Extracted from:

Programming Phoenix ≥ 1.4

Productive |> Reliable |> Fast

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Programming Phoenix ≥ 1.4

Productive |> Reliable |> Fast

Your Elixir Source

Chris McCord,
Bruce Tate,
and José Valim

edited by Jacquelyn Carter

Programming Phoenix ≥ 1.4

Productive |> Reliable |> Fast

Chris McCord
Bruce Tate
José Valim

The Pragmatic Bookshelf
Raleigh, North Carolina

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Susan Conant
Development 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*.

ISBN-13: 978-1-68050-226-8
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—October 2019

CHAPTER 1

Introducing Phoenix

The web has gone real time. The days of clicking links to load full pages are over. Websites are as interactive as desktop applications these days, if not more so. Servers manipulate widgets on a page with small data exchanges. Pages ship form data up piece by piece as it becomes available instead of waiting for one massive update. Today's web developers need a framework designed from the ground up around a real-time architecture, and Phoenix is that framework.

Ironically, most of the individual pieces from Phoenix can also be found in other places. You'll find metaprogramming capabilities that remind you of Lisp and domain-specific languages (DSLs) that remind you at times of Ruby. Our method of composing services with a series of functional transformations is reminiscent of Clojure's Ring. We achieved high throughput and reliability by climbing onto the shoulders of Erlang. Similarly, some of the groundbreaking features like channels and reactive-friendly APIs combine the best features of some of the best JavaScript frameworks but Phoenix makes it work at scale. This precise cocktail of features, where each feature multiplies the impact of the next, can't be found elsewhere and that's what makes Phoenix stand out. *Phoenix just feels right.*

After using (and writing about) frameworks spanning a half dozen languages across a couple of decades, we think the precise bundle of goodness that we'll share is powerful enough for the most serious problems you throw at it, beautiful enough to be maintainable for years to come, and—most important—fun to code. Give us a couple of pages and you'll find that the framework represents a great philosophy, one that leverages the reliability and grace of Elixir. You'll have a front-row seat to understand how we made the decisions that define Phoenix and how best to use them to your advantage.

Simply put, Phoenix is about productive, concurrent, beautiful, interactive, and reliable applications. Let's break each of these claims down.

Productive

Phoenix makes programmers productive. Right out of the box, Phoenix gives you everything you'd expect from a web framework:

- A base architecture for your application
- A database access and management library for connecting to databases
- A routing layer for connecting web requests to your code
- A templating language and helpers for you to write HTML
- Flexible and performant JSON encoding and decoding for external APIs
- Internationalization strategies for taking your application to the world
- All the breadth and power behind Erlang and Elixir so you can grow

Like all web frameworks, Phoenix provides a good cross section of features as functions so users don't have to code their own. However, features are not enough.

Productivity vs. Maintainability

All framework designers must walk a tightrope. Frameworks must anticipate change by allowing customization, but presenting customization options introduces complexity. Each new feature simply makes the high wire act more treacherous. Let's call one side of the line *productivity* and the other *maintainability*.

When developers have to learn too much too soon, they must slow down to absorb information. One way to keep developers productive early on is hiding details. When a framework designer leans too far this way, developers must pay a price because at some point, the framework will hide information their users need to solve a critical problem. Unusual customizations lead to hours of tedious searching for some mystery incantation to make things work.

Use such a framework long enough and you'll inevitably make changes that cause your application to drift away from the designers' intentions, setting yourself up for an eternal upstream battle against the framework. Whether it's a conflicting upgrade or an optimization that isn't compatible with your change doesn't matter. The framework developer's desire for short-term productivity has cost users long-term maintainability. You can find plenty of stale issues inside the issue trackers for both private and commercial web frameworks, telling this tale with stark clarity.

Sometimes, understanding this limitation, framework designers lean too far in the opposite direction. Too many options in too many places can also have rippling consequences. Options presented in the wrong way force users to make early uninformed decisions. Crippling detail work slowly starves users of the time they need at the beginning of a project, when productivity is the most important.

