Scope

With this in mind, this book is more like a "peer-to-peer" tutorial - a knowledge transfer from one professional to another. It is definitely not a complete reference on Elixir and Erlang. It does not cover every single nuance of the language, or every possible aspect of EVM. It glosses over or even completely skips many topics such as integer size, floating point precision, Unicode specifics, file I/O, unit

testing, and many more. While relevant beyond doubt, such topics are not our primary goal, and I have full confidence that you will be able to research them yourself, when the need arises.

Omitting or quickly dealing with these somewhat tedious and more conventional topics, gives us space to treat more exciting and unconventional areas with more details. Concurrent programming, the way it helps bringing scalability, fault tolerance, distribution and availability to our system - these are the core themes of the book, the topics that receive an extended treatment.

Prerequisites

Since the book deals with a lot of "upper intermediate" topics, there are some prerequisites you should meet. It is assumed that you are a professional software developer, with a couple of years of experience in developing some kind of software. The exact technology you are proficient in is not relevant: it can be Java, C#, Ruby, C++ or some other general purpose programming language. You don't need to know anything about Erlang, Elixir, or any other concurrent platform. In particular, you don't need to know anything about functional programming. It's also not necessary to be familiar about web or server side programming, although any previous knowledge in the area will come in hand. You should, however, understand the client-server principles, and the way they underpin typical server platforms, such as web servers. Some general, high-level knowledge about networking (request-response, TCP, HTTP) should be present.

A small point about functional programming. Elixir is a functional language, and if you come from an OO background this might scare you a bit. Being myself a long time OO programmer, I can sincerely tell you not to worry. Unlike for example Haskell, Elixir is a very simple language which doesn't rely on more abstract concepts such as monads, monoids, functors and the like. The underlying functional concepts in Elixir are relatively simple and should be easy to grasp. Of course, functional programming is significantly different from whatever you have seen in a typical OO language, and it takes some time getting used to. But it's definitely not a rocket science, and if you are an experienced developer, you should have no problem understanding these concepts.

Organization

The book is divided into three parts.

The first part introduces the Elixir language, presenting and describing its basic building blocks, and then covering common functional programming idioms with

more details. After finishing the first part, you will gain the basic knowledge of the Elixir language and functional programming techniques.

The second part builds on these foundations and presents the most important building blocks of Erlang systems. You will learn about Erlang concurrency model and it's many benefits, such as scalability and fault-tolerance. The most important aspects of the OTP framework will be presented, which will allow you to use community accepted patterns and practices and create scalable and reliable systems.

The final part of the book deals with the system in production. You will learn how to build distributed systems, create deployable releases, interact with running systems, and analyze their behavior.

1 First steps

This chapter covers

- Overview of Erlang
- Benefits of Elixir

This is the beginning of your journey to the world of Elixir and Erlang, two efficient and useful technologies that can significantly simplify development of large, scalable systems. Chances are, you are reading this book to learn about Elixir. However, since Elixir is built on top of Erlang, and depends heavily on it, you should first learn a bit about what exactly Erlang is, and which benefits does it offer. So let's take a brief, high-level look at Erlang.

1.1 About Erlang

Erlang is a development platform for building scalable and reliable systems that constantly provide service with very little or no downtime.

This is a very bold statement, but it's exactly what Erlang was made for. Conceived in mid 80s in Ericsson, a Swedish telecom giant, Erlang was driven by the needs of the company's own telecom systems where properties like reliability, responsiveness, scalability, and constant availability are imperative. A telephone network should always operate regardless of the number of simultaneous calls, unexpected bugs, or hardware and software upgrades taking place.

Despite being originally built for telecom systems, Erlang is in no way specialized for this domain. It doesn't contain explicit support for programming telephones, switches or other telecom devices. Instead, it is a general purpose development platform that provides special support for technical, non-functional

challenges such as concurrency, scalability, fault-tolerance, distribution, and high availability.

