Extracted from:

Adopting Elixir

From Concept to Production

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Your Elixir Source

Adopting Elixir

From Concept to Production

Ben Marx, José Valim, Bruce Tate

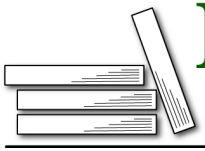
edited by Jacquelyn Carter

Adopting Elixir

From Concept to Production

Ben Marx
José Valim
Bruce Tate

The Pragmatic Bookshelf
Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Brian MacDonald
Supervising Editor: Jacquelyn Carter
Copy Editor: Jasmine Kwityn
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-252-7
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—March 2018

You've made the business case for Elixir, and started shaping your team with the right building blocks for personal growth and consistency. You have read the getting started guide, consulted the documentation, and reviewed some of the many books available, yet something is still missing. That's only natural.

If you and your team are familiar with Elixir and functional programming, you might skip ahead to the next chapter but we know from our research that a fair number of our readers are not quite comfortable with Elixir. Here's what we mean. If you've ever watched a non-native speaker learn any spoken language, you probably saw them borrow native language concepts that didn't quite fit. On this team, José is famous for his English puns, but occasionally he'll try one that has us all scratching our heads.

Learning Elixir is like that. The basics take time, and even after establishing the fundamentals, questions will remain in the journey from apprentice to master. Object-oriented developers adopting functional languages tend to try to reinvent object-oriented concepts in them. It's common for such users to have questions:

- If you are coming from an object-oriented background, what does it take to properly design applications in a functional and concurrent programming language like Elixir?
- When are modules and functions enough and when should we resort to processes?
- What's a `GenServer`, and why is it one of the most prevalent Elixir abstractions?
- What role does `Supervisor` play in building applications?

Each of these concepts is new to teams who code object-oriented applications that only dabble in concurrency. In this chapter, we will talk about these questions and more. We will cover higher level concepts and abstractions. Internalizing these foundational concepts will speed your adoption curve tremendously. Along the way, you will see examples that will provide a mental framework that lets you put your newly acquired knowledge to use. We expect that you are already familiar with Elixir data types such as lists, tuples, and maps. You will also need to know about abstractions such as tasks and agents.

Let's go beyond the basics. We want to help you apply foundational Elixir concepts in the context of the complex applications you'll encounter in the real world. Let's start with one of the most fundamental concepts of functional programming languages: immutability.

Elixir vs. Mutable Objects

Since Elixir is a functional programming language, it does not have objects. The language also has a strong focus on immutability. In Elixir we *transform* data rather than *mutate* it.

Said another way: *OO changes. FP copies.*

While this difference may be subtle and might even seem inefficient, it's transformational. Many of Elixir's most important benefits flow directly from this design decision. In this section, we're going to look at what those benefits might be, and why they matter to you. Let's take that apart.

Understanding Mutation

Mutable objects bundle three concerns that are distinct in Elixir: state, behavior, and time. Take this example:

```
dictionary.store("key", "value")
```

If this were like most object-oriented programs, `dictionary` would be an object holding a dictionary with multiple keys and values, probably in the form of a hash. That object would provide a `store` method that changes the hash *in place*. It's this in-place change that we call a mutation.

In object-oriented languages, mutations represent `time` because the value of the object will depend on when you access it. If you access that dictionary after a mutation, you get the new version and the old version no longer exists. Such changes are very hard to track, especially when more than one client uses the same piece of code. Adding more objects often introduces more moving parts, with little visibility on how those parts change through time; adding concurrency makes reasoning about such code nearly impossible.

Elixir decouples these three concepts. Data structures are immutable and represent state. Modules define our behavior. Processes send and receive messages, embodying the concept of time.

The previous code would be written in Elixir as:

```
new_map = Map.put(map, "key", "value")
```

`map` is the data and `Map.put/3` is a function defined in the `Map` module that receives three arguments. `Map.put/3` never mutates the map; it always returns a new one. `map` will never change so for this code:

```
value1 = Map.get(map, "key")
# ...
```

```
# Some other code
# ...
value2 = Map.get(map, "key")
```

value1 and value2 *will always be the same* unless map is reassigned to another value somewhere between the two calls. And even if you rebind the map variable, the underlying map does not change. The variable is just pointing somewhere new.

Now you have a guarantee. The map referenced by the variable map will never change, even if some other code is holding a reference to the same map, and *that* guarantee makes all of the difference in the world. We pass the map around confident in the fact that no other code can modify it.

If you want to intentionally violate this guarantee, you'll need to reach for another abstraction, the process. We're going to hold a tiny bit of state in another process, an agent, and we'll communicate with that process as needed. Consider this counter:

```
{:ok, pid} = Agent.start_link(fn -> 0 end)
value1 = Agent.get(pid, fn x -> x end)
Agent.update(pid, fn x -> x + 1 end)
value2 = Agent.get(pid, fn x -> x end)
```

In this example, calling Agent.get/2 with the exact same arguments may give you different results. This arrangement lets you save state using separate processes. Since you can only communicate with a process via explicit messages, Elixir allows developers to reason about how their application state changes over time. *In effect, processes such as agents isolate state change with the explicit, formal set of rules governing message passing.*

If you wanted to, you could use agents, or files, or any other stateful abstraction as mutable variables, and completely undo Elixir's stateless advantages. In fact, many beginners fall into this trap. Good languages sometimes let you run with scissors. Elixir's important decision in this regard is to keep these choices explicit. An agent *feels* like a more serious commitment than a mutable variable because it *is* a more serious level of commitment.

