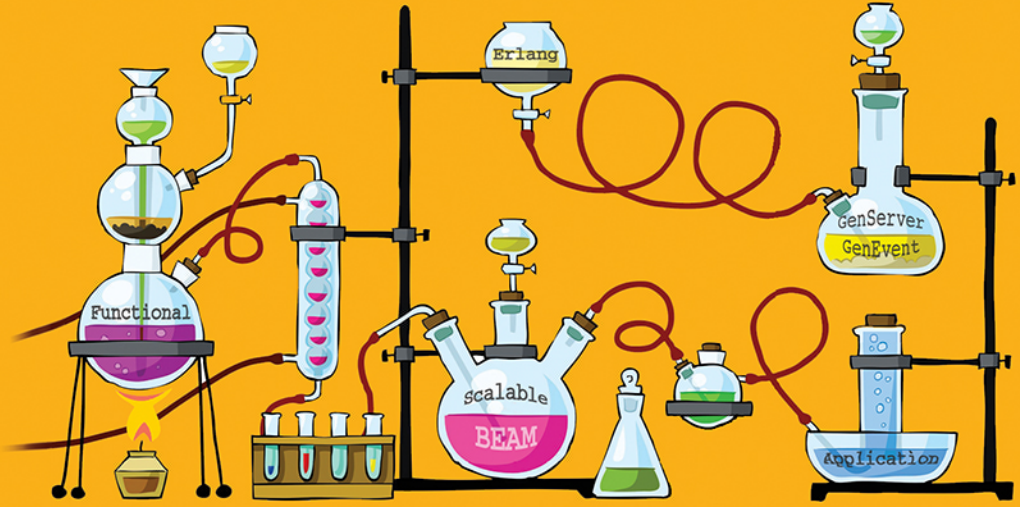
THE LITTLE Elixir & OTP GUIDEBOOK



Benjamin Tan Wei Hao

SAMPLE CHAPTER

MANNING

The Little Elixir & OTP Guidebook

by Tan Wei Hao

Chapter 1

brief contents

1 Introduction

This chapter covers

- What Elixir is
- How Elixir is different from Erlang
- Why Elixir is a good choice
- What Elixir/OTP is good for
- The road ahead

Just in case you bought this book for medicinal purposes—I'm sorry, wrong book. This book is about Elixir the programming language. No other language (other than Ruby) has made me so excited and happy to work with it. Even after spending more than two years of my life writing about Elixir, I still love programming in it. There's something special about being involved in a community that's so young and lively. I don't think any language has had at least *four* books written about it, a dedicated screencast series, and a conference—all before v1.0. I think we're on to something here.

Before I begin discussing Elixir, I want to talk about Erlang and its legendary virtual machine (VM), because Elixir is built on top of it. Erlang is a programming language that excels in building soft real-time, distributed, and concurrent systems. Its original use case was to program Ericsson's telephone switches. (Telephone switches are basically machines that connect calls between callers.)

These switches had to be concurrent, reliable, and scalable. They had to be able to handle multiple calls at the same time, and they also had to be extremely reliable—no one wants their call to be dropped halfway through. Additionally, a dropped call (due to a software or hardware fault) shouldn't affect the rest of the calls on the switch. The switches had to be massively scalable and work with a distributed network of switches. These production requirements shaped Erlang into what it is today; they're the exact requirements we have today with multicore and web-scale programming.

As you'll discover in later chapters, the Erlang VM's scheduler automatically distributes workloads across processors. This means you get an increase in speed *almost* for free if you run your program on a machine with more processors—*almost*, because you'll need to change the way you approach writing programs in Erlang and Elixir in order to reap the full benefits. Writing distributed programs—that is, programs that are running on different computers and that can communicate with each other—requires little ceremony.

1.1 Elixir

It's time to introduce Elixir. Elixir describes itself as *a functional, meta-programming-aware language built on top of the Erlang virtual machine*. Let's take this definition apart piece by piece.

Elixir is a *functional programming language*. This means it has all the usual features you expect, such as immutable state, higher-order functions, lazy evaluation, and pattern matching. You'll meet all of these features and more in later chapters.