In late 80s and early 90s, with most software being desktop based, the need for high availability may have been limited only to specialized systems, such as telecoms. Today, we face a much different situation, with the focus shifting on the Internet and Web, and most of the applications being driven and supported by some kind of a server system which processes requests, crunches data and pushes relevant informations to many connected clients. Today's popular systems are more about communication and collaboration, the examples including social networks, content management systems, on demand multimedia, or multi-player games.

All of these systems have some common non-functional requirements. The system must be responsive, regardless of the number of connected clients. The impact of unexpected errors must be as minimal as possible, instead of affecting the entire system. It's acceptable if an occasional request fails due to some bug, but it's a major problem when the entire system becomes completely unavailable. Ideally, the system should never crash or be taken down, not even when a software upgrade takes place. It should always be up and running, providing the service to its clients.

The goals above might seem hard, but they're imperative when building systems that people depend on. Unless a system is responsive and reliable, it will eventually fail to fulfill its purpose. Therefore, when building server side systems, it is essential to make the system constantly available or to get there as close as possible.

And this is exactly the intended purpose of Erlang. High availability is explicitly supported via technical concepts such as scalability, fault tolerance, or distribution. Unlike most of other modern development platforms, these concepts were the main motivation and the driving force behind the development of Erlang. Ericsson team, led by Joe Armstrong, spent a couple of years of designing, prototyping and experimenting before creating the development platform today known as Erlang. Its uses may have been limited in early 90s, but today almost any system can benefit from it.

So it comes as no surprise that Erlang has recently gained more attention, and today it powers a broad range of large systems such as Facebook chat backend,

WhatsApp messaging application, Riak distributed database, Heroku cloud, Chef deployment automation system, RabbitMQ message queue, financial systems or multiplayer backends.

As you can see, Erlang powers various large systems, and has been doing so for more than two decades. It is truly a proven technology, both in time and scale. But what is the magic behind Erlang? Let's take a look at how Erlang helps us build highly available, reliable systems.

1.1.1 High availability

Erlang was specifically created to support the development of highly available systems - systems which are always online and provide service to their clients, even when faced with unexpected circumstances. On the surface, this may seem simple, but as you probably know, many things can go wrong in production. To make our systems work 24/7 without any downtime, we have to tackle some technical challenges:

- *Fault tolerance* - A system has to keep working when something unforeseen happens. Unexpected errors will happen, bugs will creep in, components might occasionally fail, network connections might drop, even the entire machine where the system is running might crash. Whatever happens, we want to localize the impact of an error as much as possible, recover from the error and keep the system running and providing service.
- *Scalability* - A system should be able to handle any possible load. Of course that we will not buy tons of hardware just in case the entire planet population starts using our system some day. But we must be able to respond to a load increase by adding more hardware resources, without any software intervention. Ideally, this should be possible without a system restart.
- *Distribution* - To make a system that never stops, we have to run it on multiple machines. This promotes the overall stability of the system - if some machine is taken down, another one may take over. Furthermore, this gives us means to scale horizontally - we can address load increase by adding more machines to the system, thus adding more work units to support the higher demand.
- *Responsiveness* - It goes without saying that a system should always be reasonably fast and responsive. Drastic prolongations of request handling shouldn't happen, even if the load increases or unexpected errors happens. In particular, occasional long tasks shouldn't block the rest of the system, or have a significant effect on its performance.
- *Live update* - In some cases we want to push a new version of our software without restarting any server. For example, in a telephony system, we don't want to disconnect the established calls while upgrading the software.

If we manage to handle these challenges, our system will truly become highly available and be able to constantly provide the service to its users, come rain or shine.

Erlang gives us tools to address these challenges - that is exactly what it was

built for. We can gain all of these properties, and ultimately become highly available, through the power of Erlang concurrency model. So let's take a higher level look at how concurrency in Erlang works.

1.1.2 Erlang concurrency

Concurrency is at the very heart and soul of Erlang systems. Almost every non trivial Erlang based production system is highly concurrent. Even the programming language is sometimes labeled as the concurrency oriented language.