Phoenix takes a slightly different approach as it walks this tightrope. Phoenix is an opinionated framework that favors convention over configuration. But rather than *hiding complexity*, it *layers complexity*, providing details piece by piece.

Phoenix lets users see exactly what's happening by providing an explicit list of every bit of code a specific route will invoke, one after another. Phoenix hides details in layers by breaking its functionality into small functions and modules and naming them well so they can tell the story. Every application using Phoenix has an endpoint that looks like this:

```
defmodule MyApp.Endpoint do
  use Phoenix.Endpoint, otp_app: :my_app

  plug Plug.Static, ...
  plug Plug.RequestId
  plug Plug.Telemetry, ...
  plug Plug.Parsers, ...
  plug Plug.MethodOverride
  plug Plug.Head
  plug Plug.Session, ...
  plug MyApp.Router
end
```

We are going to dive deep into the mechanics later in the book. For now, what matters is that we have an overview of what our web application provides at a high level.

Rather than forcing users to configure the server with thousands of tiny decisions, Phoenix provides a default outline. If all you want to do is peek under the hood, you can open up a file. You don't need to modify this base outline at all, but when it's time to make that obscure change, you can edit this outline to your heart's content.

So often, productivity means avoiding blocks, and that means developers must have adequate information. Couple the layered architecture with Elixir's fantastic tools for documentation and you have the tools to be quite productive. For example, you can learn more about any of the components above by simply typing h Plug.Session in your Elixir terminal, or by accessing the documentation online[1] or directly in your favorite editor.

At the end of the day, Phoenix wants to optimize both productivity and maintainability. After all, maintainability means productivity over time.

Functional Programming 101: Immutability

One of the secrets for Phoenix's long-term productivity comes from a trait shared across many functional programming languages: immutability.

Imagine the following code:

```
list = [1, 2, 3]
do_something(list)
list
```

In most programming languages, you cannot assert with a 100% guarantee the list will still be [1, 2, 3] after calling do_something. That's because do_something can change the list in place. In Elixir, that's simply not possible. Our data structures are immutable, so instead of changing the list, we can only build new lists. Therefore our code is written as a series of small functions that receive everything they have to work with as input and return everything they have changed.

This plays a very important role in code readability and maintainability. You will spend much less time and brain cycles trying to figure out what object changed what or what is the current state of a certain component.

While this is a small example, you will find working with Elixir and functional programming to be full of small changes and improvements that make your code easier to understand, both for your teammates and your future self.

Concurrent

Over the last decade, we have been hearing more and more about concurrency. If you have never used a language with first-class concurrency support before, you may be wondering what all the fuss is about. In this section, we will cover why concurrency matters in the context of web applications and how Phoenix developers leverage it to build fast, performant applications. First let's talk about the different types of concurrency.

1. https://hexdocs.pm/plug/Plug.Session.html

Types of Concurrency

For our purposes, let's think of concurrency as a web application's ability to process two or more web requests at the same time. The simplest way to handle multiple requests is by executing them one right after the other, but that strategy isn't very efficient. To process a request, most web apps need to perform I/O such as making database requests or communicating with an external API. While you're waiting for those external services to complete, you could start working on the next request. This is I/O concurrency. Most programming languages provide I/O concurrency out of the box or via libraries. Sometimes, however, the I/O concurrency abstraction ends up leaking to the developer, who must write code in a confusing way, with callbacks of some form.

Another type of concurrency is multi-core concurrency, which focuses on the CPU. If your machine has more than one core, one core processes one request while a second core processes another one. For the rest of this discussion, we will consider machines with four cores in our examples, which is commonplace, as even smart watches have multiple cores today.

There are two main ways to leverage multi-core concurrency:

- With an operating system process per core: If your machine has four cores, you will start four different instances of your web application.

