brief contents

Part 1

The language

The first part of the book is an introduction to the Elixir language. We start by providing a high-level overview of Elixir and Erlang, discussing the goals and benefits of both technologies. In chapter 2, you'll learn about the basic building blocks of the Elixir language, such as modules, functions, and the type system. Chapter 3 details the treatment of pattern-matching and control-flow idioms. In chapter 4, you'll learn how to implement higher-level data abstractions with immutable data structures.

1 First steps

This chapter covers

- Overview of Erlang
- Benefits of Elixir

This is the beginning of your journey into the world of Elixir and Erlang, two efficient and useful technologies that can significantly simplify the development of large, scalable systems. Chances are, you're reading this book to learn about Elixir. But because Elixir is built on top of Erlang and depends heavily on it, you should first learn a bit about what Erlang is and the benefits it offers. 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 little or no downtime. This is a bold statement, but it's exactly what Erlang was made for. Conceived in the mid-1980s by 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's a general-purpose development platform that provides special support for technical, nonfunctional challenges such as concurrency, scalability, fault-tolerance, distribution, and high availability.

In late 1980s and early 90s, when most software was desktop based, the need for high availability was limited to specialized systems such as telecoms. Today we face a much different situation: the focus is on the internet and the Web, and most applications are driven and supported by a server system that processes requests, crunches data, and pushes relevant informations to many connected clients. Today's popular systems are more about communication and collaboration; examples include social networks, content-management systems, on-demand multimedia, and multiplayer games.

All of these systems have some common nonfunctional 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 a 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 during a software upgrade. It should always be up and running, providing the service to its clients.

These goals might seem difficult, 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's essential to make the system constantly available.

And this is the intended purpose of Erlang. High availability is explicitly supported via technical concepts such as scalability, fault tolerance, and distribution. Unlike with most other modern development platforms, these concepts were the main motivation and driving force behind the development of Erlang. The Ericsson team, led by Joe Armstrong, spent a couple of years 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.

Erlang has recently gained more attention. It powers various large systems and has been doing so for more than two decades, such as the WhatsApp messaging application, the Riak distributed database, the Heroku cloud, the Chef deployment automation system, the RabbitMQ message queue, financial systems, and multiplayer backends. It's truly a proven technology, both in time and scale. But what is the magic behind Erlang? Let's take a look at how Erlang can help you build highly available, reliable systems.

1.1.1 High availability

Erlang was specifically created to support the development of highly available systems—systems that 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 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 occur, bugs creep in, components occasionally fail, network connections drop, or the entire machine where the system is running crashes. 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, we don't buy tons of hardware just in case the entire planet's population might start using our system some day. But we should 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 a machine is taken down, another one can take over. Furthermore, this gives us means to scale horizontally—we can address load increase by adding more machines to the system, thus adding work units to support the higher demand.
- *Responsiveness*—It goes without saying that a system should always be reasonably fast and responsive. Request handling shouldn't be drastically prolonged, even if the load increases or unexpected errors happen. In particular, occasional lengthy tasks shouldn't block the rest of the system or have a significant effect on performance.
- *Live update*—In some cases, you may want to push a new version of your software without restarting any servers. For example, in a telephony system, we don't want to disconnect established calls while we upgrade the software.

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

Erlang gives us tools to address these challenges—that is what it was built for. A system can gain all these properties and ultimately become highly available through the power of the Erlang concurrency model. Next, let's look at how concurrency works in Erlang.

1.1.2 Erlang concurrency

Concurrency is at the heart and soul of Erlang systems. Almost every nontrivial Erlang-based production system is highly concurrent. Even the programming language is sometimes called a *concurrency-oriented language*. Instead of relying on heavyweight threads and OS processes, Erlang takes concurrency into its own hands, as illustrated in figure 1.1.