Instead of relying on heavy-weight threads and OS processes, Erlang takes the concurrency into its own hands, as illustrated in figure 1.1:

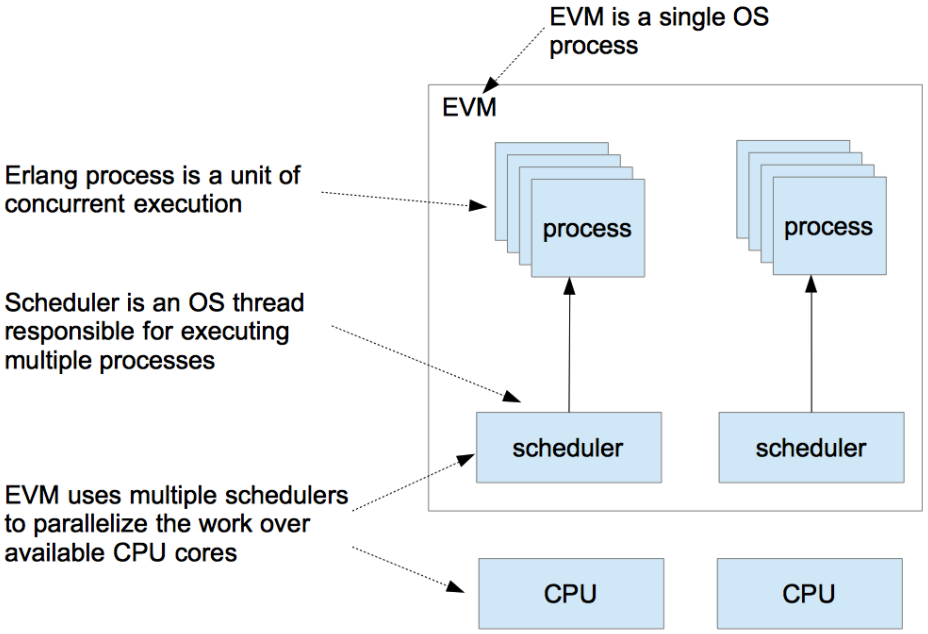

Figure 1.1 Concurrency in Erlang Virtual Machine

The basic concurrency primitive is called an *Erlang process* (not to be confused with OS processes or threads), and typical Erlang systems run thousands or even millions of such processes. The runtime, called Erlang Virtual Machine (EVM), uses its own schedulers to distribute the execution of the processes over the available CPU cores, thus parallelizing the execution as much as possible. Due to the way processes are implemented, we can obtain many benefits.

FAULT-TOLERANCE

Erlang processes are completely isolated from each other. They share no memory and a crash of one process won't cause a crash of other processes. This helps us isolate the effect of an unexpected error. If something bad happens, it only has a local impact. Moreover, Erlang provides us with means to detect a process crash and do something about it, typically start the new process in the place of the crashed one.

SCALABILITY

Sharing no memory, processes communicate via asynchronous messages. This means there are no complex synchronization mechanisms, such as locks, mutexes, or semaphores. Consequently, the interaction between concurrent entities is much simpler to develop and understand. Typical Erlang systems are divided into a large number concurrent processes, which cooperate together to provide the complete service. EVM can efficiently parallelize the execution of processes as much as possible. This makes Erlang systems scalable, since they can take advantage of all available CPU cores.

DISTRIBUTION

Communication mechanism between processes works exactly the same, regardless of whether these processes reside in the same instance of the EVM, or on two different instances on two separate, remote computers. Therefore, a typical, highly concurrent, Erlang based system is automatically ready to be distributed over multiple machines. This in turn gives us ability to scale out - to run a cluster of machines which share the total system load. Additionally, running on multiple machines makes the system truly resilient - if one machine crashes, other ones can take over.

RESPONSIVENESS

The runtime is specifically tuned to promote overall responsiveness of the system. It has been mentioned that Erlang takes the execution of multiple processes into its own hands by employing dedicated schedulers that interchangeably execute many Erlang processes. A scheduler is preemptive - it gives a small execution window to each process, then pauses it and runs another process. Since execution window is very small, a single long-running process can't block the rest of the system. Furthermore, I/O operations are internally delegated to separate threads, or kernel poll service of the underlying OS is used if available. This means that any process that waits for an I/O operation to finish will not block the execution of other processes.