Elixir is also a *meta-programmable language*. Meta-programming involves code that generates code (black magic, if you will). This is possible because code can be represented as data, and data can be represented as code. These facilities enable the programmer to add to the language new constructs (among other things) that other languages find difficult or even downright impossible.

This book is also about OTP, a framework to build fault-tolerant, scalable, distributed applications. It's important to recognize that Elixir essentially gains OTP for free because OTP comes as part of the Erlang distribution. Unlike most frameworks, OTP comes packaged with a lot of good stuff, including three kinds of databases, a set of debugging tools, profilers, a test framework, and much more. Although we only manage to play with a tiny subset, this book will give you a taste of the pure awesomeness of OTP.

NOTE OTP used to be an acronym for *Open Telecom Platform*, which hints at Erlang's telecom heritage. It also demonstrates how naming is difficult in computer science: OTP is a general-purpose framework and has little to do with telecom. Nowadays, OTP is just plain OTP, just as *IBM* is just *IBM*.

1.2 How is Elixir different from Erlang?

Before I talk about how Elixir is different from Erlang, let's look at their similarities. Both Elixir and Erlang compile down to the same bytecode. This means both Elixir and Erlang programs, when compiled, emit instructions that run on the same VM.

Another wonderful feature of Elixir is that you can call Erlang code directly from Elixir, and vice versa! If, for example, you find that Elixir lacks a certain functionality that's present in Erlang, you can call the Erlang library function directly from your Elixir code.

Elixir follows most of Erlang's semantics, such as message passing. Most Erlang programmers would feel right at home with Elixir.

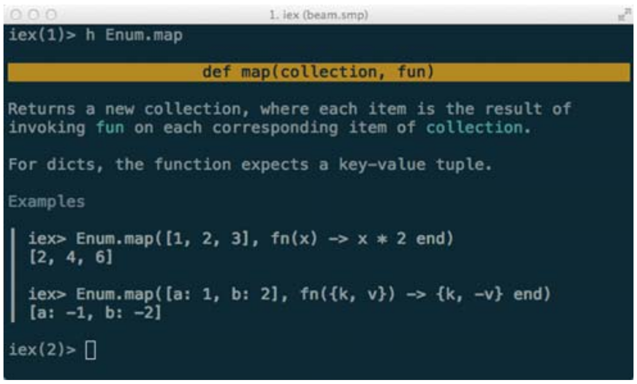
This interoperability also means a wealth of Erlang third-party libraries are at the disposal of the Elixir developer (that's you!). So why would you want to use Elixir instead of Erlang? There are at least two reasons: the tooling and ecosystem.

1.2.1 Tooling

Out of the box, Elixir comes with a few handy tools built in.

Interactive Elixir

The Interactive Elixir shell (`iex`) is a read-eval-print loop (REPL) that's similar to Ruby's `irb`. It comes with some pretty nifty features, such as syntax highlighting and a beautiful documentation system, as shown in figure 1.1.



```
iex(1)> h Enum.map

                    def map(collection, fun)

Returns a new collection, where each item is the result of
invoking fun on each corresponding item of collection.

For dicts, the function expects a key-value tuple.

Examples

  iex> Enum.map([1, 2, 3], fn(x) -> x * 2 end)
  [2, 4, 6]

  iex> Enum.map([a: 1, b: 2], fn({k, v}) -> {k, -v} end)
  [a: -1, b: -2]

iex(2)>
```

Figure 1.1 Interactive Elixir has documentation built in.

There's more to `iex`: this tool allows you to connect to *nodes*, which you can think of as separate Erlang runtimes that can talk to each other. Each runtime can live on the same computer, the same LAN, or the same network.