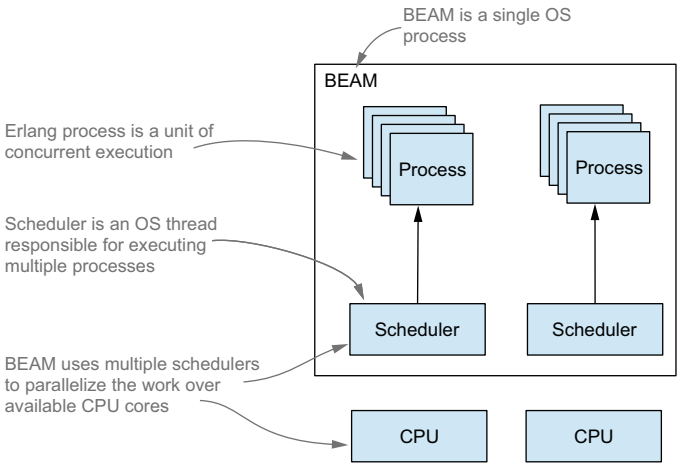

Figure 1.1 Concurrency in the Erlang virtual machine

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 Erlang virtual machine, called BEAM[1], uses its own schedulers to distribute the execution of processes over the available CPU cores, thus parallelizing execution as much as possible. The way processes are implemented provides many benefits.

FAULT TOLERANCE

Erlang processes are completely isolated from each other. They share no memory, and a crash of one process doesn't cause a crash of other processes. This helps you isolate the effect of an unexpected error. If something bad happens, it has only a local impact. Moreover, Erlang provides you with means to detect a process crash and do something about it: typically, you start a new process in 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 of concurrent processes, which cooperate together to provide the complete service. The virtual machine can efficiently parallelize the execution of processes as much as possible. This makes Erlang systems scalable, because they can take advantage of all available CPU cores.

DISTRIBUTION

Communication between processes works the same way regardless of whether these processes reside in the same BEAM instance or on two different instances on two separate, remote computers. Therefore, a typical highly concurrent Erlang-based system is

[1] Bogdan/Björn's Erlang Abstract Machine.

automatically ready to be distributed over multiple machines. This in turn gives you the ability to scale out—to run a cluster of machines that share the total system load. Additionally, running on multiple machines makes the system truly resilient—if one machine crashes, others can take over.

RESPONSIVENESS

Runtime is specifically tuned to promote overall responsiveness of the system. I've 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 and then pauses it and runs another process. Because the execution window is 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 a kernel-poll service of the underlying OS is used if available. This means any process that waits for an I/O operation to finish won't block the execution of other processes.

Even garbage collection is specifically tuned to promote 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 as needed. Such collections are much shorter and don't block the entire system for long periods of time. In fact, in a multicore system, it's possible for one CPU core to run a short garbage collection while the remaining cores are doing standard processing.

As you can see, concurrency is a crucial element in Erlang; and it's related to more than just 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 you're 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's often used in embedded environments. Its sweet spot, in my opinion, lies in server-side systems—systems that run on one or more server and must serve many simultaneous clients. The term *server-side system* indicates that it's 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.

A server-side system is often distributed on multiple machines that collaborate to produce business value. You might place different components on different machines, and you also might deploy some components on multiple servers to achieve load balancing and/or support failover scenarios.

This is where Erlang can make your life significantly simpler. By giving you primitives to make your code concurrent, scalable, and distributed, it allows you to implement the

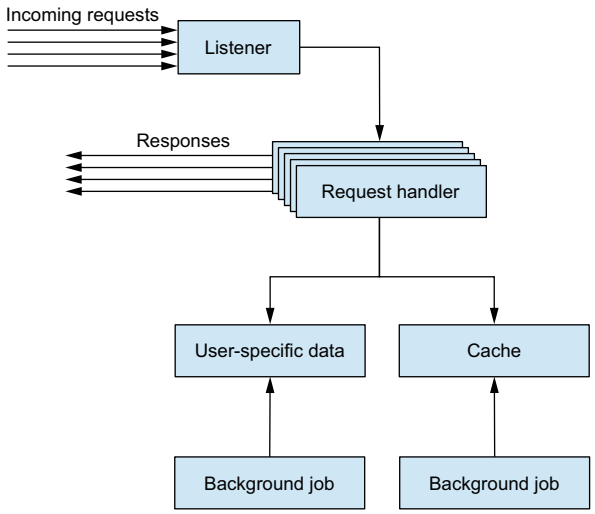

Figure 1.2 Server-side system

entire system completely in Erlang. Every component in figure 1.2 can be implemented as an Erlang process! This makes the system scalable, fault tolerant, and easy to distribute. By relying on Erlang's error-detection and recovery primitives, you can further increase reliability and recover from unexpected errors.