Even the garbage collection is specifically tuned to promote the system responsiveness. Recall that processes are completely isolated, and share no memory. This allows per-process garbage collection - instead of stopping the entire system, each process is individually collected when needed. Such collections will be much shorter and won't block the entire system for longer periods of time. In fact, in a multi-core system, it is quite possible for one CPU core to run a short garbage collection, while the remaining cores are doing some standard processing.

So as you can see, concurrency is a crucial element in Erlang, not related only to parallelism. Owing to the underlying implementation, concurrency promotes fault-tolerance, distribution and system responsiveness. Typical Erlang systems run many concurrent tasks, using thousands or even millions of processes. This can be especially useful when developing server side systems, which can often be implemented completely in Erlang.

1.1.3 Server side systems

Erlang can be used in various applications and systems. There are examples of Erlang based desktop applications, and it is often used in embedded environments. Its sweet spot, in my opinion, lies in server side systems - systems which run on one or more server and must serve many simultaneous clients. A term "server side system" indicates that it is more than a simple server that processes requests. It's an entire system that in addition to request handling, must run various background jobs and manage some kind of server-wide in-memory state, as illustrated in figure 1.2:

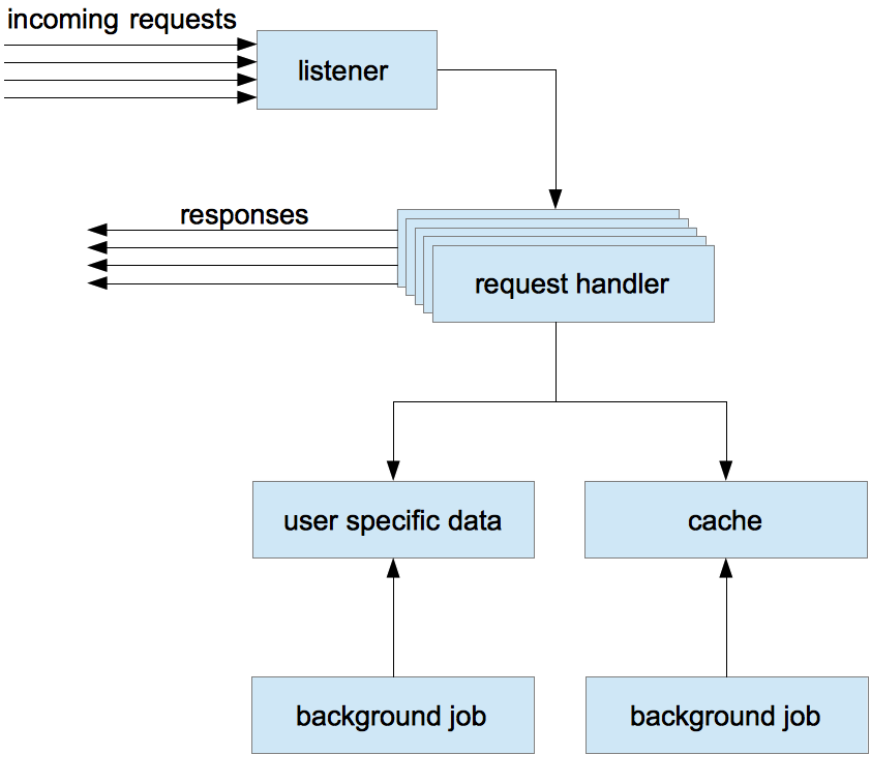

Figure 1.2 Server side system

Moreover, a server side system may often be distributed on multiple machines, which collaborate together in order to produce some business value. We might place different components to different machines, and we also might deploy some components on multiple servers to achieve load balancing and/or support failover scenarios.