`iex` has another superpower, inspired by the Ruby library Pry. If you've used Pry, you know that it's a debugger that allows you to pry into the state of your program. `iex` comes with a similarly named function called `IEx.pry`. You won't use this feature in the book, but it's an invaluable tool to be familiar with. Here's a brief overview of how to use it. Let's assume you have code like this:

```
require IEx

defmodule Greeter do
  def ohai(who, adjective) do
    greeting = "Ohai!, #{adjective} #{who}"
    IEx.pry
  end
end
```

The `IEx.pry` line will cause the interpreter to pause, allowing you to inspect the variables that have been passed in. First you run the function:

```
iex(1)> Greeter.ohai "leader", "glorious"
Request to pry #PID<0.62.0> at ohai.ex:6

      def ohai(who, adjective) do
        greeting = "Ohai!, #{adjective} #{who}"
        IEx.pry
      end
    end

Allow? [Yn] Y
```

Once you answer Yes, you're brought into `iex`, where you can inspect the variables that were passed in:

```
Interactive Elixir (1.2.4) - press Ctrl+C to exit (type h() ENTER for help)
pry(1)> who
"leader"
pry(2)> adjective
"glorious"
```

There are other nice features, like autocomplete, that you'll find handy when using `iex`. Almost every release of Elixir includes useful improvements and additional helper functions in `iex`, so it's worth keeping up with the changelog!

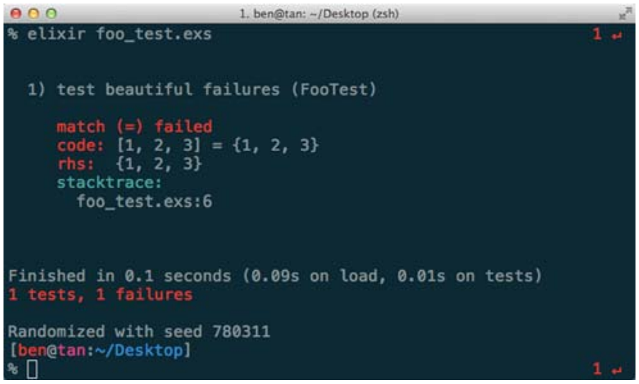
TESTING WITH EXUNIT

Testing aficionados will be pleased to know that Elixir has a built-in test framework called ExUnit. ExUnit has some useful features such as being able to run asynchronously and produce beautiful failure messages, as shown in figure 1.2. ExUnit can perform nifty tricks with error reporting mainly due to macros, which I won't cover in this book. Nonetheless, it's a fascinating topic that you may want to explore.[1]

MIX

`mix` is a build tool used for creating, compiling, and testing Elixir projects. It's also used to manage dependencies, among other things. Think of it like `rake` in Ruby and `lein` in Clojure. (Some of the first contributors to `mix` also wrote `lein`.) Projects such as the Phoenix web framework have used `mix` to great effect for things like building generators that reduce the need to write boilerplate.

[1] http://elixir-lang.org/getting-started/meta/macros.html.



```
% elixir foo_test.exs                                    1 ↵


  1) test beautiful failures (FooTest)

     match (=) failed
     code: [1, 2, 3] = {1, 2, 3}
     rhs:  {1, 2, 3}
     stacktrace:
       foo_test.exs:6


Finished in 0.1 seconds (0.09s on load, 0.01s on tests)
1 tests, 1 failures

Randomized with seed 780311
[ben@tan:~/Desktop]
%                                                        1 ↵
```

Figure 1.2 ExUnit comes with excellent error messages.

Standard library

Elixir ships with an excellent standard library. Data structures such as ranges, strict and lazy enumeration APIs, and a sane way to manipulate strings are just some of the nice items that come packaged in it.

Although Elixir may not be the best language in which to write scripts, it includes familiar-sounding libraries such as `Path` and `File`. The documentation is also a joy to use. Explanations are clear and concise, with examples of how to use the various libraries and functions.

Elixir has modules that aren't in the standard Erlang library. My favorite of these is `Stream`. Streams are basically composable, lazy enumerables. They're often used to model potentially infinite streams of values.

Elixir has also added functionality to the OTP framework. For example, it's added a number of abstractions, such as `Agent` to handle state and `Task` to handle one-off asynchronous computation. `Agent` is built on `GenServer` (this stands for *generic server*), which comes with OTP by default.

Metaprogramming

Elixir has LISP-like macros built into it, minus the parentheses. Macros are used to extend the Elixir language by giving it new constructs expressed in existing ones. The implementation employs the use of macros throughout the language. Library authors also use them extensively to cut down on boilerplate code.

1.2.2 Ecosystem

Elixir is a relatively new programming language, and being built on top of a solid, proven language definitely has its advantages.