While time adds complexity to our applications, functional programming is about making the complex parts of our system explicit. By modeling state changes with processes and message-passing, we make our software easier to understand, simpler to write, and much more stable.

Elixir as an Object-Oriented Language

You may have heard that Elixir processes are objects, according to Dr. Alan Kay's definition of "object-oriented programming." In an email discussion with Stefan Ram,[a] Kay coined the term object-oriented programming and says "OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things."

While Elixir processes do neatly fit that description, we think the comparison may cause more confusion than insight, as processes should not be used as a code design tool in the same way objects are used in most object-oriented programming languages.

a. http://www.purl.org/stefan_ram/pub/doc_kay_oop_en

Immutability and Memory

The pipe operator |> is one of the first constructs Elixir developers learn, as it embodies transformation and the decoupling between data and behavior. When we pipe between functions, it receives all data it needs as input and returns all relevant information as output. There's never hidden or mutated data. Each pipe is a standalone transformation with an explicit contract.

When writing your business logic, you may use Ecto[1] changesets to handle data casting and validation:

```
def changeset(user, params \\ %{}) do
  user
  |> cast(params, [:name, :email, :age])
  |> validate_required([:name, :email])
  |> validate_format(:email, ~r/@/)
  |> validate_inclusion(:age, 18..120)
  |> unique_constraint(:email)
end
```

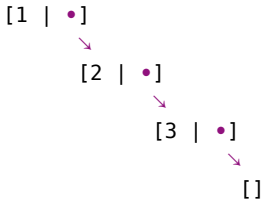
Each function along the way transforms the changeset. You may be asking yourself about the cost of immutability. Many developers assume that each time you change a map or a struct, Elixir creates a whole new one in memory. That's not true.

Elixir represents a map with multiple elements as a tree in memory. Adding, putting, or deleting an element requires changing only the path to that element on the tree. All other elements in the map are shared between the old map and newly transformed map. Let's explore how that sharing works with a list example.

1. https://github.com/elixir-ecto/ecto

Elixir represents lists internally as *cons cells*. Each cons cell is a simple data structure with two elements in the [left | right] form.

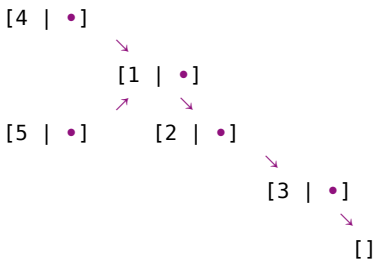
Lists are nested cons cells. The list [1, 2, 3] expressed with cons cells is [1 | [2 | [3 | []]]]. In memory, it would be represented like this:



Let's see what happens when we create a new list from an old one. Consider this code:

```
iex> list = [1, 2, 3]
[1, 2, 3]
iex> first = [4 | list]
[4, 1, 2, 3]
iex> second = [5 | list]
[5, 1, 2, 3]
```

Elixir *does not need to create two full copies*. It simply needs to create two new cons cells, one with 4 and list and another with 5 and list, like this:



That's why prepending to lists is *always* fast, while appending is slow. Prepending enables sharing, appending requires us to copy the whole list since we need to change the last cons cell to point somewhere else.

The exact transformation mechanisms and costs depend on the data structure, and we'll not go into them here. What's important is that *immutability is exactly what makes this kind of transformation efficient*, because the VM knows the data underneath *is not going to change*. For example, if you have a tuple with three elements, {:one, 2, "three"}, in memory you have a tuple container that points to :one, 2, and "three". If you change the second element of the tuple, you get a new tuple, but it will still point to the same :one and "three" exactly

because, even if another piece of the code is holding a reference to the old tuple, no one can mutate any of its contents.

This immutability contract gives Elixir tremendous freedom. Think about this simple function:

```
def one_two_three do
  [1, 2, 3]
end
```

Other languages that support mutability would likely need to return a separate copy of the list upon each invocation because each client could mutate the list. Elixir doesn't have that restriction. Each time you invoke that function, you'll get the *same exact list in the same exact memory address* because nobody will ever change it.

Immutability makes our software easier to understand and also introduces simplifications at the compiler level that make it easier to share data throughout.

There's a cost, though. In some situations immutability may have performance implications. Even though the language relies on advanced techniques such as sharing, a piece of code needing to execute millions of operations per second on the same data structure may generate an unnecessary amount of garbage. In such cases, you may need to resort to the mutable components available in Elixir, such as ETS or the process dictionary.

However, it is worth pointing out that in our 10 years of collective experience working with Elixir, we recall such performance-centric optimization was necessary only once, when implementing a data-processing engine.

Data and Behavior

By separating data and behavior, Elixir allows developers to focus on the shape of the data. The code is more explicit than languages that don't do so, and explicit code makes its intentions clear. Consider this OO code:

```
URI.parse(url).path.split("/").last
```

Each . makes it hard to track the source of each method. You might ask yourself "Where does split("/") come from?" Maybe it is a String method, or maybe there is a Path object in there somewhere. You just don't know.

Contrast that example with this one in Elixir, where each operation along the way is explicitly named:

```
URI.parse(url).path
|> String.split("/")
```

```
|> List.last
```

Granted, the Elixir version is more verbose. In exchange, you and your editor know exactly where each function comes from. The use of the pipe operator clarifies each step in the transformation. Each step transforms the data but never mutates it.