This is where Erlang can make our life significantly simpler. By giving us primitives to make our code concurrent, scalable and distributed, it allows us to implement the entire system completely in Erlang. Every single component in figure 1.2 can be implemented as an Erlang process! This will make the system scalable and fault-tolerant, and easy to distribute. By relying on Erlang's error detection and recovery primitives, we can further increase the reliability and recover from unexpected errors.

Let's take a look at a real-life example. I have professionally been involved in the development of two web servers, both having similar technical needs: they serve multitude of clients, handle long running requests, manage some server-wide in-memory state, persist the data which must survive OS process or machine restarts, and run some background jobs.

Take a look at the table 1.1 that lists technologies used in each server:

Table 1.1 Comparison of technologies used in two real-life web servers

Technical requirement	Server A	Server B
HTTP server	Nginx + Phusion Passenger	Erlang
Request processing	Ruby on Rails	Erlang
Long running requests	Go	Erlang
Server wide state	Redis	Erlang
Persistable data	Redis, MongoDB	Erlang
Background jobs	Cron, Bash scripts, Ruby	Erlang
Service crash recovery	Upstart	Erlang

The server "A" is powered by various technologies, most of them being known and popular in the community. There were specific reasons for using these technologies, each being introduced to resolve a shortcoming of the ones already present in the system. For example, Ruby on Rails handles concurrent requests in separate OS processes. We needed a way to share some data between these different processes, so we introduced Redis. Similarly, MongoDB is used to manage a persistent front-end data, most often some user related informations. So there is a rationale behind every technology used in server "A", but the entire solution still seems somewhat complex. It’s not contained in a single project, the components are deployed separately, and it is not trivial to start the entire system on a development machine. We actually had to develop a tool to help us start the entire system locally!

In contrast, the server "B" accomplishes the same technical requirements relying on a single technology, using the platform’s features created specifically for these purposes and proven in large systems. Moreover the entire server is a

single project which runs inside a single EVM instance - in production it runs inside only one OS process, using only a handful of OS threads. The concurrency is handled completely by the Erlang scheduler, and the system is scalable, responsive and fault tolerant. Being implemented as a single project, the system is easier to manage, deploy, and run locally on the development machine.

It's important to notice that Erlang tools are not always full-blown alternatives to mainstream solutions, such as web servers like Nginx, database servers like Riak, or in-memory key-value stores like Redis. However, Erlang gives us some options, making it possible to implement initial solution using exclusively Erlang, and resort to alternative technologies when Erlang solution is not good enough. This makes the entire system more homogeneous, and therefore easier to develop and maintain.

It's also worth noting that Erlang is not an isolated island. It can run in-process C code, and can communicate with practically any external component such as message queue, in-memory key-value store or external database. Therefore, when opting for Erlang you won't be deprived of relying on existing 3rd party technologies. Instead, you will have the option of using them when they are really called for, and not because your primary development platform doesn't give you any tool for your problems.

Now that you know about Erlang's strengths, and the areas where it excels, let's take a closer look at what Erlang exactly is.

1.1.4 The development platform

Erlang is more than a programming language. It is a full blown development platform, consisting of four distinct parts: the language, the virtual machine, the framework, and the tools.

Erlang, the language, is the primary way of writing code that will run in the Erlang virtual machine. It is a simple functional language with basic concurrency primitives.

Source code written in Erlang is compiled into byte code which is then executed inside the EVM. This is where the true magic happens. EVM parallelizes our concurrent Erlang programs and takes care of the process isolation, distribution, and overall responsiveness of the system.

The standard part of the release is the framework called OTP (Open Telecom Platform). Despite its somewhat unfortunate name, the framework has nothing to do with telecom systems. It is a general purpose framework that abstracts away many typical Erlang tasks:

- Concurrency and distribution patterns
- Error detection and recovery in concurrent systems
- Packaging the code into libraries
- Systems deployment
- Live code update

All of the above can be done without OTP, but it really makes no sense. OTP is battle tested in many production systems and is such an integral part of Erlang that it is hard to draw the exact line between the two. Even the official distribution is called Erlang/OTP.