Let's look at a real-life example. I have been involved professionally in the development of two web servers, both of which have similar technical needs: they serve a multitude of clients, handle long-running requests, manage server-wide in-memory state, persist data that must survive OS process and machine restarts, and run background jobs. Table 1.1 lists the 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 and Phusion Passenger	Erlang
Request processing	Ruby on Rails	Erlang
Long-running requests	Go	Erlang
Server-wide state	Redis	Erlang
Persistable data	Redis and MongoDB	Erlang
Background jobs	Cron, Bash scripts, and Ruby	Erlang
Service crash recovery	Upstart	Erlang

Server A is powered by various technologies, most of them known and popular in the community. There were specific reasons for using these technologies: each was introduced to resolve a shortcoming of those already present in the system. For example,

Ruby on Rails handles concurrent requests in separate OS processes. We needed a way to share data between these different processes, so we introduced Redis. Similarly, MongoDB is used to manage persistent front-end data, most often user-related information. Thus there is a rationale behind every technology used in server A, but the entire solution seems complex. It's not contained in a single project, the components are deployed separately, and it isn't trivial to start the entire system on a development machine. We had to develop a tool to help us start the system locally!

In contrast, server B accomplishes the same technical requirements while relying on a single technology, using platform features created specifically for these purposes and proven in large systems. Moreover, the entire server is a single project that runs inside a single BEAM instance—in production, it runs inside only one OS process, using a handful of OS threads. Concurrency is handled completely by the Erlang scheduler, and the system is scalable, responsive, and fault tolerant. Because it's 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 aren't always full-blown alternatives to mainstream solutions, such as web servers like Nginx, database servers like Riak, and in-memory key-value stores like Redis. But Erlang gives you options, making it possible to implement an initial solution using exclusively Erlang and resort to alternative technologies when an Erlang solution isn't sufficient. This makes the entire system more homogeneous and therefore easier to develop and maintain.

It's also worth noting that Erlang isn't an isolated island. It can run in-process C code and can communicate with practically any external component such as message queues, in-memory key-value stores, and external databases. Therefore, when opting for Erlang, you aren't deprived of using existing third-party technologies. Instead, you have the option of using them when they're called for, not because your primary development platform doesn't give you a tool to solve 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 *is*.

1.1.4 The development platform

Erlang is more than a programming language. It's 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 runs in the Erlang virtual machine. It's a simple, functional language with basic concurrency primitives.

Source code written in Erlang is compiled into byte code that is then executed in the BEAM. This is where the true magic happens. The virtual machine parallelizes your concurrent Erlang programs and takes care of process isolation, distribution, and overall responsiveness of the system.

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

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

All these things can be done without OTP, but that makes no sense. OTP is battle tested in many production systems and is such an integral part of Erlang that it's hard to draw a 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 a BEAM instance, creating deployable releases, running the interactive shell, connecting to the running BEAM instance, and so on. Both BEAM and its 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 source, and you can find the source on the official site (http://erlang.org) or on the Erlang GitHub repository (https://github.com/erlang/otp). Ericsson is still in charge of the development process and releases a new version on a regular basis, once a year.

That concludes the story of Erlang. But if Erlang is so great, why do you need Elixir? The next section aims to answer this question.

1.2 About Elixir

Elixir is an alternative language for the Erlang virtual machine that allows you to write cleaner, more compact, code that does a better job of revealing your intentions. You write programs in Elixir and run them normally in BEAM.

Elixir is an open source project, originally started by José Valim. Unlike Erlang, Elixir is more of a collaborative effort; presently it has about 200 contributors. New features are frequently discussed on mailing lists, the GitHub issue tracker, and the #elixir-lang freenode IRC channel. José has the last word, but the entire project is 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 at https://github.com/elixir-lang/elixir.

Elixir targets the Erlang runtime. The result of compiling the Elixir source code are BEAM-compliant byte-code files that can run in a BEAM instance and can normally cooperate with pure Erlang code—you can use Erlang libraries from Elixir and vice versa. There is nothing you can do in Erlang that can't be done in Elixir, and usually the Elixir code is as performant as its Erlang counterpart.

Elixir is semantically close to Erlang: many of its language constructs map directly to the Erlang counterparts. But 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 and provides some nice syntactic sugar 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 is usually easier to develop and maintain.

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

1.2.1 Code simplification

One of the most important benefits of Elixir is the ability to radically reduce boilerplate and eliminate noise from code, which results in simpler code that is easier to write and maintain. Let's see what this means by contrasting Erlang and Elixir code. A frequently used building block in Erlang concurrent systems is the *server process*. You can think of server processes as something like concurrent objects—they embed private state and can interact with other processes via messages. Being concurrent, different processes may run in parallel. Typical Erlang systems rely heavily on processes, running thousands or even millions of them. The following example Erlang code implements a simple server process that adds two numbers.