- With user space routines: If your machine has four cores, you start a single instance of your web application that is capable of using all cores efficiently.

The downside of using operating system processes is that those four instances cannot share memory. This solution typically leads to higher resource usage and more complex solutions.

Thanks to the Erlang VM, Elixir provides I/O concurrency without callbacks, with user-space multi-core concurrency. In a nutshell, this means Elixir developers write code in the simplest and most straightforward fashion and the virtual machine takes care of using all of the resources, both CPU and I/O, for you. The result is a better application. Let's talk about why.

Simpler Solutions

One issue with concurrency via operating system processes is the poor resource usage. Each core needs a separate instance of your application. If you have 16 cores, you need 16 instances, each on its own memory space.

With user space concurrency, you always start a single instance of your application. As you receive new requests, they are naturally spread throughout all cores. Furthermore, they all share the same memory. This small drawback might seem a little vague, so let's make it more explicit by looking at one specific problem, a common dashboard.

Imagine each user in your application has a dashboard. The data in this dashboard takes around 200ms to load and it takes about 100kB in memory. Since we want to provide good experience to users, we decide to cache this data. Let's say your web application supports only operating system process concurrency. That means each application instance needs to keep its own cache. For ten thousand (10,000) active users, that's a 1GB data cache for all of the dashboards *per instance*. For 16 cores with 16 instances, that's 16GB of cache, and it's only for the dashboard data. Furthermore, since each instance has its own cache shared across all users, each cache will be less effective at startup time because cache hit rates will be lower, leading to poor startup times.

To save memory and improve the hit rates, you may decide to put the data in an external caching system, such as Redis or memcached. This external cache increases your complexity for both development and deployment concerns because you now have a new external dependency. Your application is much faster than it would be if you were simply querying the database, but every time users access the dashboard, your application still needs to go over the network, load the cache data, and deserialize it.

In Elixir, since we start a single web application instance across all cores, we have a single cache of 1GB, shared across all cores, regardless of whether the machine has 1, 4, or 16 cores. We don't need to add external dependencies and we can serve the dashboard as quickly as possible because we don't need to go over the network.

Does this mean Elixir eliminates the need for caching systems? Surely not. For example, if you have a high number of machines running in production, you may still want an external caching system as a fallback to the local one. We just don't need external cache systems nearly as often. Elixir developers typically get a lot of mileage from their servers, without a need to resort to external caching. For example, Bleacher Report was able to replace 150 instances running Ruby on Rails with 5 Phoenix instances, which has been proven to handle eight times their average load at a fraction of the cost.[2]

2. https://www.techworld.com/apps-wearables/how-elixir-helped-bleacher-report-handle-8x-more-traffic-3653957/

And while this is just one example, we have the option to make similar trade-offs at different times in our stacks. For simple asynchronous processing, you don't need a background job framework. For real-time messaging across nodes, you don't need an external queue system. We may still use those tools, but Elixir developers don't need to reach for them as often as other developers might. We can avoid or delay buying into complex solutions, spending more time on domain and business logic.

Performance for Developers

Developers are users too. Elixir's concurrency can have a dramatic impact on our experience as we write software. When we compile software, run tests, or even fetch dependencies, Elixir is using all cores in your machine, and these shorter cycles over the course of a day can stack up.

Here is a fun story. In its first versions, Elixir used to start as many tests concurrently as the number of cores in your machine. For instance, if your machine has four cores, it would run at most four tests at the same time. This is a great choice if your tests are mostly using the CPU.

However, for web applications, it is most likely that your tests are also waiting on I/O, due to the database or external systems. Based on this insight, the Elixir team bumped the default number of concurrent tests to double the number of cores. The result? Users reported their test suites became 20%-30% faster. Overall, it is not uncommon for us to hear about web applications running thousands of tests in under 10 seconds.