The tools are used for various typical tasks, such as compiling Erlang code, starting an EVM instance, creating deployable releases, running the interactive shell, connecting to the running EVM instance, and so on.

Both EVM and accompanying tools are cross platform. You can run them on most mainstream operating systems such as Unix, Linux and Windows. The entire Erlang distribution is open sourced and you can find the source on the official site or on the Erlang GitHub repository. Ericsson is still in charge of the development process, and releases a new version on a regular basis, once a year.

So that concludes the story about Erlang. But if Erlang is so great, why do we need Elixir? The next section aims to answer this question.

1.2 About Elixir

Elixir is the alternative language for the Erlang virtual machine that allows us to write cleaner, more compact, better intention-revealing code. You write your programs in Elixir and run them normally on EVM.

Elixir is an open source project, originally started by José Valim. Unlike Erlang, Elixir is more of a collaborative effort, presently counting more than 100 contributors. New features are often being discussed on mailing lists, GitHub issue tracker or #elixir-lang freenode IRC channel. José still has the last word, but the entire project seems more like a true open source collaboration, attracting an interesting mixture of seasoned Erlang veterans and talented young developers. The source code can be found on the GitHub repository.

Elixir targets the Erlang runtime. The result of compiling the Elixir source code are EVM compliant byte-code files which can run inside an EVM instance and can normally cooperate with the pure Erlang code - we can use Erlang libraries from

Elixir and vice versa. There is nothing we can do in Erlang that cannot be done in Elixir, and most often the Elixir code will be as performant as its Erlang counterpart.

Elixir is semantically very close to Erlang, with many of its language constructs mapping directly to the Erlang counterparts. However, Elixir provides some additional constructs that make it possible to radically reduce boilerplate and duplication. In addition, it tidies up some important parts of the standard libraries, provides some nice syntactic sugars, and a uniform tool for creating and packaging systems. Everything you can do in Erlang is possible in Elixir, and vice versa, but in my experience, the Elixir solution will usually be easier to develop and maintain.

Let's take a closer look at how Elixir improves on some of Erlang features. We'll start with boilerplate and noise reduction.

1.2.1 Code simplification

On of the most important benefits of Elixir is the ability to radically reduce boilerplate and eliminate noise from the code, which results in a simpler code that is easier to write and maintain.

Let's see what this means by contrasting Erlang and Elixir code. One of the basic building blocks of Erlang concurrent systems are *actors*. Actors are something like concurrent objects - they embed private state and can interact with other actors via messages. Unlike objects, actors are concurrent, which means that two actors may run in parallel, assuming at least two CPU cores are available. Typical Erlang systems heavily rely on actors, running thousands or even millions of them. So actors are by all means a very important construct in the Erlang world.

Let's take a look at an Erlang code that implements a simple actor that adds two numbers. The actor is implemented in listing 1.1:

Listing 1.1 Erlang based actor that adds two numbers

```
-module(sum_actor).

-behaviour(gen_server).
-export([
  start/0, sum/3,
  init/1, handle_call/3, handle_cast/2, handle_info/2, terminate/2,
  code_change/3
]).

start() -> gen_server:start(?MODULE, [], []).
sum(Server, A, B) -> gen_server:call(Server, {sum, A, B}).

init(_) -> {ok, undefined}.

handle_call({sum, A, B}, _From, State) -> {reply, A + B, State};
handle_cast(_Msg, State) -> {noreply, State}.
handle_info(_Info, State) -> {noreply, State}.
terminate(_Reason, _State) -> ok.
code_change(_OldVsn, State, _Extra) -> {ok, State}.
```

Even without any knowledge about Erlang, this seems like a lot of code for something that only adds two numbers. To be fair, the addition is concurrent (it is an actor), but regardless, due to the amount of code it's hard to see the forest for the trees. It's definitely not immediately obvious what the code exactly does. Moreover it is difficult to write such code. Personally, even after years of production level Erlang development, I still can't write this without consulting the Internet or some previously written code.