Listing 1.1 Erlang-based server process that adds two numbers

```
-module(sum_server).

-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 of Erlang, this seems like a lot of code for something that only adds two numbers. To be fair, the addition is concurrent; 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 does. Moreover, it's difficult to write such code. Even after years of production-level Erlang development, I still can't write this without consulting the documentation or copying and pasting it from previously written code.

The problem with Erlang is that this boilerplate is almost impossible to remove, even if it's identical in most places (which in my experience is the case). The language provides almost no support for eliminating this noise. In all fairness, there is a way to reduce the boilerplate using a construct called `parse transform`, but it's clumsy and complicated to use. So in practice, Erlang developers write their server processes using the just-presented pattern.

Because server processes are an important and frequently used tool in Erlang, it's an unfortunate fact that Erlang developers have to constantly copy-paste this noise and work with it. Surprisingly, many people get used to it, probably due to the wonderful things BEAM does for them. It's often said that Erlang makes hard things easy and easy things hard. Still, the previous code leaves an impression that you should be able to do better.

Let's see the Elixir version of the same server process, presented in the next listing.

Listing 1.2 Elixir-based server process that adds two numbers

```
defmodule SumServer do
  use GenServer

  def start do
    GenServer.start(__MODULE__, nil)
  end

  def sum(server, a, b) do
    GenServer.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. Its intention is more clearly revealed, it's less burdened with noise. And yet it's as capable and flexible as the Erlang version. It behaves exactly the same at runtime and retains the complete semantics. There is nothing you can do in the Erlang version that's not possible in its Elixir counterpart.

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

But the language gives you tools to further eliminate whatever you may regard as noise and duplication. For example, I have developed my own Elixir library called ExActor that makes the server process definition dense, as shown next.

Listing 1.3 Elixir-based server process to add numbers based on a third-party abstraction

```
defmodule SumServer do
  use ExActor.GenServer

  defstart start

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

The intention of this code should be obvious even to developers with no previous Elixir experience. At runtime, the code works almost completely the same as the two previous versions. The transformation that makes this code behave like the previous ones happens at compile time. When it comes to the byte code, all three versions are similar.

> **NOTE** I mention the ExActor library only to illustrate how much you can abstract away in Elixir. But you won't use this library in this book, because it's a third-party abstraction that hides important details of how server processes work. To completely take advantage of server processes, it's important that you understand what makes them tick, which is why in this book you'll learn about lower-level abstractions. Once you understand how server processes work, you can decide for yourself whether you want to use ExActor to implement server processes.

This last implementation of the `sum` server process is powered by the Elixir macros facility. A *macro* is Elixir code that runs at *compile time*. A macro takes an internal representation of your source code as input and can create alternative output. Elixir macros are inspired by LISP and should not be confused with C-style macros. Unlike C/C++ macros, which work with pure text, Elixir macros work on abstract syntax tree (AST) structure, which makes it easier to perform nontrivial manipulations of the input code to obtain alternative output. Of course, Elixir provides helper constructs to simplify this transformation.

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

```
defcall sum(a, b) do
  reply(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
  GenServer.call(server, {:sum, a, b})
end

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

Because macros are written in Elixir, they're 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, the 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 that looks deceptively like the part of the language:

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

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

Elixir macros are something of a black art, but they make it possible to flush out nontrivial boilerplate at compile time and extend the language with your own DSL-like constructs. But Elixir isn't all about macros. Another worthy improvement is some 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 that transform data. One of the supposed benefits of this approach is that code is divided into many small, reusable, composable functions.

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

1. Apply the 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 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, you could eliminate the 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 it's clumsy and hard to read. To understand what goes on here, you have to manually parse it inside-out.

Although Erlang programmers are more or less limited to such clumsy approaches, Elixir gives you an elegant way to chain 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 to the next one as the first argument. The resulting code is clean, contains no temporary variables, and reads like the prose, top to bottom, left to right. Under the hood, this code is transformed at compile time to the staircased version. This is again possible because of Elixir's macro system.