But Concurrency Is Hard

You may have heard that concurrency is hard and we don't dispute that. We *do* claim that traditional languages make concurrency considerably harder than it should be. Many of the issues with concurrency in traditional programming languages come from in-memory race conditions, caused by mutability.

Let's take an example. If you have two user space routines trying to remove an element from the same list, you can have a segmentation fault or similarly scary error, as those routines may change the same address in memory at the same time. This means developers need to track where all of the state is and how it is changing across multiple routines.

In functional programming languages, such as Elixir, the data is immutable. If you want to remove an element from a list, you don't change that list in memory. You create a new list instead. That means as a functional developer, you don't need to be concerned with bugs that are caused by concurrent access to memory. You'll deal only with concurrency issues that are natural to your domain.

For example, what is the issue with this code sample?

```
product = get_product_from_the_database(id)
product = set_product_pageviews(get_product_pageviews(product) + 1)
update_product_in_the_database(product)
```

Consider a product with 100 pageviews. Now imagine two requests are happening at the same time. Each request reads the product from the database, sees that the counter is 100, increments the counter to 101, and updates the product in the database. When both requests are done, the end result *could* be 101 in the database while we expected it to be 102. This is a race condition that will happen regardless of the programming language you are using. Different databases will have different solutions to the problem. The simplest one is to perform the increment atomically in the database.

Therefore, when talking about web applications, concurrency issues are natural. Using a language like Elixir and a framework such as Phoenix makes all of the difference in the world. When your chosen environment is equipped with excellent tools to reason about concurrency, you'll have all of the tools you need to grow as a developer and improve your reasoning about concurrency in the wild.

In Elixir, our user-space abstraction for concurrency is also called *processes*, but do not confuse them with operating system processes. Elixir processes are abstractions inside the Erlang VM that are very cheap and very lightweight. Here is how you can start 100,000 of them in a couple of seconds:

```
for i <- 1..100_000 do
  spawn(fn -> Process.sleep(:infinity) end)
end
```

From now on, when you read the word *process*, you should think about Elixir's lightweight processes rather than operating system processes. That's enough about concurrency for now but we will be sure to revisit this topic later.

Beautiful Code

Elixir is perhaps the first functional language to support Lisp-style macros with a more natural syntax. This feature, like a *template for code*, is not always the right tool for everyday users, but macros are invaluable for extending the Elixir language to add the common features all web servers need to support.

For example, web servers need to map routes onto functions that do the job:

```
pipeline :browser do
  plug :accepts, ["html"]
  plug :fetch_session
  plug :protect_from_forgery
end

pipeline :api do
  plug :accepts, ["json"]
end

scope "/", MyApp do
  pipe_through :browser

  get "/users", UserController, :index
  ...
end

scope "/api/", MyApp do
  pipe_through :api

  ...
end
```

You'll see this code a little later. You don't have to understand exactly what it does. For now, know that the first group of functions will run for all browser-based applications, and the second group of functions will run for all JSON-based applications. The third and fourth blocks define which URLs will go to which controller.

You've likely seen code like this before. Here's the point. You don't have to sacrifice beautiful code to use a functional language. Your code organization can be even better. In Phoenix, you won't have to read through inheritance chains to know how your code works. You'll just build a pipeline for each group of routes that work the same way.

You can find an embarrassing number of frameworks that break this kind of code down into something that is horribly inefficient. Consultancies have made millions on performance tuning by doing nothing more than tuning route tables. This Phoenix example reduces your router to pattern matching that's further optimized by the virtual machine, becoming extremely fast. We've built a layer that ties together Elixir's pattern matching with the macro syntax to provide an excellent routing layer, and one that fits the Phoenix framework well.

You'll find many more examples like this one, such as Ecto's elegant query syntax or how we express requests as a pipeline of functions that compose well and run quickly. In each case, you're left with code that's easier to read, write, and understand.

We're not here to tell you that macros are the solution to all problems, or that you should use a DSL when a function call should do. We'll use macros when they can dramatically simplify your daily tasks, making them easier to understand and produce. When we do build a DSL, you can bet that we've done our best to make it fast and intelligent.