The real problem of Erlang is that this boilerplate is almost impossible to remove, even if it is completely the same in most places (which in my experience is the case). The language gives us almost no support to somehow eliminate this noise. In all fairness, there is a way to reduce the boilerplate, using the construct called parse transform, but it is very clumsy and complicated to use. So in practice, Erlang developers simply write their actors using the just presented pattern, and learn to live with the accompanying noise.

Given that actors are basic building block of scalable and reliable Erlang based systems, it is an unfortunate state that Erlang developers have to constantly copy-paste this noise and work with it. Surprisingly, many people get used to it, probably due to wonderful things EVM does for them. It is often said that Erlang makes hard things easy and easy things hard. Still, the code above leaves an impression that we should do better.

Let’s see the Elixir version of the same actor, presented in the listing 1.2:

Listing 1.2 Elixir based actor that adds two numbers

```
defmodule SumActor do
  use GenServer.Behaviour

  def start, do
    :gen_server.start(__MODULE__, [], [])
  end

  def sum(server, a,b) do
    :gen_server.call(server, {:sum, a, b})
  end

  def handle_call({:sum, a, b}, _from, state) do
    {:reply, a + b, state}
  end
end
```

This code is significantly smaller, and therefore easier to read and maintain. It is more intention revealing and less burdened with noise. And still, it is exactly as capable and flexible as the Erlang version. It behaves exactly the same in runtime, retains complete semantics, and there is nothing you can do in Erlang version that’s not possible in Elixir counterpart.

Despite being significantly smaller, the Elixir version of a sum actor still feels somewhat noisy, given that all it does is an addition of two numbers. The excess noise exists because Elixir retains 1:1 semantic relation to the underlying Erlang library that is used to create actors.

However, the language gives us tools to further eliminate whatever we may regard as noise and duplication. For example, I have developed my own Elixir library called ExActor (Elixir Actor), which makes the actor definition really dense. The actor implementation is given in a listing 1.3:

Listing 1.3 Elixir based actor that adds two numbers, based on the 3rd party abstraction

```
defmodule SumActor do
  use ExActor

  defcall sum(a, b) do
    a + b
  end
end
```

The intention of this code should be obvious even to developers with no previous Elixir experience. In runtime, this code will work almost completely the same as the two previous versions. Transformation that makes this code behave like the previous one will happen in compile time. When it comes to the byte code, all three versions will be very similar, and there will be no significant differences in run-time behavior.

NOTE **Disclaimer**

ExActor library is mentioned here only to illustrate how much can we abstract away in Elixir. However, in this book we will not use this library, because it's a 3rd party abstraction that hides away important details of how actors really work. To completely take advantage of actors, it's important that you understand what makes them tick, which is why we will rely on lower level abstractions. Once you understand how actors work, you can decide for yourself whether you want to use ExActor, or some of the other available higher level abstractions.

This last implementation of the sum actor is powered by the Elixir macros facility (not to be confused with C style macros). A macro is an Elixir code which runs in *compile time*. Macro takes internal representation of your source code as input, and can create alternative output. Unlike C/C++ macros which work with pure text, Elixir macros work on abstract syntax tree (AST) structure, which makes it easier to perform non trivial manipulations of the input to some kind of alternative output. Of course, Elixir gives us some helper constructs to simplify this transformation.

Take another look at how the actor `sum` operation is defined in the last example:

```
defcall sum(a, b) do
  a + b
end
```

Notice the *defcall* at the beginning. There is no such keyword in Elixir. This is a custom macro that translates the given definition to something like the following:

```
def sum(server, a,b) do
  :gen_server.call(server, {:sum, a, b})
end
```

```
def handle_call({:sum, a, b}, _from, state) do
  {:reply, a + b, state}
end
```

Since macros are written in Elixir, they are very flexible and powerful, making it possible to extend the language and introduce new constructs that look like an integral part of the language. For example, an open source Ecto project, which aims to bring LINQ style queries to Elixir, is also powered by Elixir macro support, and provides an expressive query syntax which deceptively looks like the part of the language:

```
from w in Weather,
  where: w.prcp > 0 or w.prcp == nil,
  select: w
```

Due to macros support and smart compiler architecture, most of Elixir is actually written in Elixir. Language constructs like `if`, `unless`, or records support are implemented via Elixir macros. Only the smallest possible core is done in Erlang, and everything else is then built on top of it in Elixir!