The pipeline operator highlights the power of functional programming. You treat 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 approach. The API for standard libraries is cleaned up and follows some defined conventions. Syntactic sugar is introduced that simplifies typical idioms; and some Erlang data structures, such as the key-value dictionary and set, are rewritten to gain more performance. A concise syntax for working with structured data is provided. String manipulation is improved, and the language has explicit support for Unicode manipulation. In the tooling department, Elixir provides a `mix` tool that simplifies common tasks such as creating applications and libraries, managing dependencies, and compiling and testing code. In addition, a package manager called Hex (https://hex.pm/) is available that makes it simpler to package, distribute, and reuse dependencies.

The list goes on and on; but instead of presenting each feature, I'd like to express a personal sentiment based on my own production experience. Personally, I find it much more pleasant to code in Elixir. The resulting code seems simpler, more readable, and less burdened with boilerplate, noise, and duplication. At the same time, you retain the complete runtime characteristics of pure Erlang code. And you can use all the available libraries from the Erlang ecosystem, both standard and third party.

1.3 Disadvantages

No technology is a silver bullet, and Erlang and Elixir are definitely not exceptions. Thus it's worth mentioning some of their shortcomings.

1.3.1 Speed

Erlang is by no means the fastest platform out there. If you look at various synthetic benchmarks on the Internet, you usually won't see Erlang high on the list. Erlang programs are run in BEAM and therefore can't achieve the speed of machine-compiled languages, such as C and C++. But this isn't accidental or poor engineering on behalf of the Erlang/OTP team.

The goal of the platform isn't to squeeze out as many requests per seconds as possible, but to keep performance predictable and within limits. The level of performance your Erlang system achieves on a given machine shouldn't degrade significantly, meaning there shouldn't be unexpected system hiccups due to, for example, the garbage collector kicking in. Furthermore, as explained earlier, long-running BEAM processes don't block or significantly impact the rest of the system. Finally, as the load increases, BEAM can use as many hardware resources as possible. If the hardware capacity isn't enough, you can expect graceful system degradation—requests will take longer to process, but the system won't be paralyzed. This is due to the preemptive nature of the BEAM scheduler, which performs frequent context switches that keep the system ticking and favors short-running processes. And of course, you can address higher system demand by adding more hardware.

Nevertheless, intensive CPU computations aren't as performant as, for example, their C/C++ counterparts, so you may consider implementing such tasks in some other language and then integrating the corresponding component into your Erlang system. If most of your system's logic is heavily CPU bound, then you should probably consider some other technology.

1.3.2 Ecosystem

The ecosystem built around Erlang isn't small, but it definitely isn't as big as that of some other languages. At the time of writing, a quick search on GitHub reveals just over 11,000 Erlang-based repositories. In contrast, there are more than 600,000 Ruby repositories and almost 1,000,000 for JavaScript. Elixir currently has slightly over 2,000, which is impressive given its young age.

Numbers aside, I have mixed feeling about the available Erlang-based libraries. I can usually find client libraries for what I need (such as MongoDb and 0MQ), but I sometimes have the feeling they aren't mature as I would expect: they either lack proper documentation or don't support all possible features. On the plus side, the community is usually supportive, and I have often seen developers of third-party components provide assistance and extend their libraries if requested.

Regardless, you should be prepared that the choice of libraries won't be as abundant as you may be used to, and in turn you may end up spending extra time on something that would take minutes in other languages. If that happens, keep in mind all the benefits you get from Erlang. As I've explained, Erlang goes a long way toward making it possible to write fault-tolerant systems that can run for a long time with hardly any downtime. This is a big challenge and a specific focus of the Erlang platform. So although it's admittedly unfortunate that the ecosystem isn't as mature as it could be, my sentiment is that Erlang significantly helps with hard problems, even if simple problems can sometimes be more clumsy to solve. Of course, those difficult problems may not always be important. Perhaps you don't expect a high load, or a system doesn't need to run constantly and be extremely fault-tolerant. In such cases, you may want to consider some other technology stack with a more evolved ecosystem.

Elixir's ecosystem currently isn't as mature as Erlang's, but you can use practically any Erlang library from Elixir (with some minor exceptions). Because the language is gaining traction and offers some compelling benefits versus Erlang, this will hopefully improve, and ultimately the Erlang ecosystem will improve as well.

1.4 Summary

This chapter defined the purpose and benefits of Erlang and Elixir. There are a couple of points worth remembering:

- Erlang is a technology for developing highly available systems that constantly provide service with 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 development for the Erlang platform much more pleasant. It helps organize code more efficiently and abstracts away boilerplate, noise, and duplication.

Now you can start learning how to develop Elixir-based systems. In the next chapter, you learn about the basic building blocks of Elixir programs.