Effortlessly Extensible Architecture

The Phoenix framework gives you the right set of abstractions for extension. Your applications will break down into individual functions. Rather than rely on other mechanisms like inheritance that hide intentions, you'll roll up your functions into *pipelines*, where each function feeds into the next. It's like building a shopping list for your requests.

In this book, you'll write your own authentication code, based on secure open standards. You'll see how easy it is to tune behavior to your needs and extend it when you need to.

The Phoenix abstractions, in their current incarnation, are new, but each has withstood the test of time. When it's time to extend Phoenix—whether you're plugging in your own session store or doing something as comprehensive as attaching third-party applications like a Twitter wrapper—you'll have the right abstractions available to ensure that the ideas can scale as well as they did when you wrote the first line of code.

Interactive

By this point, you may be noticing that each concept builds on the previous one. Elixir makes productive, explicit layers available to programmers who can use them to build concurrent applications. Phoenix introduces beautiful, concurrent abstractions for use in beautiful APIs.

For the first four years, the Phoenix team worked at building this infrastructure, and this past year we've seen the culmination of this investment in new, exciting APIs for building interactive applications. The best example is Phoenix LiveView, a library for building applications without custom JavaScript. Until the right infrastructure was in place, LiveView could be only a dream.

Building interactive applications does require APIs that shield many different concerns from an end user, but APIs are just the tip of the iceberg. Underneath that tip is a tremendous amount of infrastructure. Let's take a peak beneath the surface.

Scaling by Forgetting

Traditional web servers scale by treating each tiny user interaction as an identical stateless request. The application doesn't save state between requests at all. It simply looks up the user and simultaneously looks up the context of the conversation in a user session. Presto. All scalability problems go away because there's only one type of connection.

But there's a cost. The developer must keep track of the state for each request, and that burden can be particularly arduous for newer, more interactive applications with intimate, long-running rich interactions. As a developer, until now, you've been forced to make a choice between applications that intentionally forget important details to scale and applications that try to remember too much and break under load.

Processes and Channels

With Elixir, you can create hundreds of thousands of *lightweight processes* without breaking a sweat. Lightweight processes also mean lightweight connections, and that matters because *connections can be conversations*. Whether you're building a chat on a game channel or a map to the grocery store, you won't have to juggle the details by hand anymore. This application style is called *channels*, and Phoenix makes it easy. Here's what a typical channels feature might look like:

```
def handle_in("new_annotation", params, socket) do
  broadcast! socket, "new_annotation", %{
    user: %{username: "anon"},
    body: params["body"],
    at: params["at"]
  }

  {:reply, :ok, socket}
end
```

You don't have to understand the details. Just understand that when your application needs to connect your users and broadcast information in real time, your code can get much simpler and faster.

Even now, you'll see many different types of frameworks begin to support channel-style features, from Java to JavaScript and even Ruby. Here's the problem. None of them comes with the simple guarantees that Phoenix has: isolation and concurrency. Isolation guarantees that if a bug affects one channel, all other channels continue running. Breaking one feature won't bleed into other site functionality. Concurrency means one channel can never block another one, whether code is waiting on the database or crunching data. This key advantage means that the UI never becomes unresponsive because the user started a heavy action. Without those guarantees, the development bogs down into a quagmire of low-level concurrency details.

You may also be wondering whether keeping an open connection per user can scale. The Phoenix team decided to benchmark their channels abstraction and they were able to reach two million connections on a single node.[3] And while that proves Phoenix Channels scale vertically (i.e., on powerful machines), it also scales horizontally. If you need to run a cluster of Phoenix instances, Phoenix will broadcast messages across all nodes out of the box, without a need for external dependencies.

It's true, you can build these kinds of applications without Phoenix, but building them without the guarantees of isolation and concurrency is never pleasant. The results will almost universally be infected with reliability and scalability problems, and your users will never be as satisfied as you'd like to make them.