Macros are somewhat of a black art of Elixir, but they make it possible to flush out non-trivial boilerplate in compile time, and extend the language with our own DSL like constructs.

But Elixir is not all about macros. Another worthy improvement is a seemingly simple syntactic sugar that makes functional programming much easier.

1.2.2 Composing functions

Both Erlang and Elixir are functional languages. They rely on immutable data and functions which transform this data. One of the supposed benefits of this approach is that code is divided into many small reusable and composeable functions.

Unfortunately, the composability feature works clumsy in Erlang. Let’s look at an adapted example from my own production. One piece of code, I am responsible for, maintains some in-memory model, and receives xml messages that modify that model. When an xml message arrives, the following actions must be done:

1. Apply xml to the in-memory model
2. Process the resulting changes
3. Persist the model

Here's an Erlang sketch of the corresponding function:

```
process_xml(Model, Xml) ->
  Model1 = update(Model, Xml),
  Model2 = process_changes(Model1),
  persist(Model2).
```

I don't know about you, but this doesn't look very composable to me. Instead, it seems fairly noisy, and error prone. The temporary variables `Model1` and `Model2` are introduced here only to take the result of one function and feed it to the next.

Of course, we could eliminate temporary variables and inline the calls:

```
process_xml(Model, Xml) ->
  persist(
    process_changes(
      update(Model, Xml)
    )
  ).
```

This style, known as staircasing, is admittedly free of temporary variables, but is very clumsy, and hard to read. To understand what goes on here we have to manually parse it "inside-out".

While Erlang programmers are more or less limited to such clumsy approaches, Elixir gives us a very elegant way of chaining multiple function calls together:

```
def process_xml(model, xml) do
  model
  |> update(xml)
  |> process_changes
  |> persist
end
```

The *pipeline* operator `|>` takes the result of the previous expression and feeds it as the first argument to the next one. The resulting code is clean, contains no temporary variables and reads like the prose, top to bottom, left to right. Under the hood this is transformed in compile time to the staircased version. This is again possible because of the Elixir's macro system.

The pipeline operator truly highlights the power of functional programming. We treat the functions as data transformations, and then combine them in different

ways to gain the desired effect.

1.2.3 The big picture

There are many other areas where Elixir improves the original Erlang approaches. The API for standard libraries is standardized, some conventions and norms are introduced which don't exist in Erlang. Syntactic sugars are introduced that simplify typical idioms, and even some Erlang data structures, such as key-value dictionary and set, are rewritten to gain more performance. Concise syntax for working with structured data is provided. Strings manipulation is improved, and the language has some explicit support for unicode manipulation. In addition, Elixir distribution includes a tool that provides the standard ways of creating applications and libraries, managing dependencies, compiling and testing the code, and packaging releases.

The list goes on and on, but instead of presenting each feature, I'd like to express a personal sentiment based on the professional experience. Personally, I find it much more pleasant to code in Elixir. The resulting code is much cleaner, more readable, and less burdened with the boilerplate, noise, and duplication. At the same time we retain the complete run-time characteristics of the pure Erlang code. Finally, we can still use all of the available libraries from the Erlang ecosystem, both standard and 3rd party.

1.3 Summary

This chapter was somewhat theoretical, but it was important to define the purpose and the benefits of the technologies, this book deals with. There are a couple of points worth remembering:

- Erlang is a technology for developing highly available systems that constantly provide service with very little or no downtime. It has been battle tested in diverse large systems for more than two decades.
- Elixir is a modern language that makes the development for Erlang platform much more pleasant, helping us to organize the code more efficiently and abstract away boilerplate, noise and duplication.

Now you can really start the journey by looking at the basic building blocks of Elixir programs.