Extracted from:

Designing Elixir Systems with OTP

Write Highly Scalable, Self-Healing Software with Layers

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Designing Elixir Systems with OTP

Write Highly Scalable, Self-Healing Software with Layers

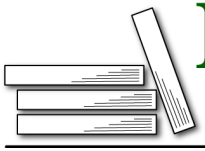
James Edward Gray, II
Bruce A. Tate
edited by Jacquelyn Carter

Designing Elixir Systems with OTP

Write Highly Scalable, Self-Healing Software with Layers

James Edward Gray, II
Bruce A. Tate

The Pragmatic Bookshelf
Raleigh, North Carolina



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
Executive Editor: Dave Rankin
Development Editor: Jacquelyn Carter
Copy Editor: Jasmine Kwytin
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*.

ISBN-13: 978-1-68050-661-7
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—December 2019

CHAPTER 1

Build Your Project in Layers

Don't let anyone tell you differently. Building great software is hard, and Elixir's not a silver bullet. Though it makes dealing with processes *easier*, concurrent code will never be *easy*. If your checklist includes intimidating scalability requirements, performance consistency under load, or highly interactive experiences or the like, programming gets harder still. In this book, we won't shy away from these demands.

If you're like us, you found a valuable companion in Elixir, with some characteristics you believe can help you with some of these challenges, even if you don't fully understand it. Perhaps Elixir is your first functional language, as it is for many of us. You may need some guidance for how to choose your data structures or organize your functions. Or, you might have found several ways to deal with concurrency and need some advice on which approach to use.

We can tell you definitively that you're not alone and we're here to help. We won't offer panaceas, or full solutions to toy problems that have general advice about *design*. We will offer some mental models for how to deal with complexity piece by piece.

With most any new endeavor, progress comes at a price. Our first payment is a willingness to change.

We Must Reimagine Design Choices

We believe good software design is about building layers, so perhaps the most important aspect of this book is *helping good programmers understand where layers should go and how they work.* Some of the techniques that we used when the internet was young are not the ones we'll be using into the future, but take heart. This author team doesn't have all of the answers, but both of us have a strong corpus to draw from.

Some of our inspiration comes from the past. Throughout this book, we're going to distill much of the conventional wisdom from functional programmers and we're not shy about crossing language boundaries to learn. We're going to draw on the expertise of Elixir programmers, including many of the people who shaped the language as it was formed.

We'll also draw inspiration from Erlang, Clojure, and Elm for algorithms and techniques to solve problems similar to the ones we're facing as we determine what the right set of layers should be. We'll rely heavily on Erlang, especially the OTP framework that helps manage concurrency state and lifecycle.

This book is about design, and because Elixir heavily uses OTP, we must address how to construct layers around an OTP program. Let's define that term quickly with a brief generality. OTP is a library that uses processes and layers to make it easy to build concurrent, self-healing software. Throughout the book, we'll deepen that understanding.

In this brief journey together, we will show you how to write effective Elixir by showing you how to use layers to hide complex features until you need to think about them. We'll extend our layers to take advantage of OTP, offering some intuition for how it works and some guidance for how to incorporate it into your layered designs.

If you find some tools to improve that skill, even if you don't use every technique in this book, you'll be much better positioned to create good Elixir code that takes full advantage of the wide variety of libraries and frameworks in the Elixir ecosystem.

The first question you may be asking is which layers you should build. In the sections that follow, we'll offer some guidance to help you choose.