Presence and LiveView

As Phoenix grows and matures, the team continues to provide tools developers can use to build interactive applications. The first addition was support for tracking presence. Tracking which users are connected to a cluster of machines is a notoriously difficult problem. But in Phoenix, it takes as little as ten lines of code to track which users, fridges, cars, doors, or houses are connected to your application. In a world that is getting more and more connected, this feature is essential.

The best part about presence is that it doesn't require any external dependencies. Regardless of whether you are running two Phoenix nodes or twenty, those nodes will communicate with each other, making sure to track connections regardless of where they happen in the cluster. You get a fantastic feature set right out of the box.

The most recent interactive development tool is LiveView. LiveView allows developers to build rich, interactive real-time applications without writing custom JavaScript.[4] For the JavaScript developers out there, it can be summarized as "server-side React". Here is a simple counter built with LiveView:

3. https://phoenixframework.org/blog/the-road-to-2-million-websocket-connections
4. https://dockyard.com/blog/2018/12/12/phoenix-liveview-interactive-real-time-apps-no-need-to-write-javascript

```
defmodule DemoWeb.CounterLive do
  use Phoenix.LiveView

  def render(assigns) do
    ~L"""
    <span><%= @val %></span>
    <button phx-click="inc">+</button>
    """
  end

  def mount(_session, socket) do
    {:ok, assign(socket, val: 0)}
  end

  def handle_event("inc", _, socket) do
    {:noreply, update(socket, :val, &(&1 + 1))}
  end
end
```

When Phoenix renders the page the first time, it works just like any other static page. That means browsers get a fast first-page view and search engines have something to index. Once rendered, Phoenix connects to the LiveView on the server, using WebSockets and Channels. LiveView applications are breathtakingly simple:

- A function renders a web page.
- That function accepts state as an input and returns a web page as output.
- Events can change that state, bit by bit.

State is a simple data structure that can hold whatever you want it to. Events that change your state can come from a button or a form on a web page. Other events can come from your application, like a low-battery sensor elsewhere in your application.

The best part is that LiveView is smart enough to send only what changes, and only when it changes. And once again, all you need to make this work is Phoenix.

Combine LiveView with Phoenix's ability to broadcast changes and track users in a cluster and you have the most complete tooling for building rich and interactive applications out of the box.

Reliable

As Chris followed José into the Elixir community, he learned to appreciate the frameworks that Erlang programmers have used to make the most reliable applications in the world. Before Elixir, the language of linked and supervised processes wasn’t part of his vocabulary. After spending some time with Elixir, he found the missing pieces he’d been seeking.

You see, you might have interactive applications built from beautiful, concurrent, responsive code, but it doesn’t matter unless your code is reliable. Erlang applications have always been more reliable than others in the industry. The secret is the process linking structure and the process communication, which allow effective supervision. You can start concurrent tasks and services that are fully supervised. When one crashes, Elixir can restart it in the last known good state, along with any tainted related service. Erlang’s supervisors can have supervisors too, so your whole application will have a tree of supervisors.

The nice thing is that you won’t have to write that supervision code yourself. By default, Phoenix has set up most of the supervision structure for you. For example, if you want to talk to the database, you need to keep a pool of database connections, and Phoenix provides one out of the box. As you’ll see later on, we can monitor and introspect this pool. It’s straightforward to study bottlenecks and even emulate failures by crashing database connections on purpose, only to see supervisors establishing new connections in their place. As a programmer, these abstractions will give you the freedom of a carpenter building on a fresh clean slab, *but your foundation solves many of your hardest problems before you even start.* As an administrator, you’ll thank us every day of the week because of the support calls that don’t come in.

In the next chapter, you’ll dive right in. From the beginning, you’ll build a quick application, and we’ll walk you through each layer of Phoenix. The water is fine. Come on in!