Extracted from:

Programming Phoenix

Productive |> Reliable |> Fast

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Programming Phoenix

Productive |> Reliable |> Fast

TM

Chris McCord,
Bruce Tate,
and José Valim

edited by Jacquelyn Carter

Programming Phoenix

Productive |> Reliable |> Fast

Chris McCord
Bruce Tate
and José Valim

The Pragmatic Bookshelf
Dallas, Texas • Raleigh, North Carolina

The team that produced this book includes:

Jacquelyn Carter (editor)
Potomac Indexing, LLC (index)
Eileen Cohen (copyedit)
Gilson Graphics (layout)
Janet Furlow (producer)

Printed in the United States of America.
ISBN-13: 978-1-68050-145-2
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—April 2016

CHAPTER 1

Introducing Phoenix

In the first few paragraphs to open this book, you probably expect us to tell you that Phoenix is radically different—newer and better than anything that's come before. We know that Phoenix is a bold name for a bold framework, but look. By now, you've likely seen enough web frameworks to know most of our ideas aren't new. It's the combination of so many of the best ideas from so many other places that has so many people excited.

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 such throughput and reliability by climbing onto the shoulders of Erlang and even native Cowboy. Similarly, some of the groundbreaking features like channels and reactive-friendly APIs combine the best features of some of the best JavaScript frameworks you'll find elsewhere. The precise cocktail of features seems to fit, where each feature multiplies the impact of the next.

We spent time on the right base abstractions for simplicity, and later we noticed that things weren't just fast, but among the fastest in the industry. When we pushed on performance and concurrency, we got functions that composed better and were simpler to manage. When our focus was on the right abstractions, we got better community participation. We now find ourselves in a firestorm of improvement. *Phoenix just feels right.*

After using and writing about frameworks spanning a half a 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 fast, concurrent, beautiful, interactive, and reliable applications. Let's break each of these claims down.

Fast

Let's cut to the chase. Elixir is both fast and concurrent, as you'd expect from a language running on the Erlang virtual machine. If you're looking for raw speed, Phoenix is hard to beat. In July 2015, we (Chris McCord) compared Phoenix with Ruby on Rails. The firebird was nearly an order of magnitude faster than the locomotive, and it used just over one fourth of the processing power and just under one sixth of the total memory. Those numbers are staggering, but not many Rails users are after naked power.

Let's compare Phoenix with some other popular frameworks. Check out the measurements of some major web frameworks at the Phoenix/mroth showdown.[1] Those results are impressive, rivaling the best in the industry. Among these servers are some of the fastest available. As these numbers rolled in, the core team got increasingly excited. Little did we know that the story was only beginning.

We kept noticing that as you add cores, *the story gets even better*. Another run of this benchmark on the most powerful machines at Rackspace[2] tells an even more compelling story. Check the link for details, but you can see the bare bones here:

Framework	Throughput (req/s)	Latency (ms)	Consistency (σ ms)
Plug	198328.21	0.63	2.22
Phoenix 0.13.1	179685.94	0.61	1.04
Gin	176156.41	0.65	0.57
Play	171236.03	1.89	14.17
Phoenix 0.9.0	169030.24	0.59	0.30
Express Cluster	92064.94	1.24	1.07
Martini	32077.24	3.35	2.52
Sinatra	30561.95	3.50	2.53
Rails	11903.48	8.50	4.07

1. https://github.com/mroth/phoenix-showdown/blob/master/README.md
2. https://gist.github.com/omnibs/e5e72b31e6bd25caf39a

Throughput is the total number of transactions, latency is the total waiting time between transactions, and consistency is a statistical measurement of the consistency of the response. Phoenix is the fastest framework in the benchmark and among the most consistent. The slowest request won't be *that* much slower than the fastest. The reason is that Elixir's lightweight concurrency removes the need for stop-the-world garbage collectors. You can see results for Plug, the Elixir library that serves as a foundation for Phoenix, as well as results for two different versions of Phoenix. You can see that over time, Phoenix performance is getting *better*, and the performance is in the same ballpark with the lower-level Plug.