Thank you, Erlang!

I think the biggest benefit for Elixir is the years of experience and tooling available from the Erlang community. Almost any Erlang library can be used in Elixir with little effort. Elixir developers don't have to reinvent the wheel in order to build rock-solid applications. Instead, they can happily rely on OTP and can focus on building additional abstractions based on existing libraries.

Learning resources

The excitement around Elixir has led to a wellspring of learning resources (not to beat my own drum). There are already multiple sources for screencasts, as well as books and conferences. Once you've learned to translate from Elixir to Erlang, you can also benefit from the numerous well-written Erlang books, such as *Erlang and OTP in Action* by Martin Logan, Eric Merritt, and Richard Carlsson (Manning Publications, 2010); *Learn You Some Erlang for Great Good!* by Fred Hébert (No Starch Press, 2013); and *Designing for Scalability with Erlang/OTP* by Francesco Cesarini and Steve Vinoski (O'Reilly Media, 2016).

Phoenix

Phoenix is a web framework written in Elixir that has gotten a lot of developers excited, and for good reason. For starters, response times in Phoenix can reach microseconds. Phoenix proves that you can have both high performance and a simple framework coupled with built-in support for WebSockets and backed by the awesome power of OTP.

It's still evolving

Elixir is constantly evolving and exploring new ideas. One of the most interesting notions I've seen arise are the concurrency abstractions that are being worked on. Even better, the Elixir core team is always on the hunt for great ideas from other languages. There's already (at least!) Ruby, Clojure, and F# DNA in Elixir, if you know where to look.

1.3 Why Elixir and not X?

On many occasions, when I give a talk about Elixir or write about it, the same question pops up: "Should I learn Elixir instead of *X*?" *X* is usually Clojure, Scala, or Golang. This question usually stems from two other questions: "Is Elixir gaining traction?" and "Are jobs available in Elixir?" This section presents my responses.

Elixir is a young language (around five years old at the time of writing), so it will take time for the language, ecosystem, and community to mature. You can use this to your advantage. First, functional programming is on the rise, and certain principles remain more or less the same in most functional programming languages. Whether it's Scala, Clojure, or Erlang, these skills are portable.

Erlang seems to be gaining popularity. There's also a surge of interest in distributed systems and the internet of things (IoT), domains that are right up Elixir's alley.

I have a gut feeling that Elixir will take off soon. It's like Java in its early days: not many people bothered with it when it first came out, but the early adopters were hugely rewarded. The same went for Ruby. There's definitely an advantage to being ahead of the curve.

It would be selfish of me to keep everyone else from learning and experiencing this wonderful language. Cast your doubts aside, have a little faith, and enjoy the ride!

1.4 What is Elixir/OTP good for?

Everything that Erlang is great for also applies to Elixir. Elixir and OTP combined provide facilities to build concurrent, scalable, fault-tolerant, distributed programs. These include, but obviously aren't limited to, the following:

- Chat servers (WhatsApp, ejabberd)
- Game servers (Wooga)
- Web frameworks (Phoenix)
- Distributed databases (Riak and CouchDB)
- Real-time bidding servers
- Video-streaming services
- Long-running services/daemons
- Command-line applications

From this list, you probably gather that Elixir is ideal for building server-side software—and you're right! These software programs share similar characteristics. They have to

- Serve multiple users and clients, often numbering in the thousands or millions, while maintaining a decent level of responsiveness
- Stay up in the event of failure, or have graceful failover mechanisms
- Scale gracefully by adding either more CPU cores or additional machines

Elixir is no wonder drug (pun intended). You probably won't want to do any image processing, perform computationally intensive tasks, or build GUI applications on Elixir. And you wouldn't use Elixir to build hard real-time systems. For example, you shouldn't use Elixir to write software for an F-22 fighter jet.

But hey, don't let me tell you what you can or can't do with Elixir. Let your creativity flow. That's why programming is so awesome.

1.5 The road ahead

Now that I've given you some background on Elixir, Erlang, and the OTP framework, the following appetite-whetting sections provide a high-level overview of what's to come.

1.5.1 A sneak preview of OTP behaviors

Say you want to build a weather application. You decide to get some venture capital, and before you know it, you're funded.