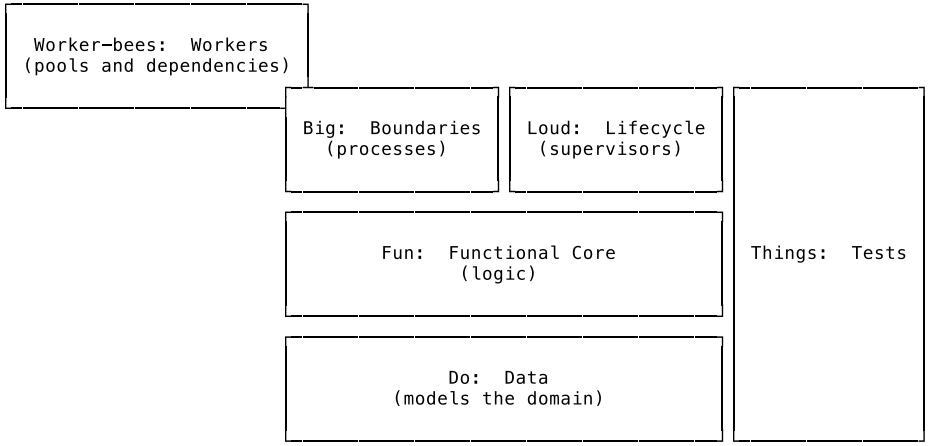
Choose Your Layers

The layers we will present to write a typical project are not set in stone. Instead, they are a rough scaffold, a framework for thinking about solutions to common design problems. We're not slaves to these systems but they help to free us from dealing with mechanical details so that we can focus on solving problems.

We recommend the software layers: data structures, a functional core, tests, boundaries, lifecycle, and workers. Not every project will have all of these layers, but some will. It's your job as the author of a codebase to decide which layers are worth the price and which ones to eliminate. It's a lot to remember, so use this sentence as a mnemonic:

> Do fun things with big, loud worker-bees.

The first letter of the essential words in the sentence match the first letters in our layers: data, functional core, tests, boundaries, lifecycles, workers. You can see how they all fit together in the following figure:



Do fun things with big, loud worker-bees.

In this chapter, we will explore each layer in detail. We'll call each unit of software you build that honors these concepts a *component*.

To help you understand what each of these layers do, we're going to build two components in this book. The first will be a trivial counter. We know you understand how counters work, but building this component will help you internalize the *design framework* we've established, and what each of the layers means.

The next component, a project called Mastery, will be much more complex, and will comprise the whole rest of the book. It will be a quiz, but not a typical one. This quiz will tailor itself as the user answers questions. Its purpose will be to help you learn to *use that design framework in context* to build a project with real complexity.

Let's get started with that first component, the counter. Rather, let's *not* get started. It always pays to think first.

Think Before You Start

This isn't as much a layer in our framework as a philosophy for coding. Most programmers don't think enough before opening the editor. It's healthy to start every problem with whatever tools help you think. It may just mean propping your feet up on a desk; it may be spending a little bit of time with

a whiteboard or even a pen and paper. Testing zealots like us believe bugs are less expensive to fix before they reach the customer. We'll take this idea further. Bugs are cheapest to catch before you write your first line of code.

At this stage, your first goal is to understand how to break down the major components in your system. Within the Elixir community, you won't find any single answer to how fine you should break down your components.

Here's the thing. If you think of OTP as a way to encapsulate data, or even objects, you're going to get it wrong. Elixir processes work best when they span a few modules that belong together. Breaking your processes up too finely invites integrity problems the same way that global variables do.

We believe that whenever possible, concepts that belong together should be packaged together as part of the same component. For example, we'd rather wrap a process around a *chess game* as a standalone component than have each *piece* in its own process, so we can enforce the integrity of the board at the *game level*.

Our counter is a standalone component that we'll use to count things in isolation. The data is an integer, does not need to persist through a failure or restart. The counter has a two function API to increment the counter and get the value. We only have a single component so we don't have to *divide responsibilities*.

We'll make the critical assumption that persisting state is unimportant and we don't have to worry about guaranteed delivery of messages, even across restarts, but our counter should track a value transiently, and that value should be available to other processes. Such state is *ephemeral*. Freedom from persistence allows us much more flexibility than we'd otherwise experience. Elixir is extremely good at managing ephemeral state such as counters and caches. In later chapters, you'll see a good way to add persistence to a component as we deal with the second component.

Create a Mix Project

With those details firmly in place, we can create our software. You might have noticed that until now, we've steadfastly avoided the word "application." There's a reason for that decision. The term is overloaded. To any given Elixir developer, an application might be the thing you:

- Build with OTP
- Create when you type `mix new`
- Create when you type `mix phx.new`
- Deploy

And each of these, in some context, is right. We're going to refrain from using "application" in the context of the thing we're creating with `mix new`. That thing is a *project*. Let's create one now.

Create a new project from your OS console. Type `mix new counter` and change into the `counter` directory. We are finally ready to build our first layer.

Begin with the Right Datatypes

The "data" layer has the simple data structures your functions will use. Just as an artist needs to learn to use the colors on their palette, Elixir developers need to learn the best ways to mix the data structures. Every programmer making a transition to functional programming needs to understand its impact on data design.