After some thinking, you realize that what you're building essentially is a simple client-server application. Of course, you don't tell your investors this. Basically, clients (via HTTP, for example) will make requests, and your application will perform some computations and return the results to each client in a timely manner.

You implement your weather application, and it goes viral! But suddenly your users begin to encounter all sorts of issues: slow load times and, even worse, service disruptions. You attempt to do some performance profiling, you tweak settings here and there, and you try to add more concurrency.

Everything seems OK for a while, but that's just the calm before the storm. Eventually, users experience the same issues again, plus they see error messages, mysterious deadlocks occur, and other weird issues appear. In the end, you give up and write a long blog post about how your startup failed and why you should have built the application in Node.js or Golang. The post is #1 on Hacker News for a month. You then stumble upon OTP and learn that Elixir combined with OTP can be used to build concurrent, scalable, fault-tolerant, distributed programs.

Although this book won't explain how to get venture capital, it will show you how to build a weather service using OTP, among other fun things. The OTP framework is what gives BEAM languages (Erlang, Elixir, and so on) their superpowers, and it comes bundled with Elixir.

One of the most important concepts in OTP is the notion of *behaviors*. A behavior can be thought of as a contract between you and OTP.

When you use a behavior, OTP expects you to fill in certain functions. In exchange for that, OTP takes care of a slew of issues such as message handling (synchronous or asynchronous), concurrency errors (deadlocks and race conditions), fault tolerance, and failure handling. These issues are general—almost every respectable client/server program has to handle them somehow, but OTP steps in and handles all of these for you. Furthermore, these generic bits have been used in production and battle-tested for years.

In this book, you'll work with two of the most-used behaviors: `GenServer` and `Supervisor`. Once you're comfortable with them, learning to use other behaviors will be straightforward. You could roll your own `Supervisor` behavior, but there's no good reason to do so 99.999999999% of the time. The implementers have thought long and hard about the features that need to be included in most client-server programs, and they've also accounted for concurrency errors and all sorts of edge cases.

How do you use an OTP behavior? The following listing shows a minimal implementation of a weather service that uses `GenServer`.

Listing 1.1 Example `GenServer`

```
defmodule WeatherService do
  use GenServe4r # <- This brings in GenServer behavior

  def handle_call({:temperature, city}, _from, state) do    <-- Synchronous request
    # ...
  end

  def handle_cast({:email_weather_report, email}, state) do    <-- Asynchronous request
    # ...
  end

end
```

This implementation is obviously incomplete; the important thing to realize (and you'll see this as you work through the book) is how many things you *don't* need to do. For example, you don't have to implement how to handle a synchronous or an asynchronous request. I'll leave you in suspense for now (this is just a sneak preview), but in chapters 3 and 4 you'll build the same application without OTP and then with OTP.

OTP may look complicated or scary at first sight, but you'll see that this isn't the case as you work through the examples in the book. The best way to learn how something works is to implement it yourself. In that spirit, you'll learn how to implement the `Supervisor` behavior from scratch in chapter 5. The point is to demonstrate that there's little magic involved—the language provides the necessary tools to build out these useful abstractions.

You'll also implement a worker pool application from scratch and evolve it in stages in chapters 6 and 7. This will build on the discussion of `GenServer` and `Supervisor`.

1.5.2 Distribution for load balancing and fault tolerance

Elixir with OTP is an excellent candidate to build distributed systems. In this book, you'll build two distributed applications, highlighting two different uses of distribution.

One reason you might want to create a distributed application is to spread the load across multiple computers. In chapter 8, you'll create a load tester and see how you can exploit distribution to scale up the capabilities of your application. You'll see how Elixir's message-passing-oriented nature and the distribution primitives available make building distributed applications a much more pleasant experience compared to other languages and platforms.

Another reason you might require distribution is to provide fault tolerance. If one node fails, you want another node to stand in its place. In chapter 9, you'll see how to create an application that does this, too.

1.5.3 Dialyzer and type specifications

Because Elixir is a dynamic language, you need to be wary of introducing type errors in your programs. Therefore, one aspect of reliability is making sure your programs are type-safe.

Dialyzer is a tool in OTP that aims to detect some of these problems. You'll learn how to use Dialyzer in a series of examples in chapter 10. You'll also learn about Dialyzer's limitations and how to overcome some of them using type specifications. As you'll see, type specifications, in addition to helping Dialyzer, serve as documentation. For example, the following listing is taken from the `List` module.

Listing 1.2 Function that has been annotated with type specifications

```
@spec foldl([elem], acc, (elem, acc -> acc)) :: acc when elem: var, acc: var
def foldl(list, acc, function) when is_list(list) and is_function(function)
     ➥do
  :lists.foldl(function, acc, list)
end
```

After reading chapter 10, you'll appreciate type specifications and how they can help make your programs clearer and safer.

1.5.4 Property and concurrency testing

Chapter 11 is dedicated to property-based and concurrency testing. In particular, you'll learn how to use QuickCheck and Concuerror. These tools don't come with Elixir or OTP by default, but they're extremely useful for revealing bugs that traditional unit-testing tools don't.

You'll learn about using QuickCheck for property-based testing and how property-based testing turns traditional unit testing on its head. Instead of thinking about specific examples, as in unit testing, property-based testing forces you to come up with general properties your tested code should hold. Once you've created a property, you can test it against hundreds or thousands of generated test inputs. Here's an example that says reversing a list twice gives you back the same list:

```
@tag numtests: 100
property "reverse is idempotent" do
  forall l <- list(char) do
    ensure l |> Enum.reverse |> Enum.reverse == l
  end
end
```

This code generates 100 lists and asserts that the property holds for each of those generated lists.

The other tool we'll explore in chapter 11 is Concuerror, which was born in academia but has seen real-world use. You'll learn how Concuerror reveals hard-to-detect concurrency bugs such as deadlocks and race conditions. Through a series of intentionally buggy examples, you'll use Concuerror to disclose the bugs.

1.6 Summary

In this chapter, I introduced Elixir and Erlang. In addition, you learned about the following:

- The motivations behind the creation of Erlang, and how it fits perfectly into the multi-core and web-scale phenomena we have today
- The motivations behind the creation of Elixir, and a few reasons Elixir is better than Erlang, such as Elixir's standard library and tool chain
- Examples for which Elixir and OTP are perfect use cases

PROGRAMMING

THE LITTLE Elixir & OTP GUIDEBOOK

Benjamin Tan Wei Hao

Free eBook
SEE INSERT

Elixir is an elegant programming language that combines the expressiveness of Ruby with the concurrency and fault-tolerance of Erlang. It makes full use of Erlang's BEAM VM and OTP library, so you get two decades' worth of maturity and reliability right out of the gate. Elixir's support for functional programming makes it perfect for modern event-driven applications.

The Little Elixir & OTP Guidebook gets you started writing applications with Elixir and OTP. You'll begin with the immediately comfortable Elixir language syntax, along with just enough functional programming to use it effectively. Then, you'll dive straight into several lighthearted examples that teach you to take advantage of the incredible functionality built into the OTP library.

WHAT'S INSIDE

- Covers Elixir 1.2 and 1.3
- Introduction to functional concurrency with actors
- Experience the awesome power of Erlang and OTP

Written for readers comfortable with a standard programming language like Ruby, Java, or Python. FP experience is helpful but not required.

Benjamin Tan Wei Hao is a software engineer at Pivotal Labs, Singapore. He is also an author, a speaker, and an early adopter of Elixir.

"Move seamlessly from learning the basics of Elixir to mastering the key concepts of OTP."

—Roberto Infante, Devqf Ltd.

"Engaging. Practical. Informative. Thumbs up!"

—Dane Balia, Hetzner

"If you've never touched Elixir or Erlang before, this book will open the door to a new universe for you."

—Thomas Peklak, Emakina CEE

"Offers techniques and insights difficult or impossible to find anywhere else."

—Kosmas Chatzimichalis, Mach7x

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/the-little-elixir-and-otp-guidebook

ISBN-13: 978-1-63343-011-2
ISBN-10: 1-63343-011-1
53999
9 781633 430112

MANNING US $39.99 / Can $45.99 [including eBook]