In this book, we won't tell you what maps or lists are, but we will provide an overview of what kinds of datatypes to choose for selected tasks and how you can weave them together into a good functional data strategy. We'll give you some dos and don'ts for the most common datatypes, and provide you some tips for choosing good ways to express the concepts in your program as data.

Our counter's datatype couldn't be simpler. It's an integer. Normally, you'll spend much more time thinking about your data than we do here. You'll likely begin to code up the major entities in your system. We don't need to do that for our counter because Elixir already has the integer, and it already supports the kinds of things we'll do to it.

As this book grows, we'll spend a good amount of time working through data structures. Our focus will be primarily in three areas:

- We'll look at what's idiomatic and efficient in Elixir.
- We'll review how our structures will influence the designs of our functions.
- We'll consider some of the trade-offs around cohesion, meaning how closely we group related bits of data.

When the data structure is right, the functions holding the algorithms that do things can seem to write themselves. Get them wrong and it doesn't really matter how good a programmer you are; your functions will feel clumsy and awkward.

Since we don't have any custom data structures, we can move on. Let's write some functions.

Build Your Functional Core

Now we'll finally start coding. Our functional core is what some programmers call the business logic. This inner layer does not care about any of the machinery related to processes; it does not try to preserve state; and it has no side effects (or, at least, the bare minimum that we must deal with). It is made up of functions.

Our goal is to deal with complexity in isolation. Make no mistake, processes and side effects add complexity. Building our core allows us to isolate the inherent complexity of our *domain* from the complexity of the *machinery* we need to manage processes, handle side effects, and the like.

In a chess game, this logic would have functions that take a board, move an individual piece, and return an updated board. It may also have a function to take a board with all of its pieces and calculate the relative strength of a position. In a calculator, the core would handle all of the numeric operators for the calculator.

Let's look at a specific example, our counter. Our business logic will count numbers. This code should be as side effect free as we can make it. It should observe two rules:

- It must not have side effects, meaning it should not alter the state of its environment in any way.
- A function invoked with the same inputs will always return the same outputs.

Our counter's business logic increments a value. Let's write that inner functional core now. Crack open lib/counter/core.ex and make it look like this:

GettingStarted/counter/lib/counter/core.ex

```
defmodule Counter.Core do
  def inc(value) do
    value + 1
  end
end
```

Though you can't yet behold the power of the fully operational counter, the business logic makes it easy to track exactly what is happening. Our public API has two functions: one to advance the counter and one to return state. The process we'll use to manage state doesn't belong here so we need only the inc function. Let's take it for a quick spin. Open it with iex -S mix, like this:

```
iex(1)> Counter.Core.inc(1)
2
```

Documentation and Typespecs

Before we dive into code, let's say a brief word about documentation. We'll mainly strip out the module docs and doc tests when we initially work on a project because we want to keep a tight feedback loop. A book is a poor place for comments and documentation fixtures in code because prose serves that role. In practice, when code reaches a fairly mature point, we'll add typespecs and module docs, and possibly even doc tests if they make sense. We also made the tough decision to remove typespecs because books are about trade-offs between space and concept. We believe the story arc flows better without them.

All of this is to say documentation and typespecs are important, but do what works for you. If you want to read more, check out *Adopting Elixir [Tat18]*.

That's all our functional core needs, just the functions that manipulate our data structure. If you want to see this code in the context of a program, spin up the following program:

```
defmodule Clock do
  def start(f) do
    run(f, 0)
  end

  def run(your_hearts_desire, count) do
    your_hearts_desire.(count)
    new_count = Counter.Core.inc(count)
    :timer.sleep(1000)
    run(your_hearts_desire, new_count)
  end
end
```

If you want to run this much, open up a new IEx shell because we'll have to kill the following one after running the timer since it loops forever. Then pick what you want to do every cycle by passing whichever function your heart desires into `run`, like this:

```
iex> Clock.start(fn(tick) -> IO.puts "The clock is ticking with #{tick}" end)
The clock is ticking with 1
The clock is ticking with 2
The clock is ticking with 3
...
```

And you'll have to kill that session with hot fire because it loops forever. Still, you can see the way we build our inner layer into a functional core.

We've addressed the data and functional core in "Do fun things"; we will come back to tests. For now, we understand that our counter must be more than a simple library. Counters exist to count and that means saving state. It's

time to address the process machinery, the “big, loud worker-bees” part of our mnemonic. We’ll start with a boundary layer.