You'll see several reasons for this blindingly fast performance:

- Erlang has a great model for concurrency. Facebook bought WhatsApp for $21 billion. That application achieved two million concurrently running connections on a single node.
- The router compiles down to the cat-quick pattern matching. You won't have to spend days on performance tuning before you even leave the routing layer.
- Templates are precompiled. Phoenix doesn't need to copy strings for each rendered template. At the hardware level, you'll see caching come into play for these strings where it never did before.
- Functional languages do better on the web. Throughout this book, you'll learn why.

Performance with Phoenix isn't an afterthought. Nor will you have to trade beautiful, maintainable code to get it.

Concurrent

If you're using an object-oriented web framework, chances are that you're watching the evolution of multicore architectures anxiously. You probably already know that the existing imperative models won't scale to handle the types of concurrency we'll need to run on hardware with thousands of cores. The problem is that languages like Java and C# place the burden of managing concurrency squarely on the shoulders of the programmer. Languages like PHP and Ruby make threading difficult to the point where many developers try to support only one web connection per operating-system process, or some structure that is marginally better. In fact, many people that come to Phoenix find us precisely because concurrency is so easy.

Consider PhoenixDelayedJob or ElixirResque—complex packages that exist only to spin off reliable processes as a separate web task. You don't need one. Don't get us wrong. In Ruby, such packages are well conceived and a critical part of any well-crafted solution. In Elixir, those frameworks turn into primitives. The Elixir programming model makes reasoning about concurrent systems almost as easy as reasoning about single-threaded ones. When you have two database fetches, you won't have to artificially batch them together with a stored procedure or a complex query. You can let them work *at the same time:*

```
company_task  = Task.async(fn -> find_company(cid) end)
user_task     = Task.async(fn -> find_user(uid) end)
cart_task     = Task.async(fn -> find_cart(cart_id) end)

company = Task.await(company_task)
user = Task.await(user_task)
cart = Task.await(cart_task)

...
```

You don't have to wait for the combined time for three database requests. Your code will take as long as the single longest database request. You'll be able to use more of your database's available power, and other types of work—like web requests or long exports—will complete much more quickly.

In aggregate, your code will spend *less time waiting* and *more time working*.

Here's the kicker. This code is more reliable. Elixir is based on the libraries that form the backbone of the most reliable systems in the world. 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.

Reliability and performance don't need to be mutually exclusive.

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 dozens of skip_before_filter commands 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 controllers 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.

Simple Abstractions

One continuous problem with web frameworks is that they tend to bloat over time, sometimes fatally. If the underlying abstractions for extending the framework are wrong, each new feature will increase complexity until the framework collapses under its own weight. Sometimes, the problem is that the web framework *doesn't include enough*, and the abstractions for extending the framework aren't right.

This problem is particularly acute with object-oriented languages. Inheritance is simply not a rich enough abstraction to represent the entire ecosystem of a web platform. Inheritance works best when a single feature extends a framework across a single dimension. Unfortunately, many ambitious features span several different dimensions.

Think about authentication, a feature that will impact every layer in your system. Database models must be aware, because authentication schemes require a set of fields to be present, and passwords must be hashed before being saved. Controllers are not immune, because signed-in users must be treated differently from those who are not. View layers, too, must be aware, because the contents of a user interface can change based on whether a user is signed in. Each of those areas must then be customized by the programmer.

Effortlessly Extensible

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 explicit lists called *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

Chris started the Phoenix project after working to build real-time events into his Ruby on Rails applications. As he started to implement the solution, he had a threading API called Event Machine and noticed that his threads would occasionally die. He then found himself implementing code to detect dead threads.

Over time, the whole architecture began to frustrate him. He was convinced that he could make it *work*, but he didn't think he could ever make it *beautiful* or *reliable*.

If you're building interactive applications on a traditional web stack, you're probably working harder than you need to. There's a reason for that. In the years before web programming was popular, client-server applications were simple. A client process or two communicated to its own process on the server. Programmers had a difficult time making applications scale. Each application connection required its own resources: an operating-system process, a network connection, a database connection, and its own memory. Hardware didn't have enough resources to do that work efficiently, and languages couldn't support many processes, so scalability was limited.

Scaling by Forgetting

Traditional web servers solve the scalability problem by treating each tiny piece of a user interaction as an identical request. The application doesn't save state 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 have both performance and productivity, because it supports a feature called *lightweight processes*. In this book, when you read the word *process*, you should think about Elixir lightweight processes rather than operating-system processes. You can create hundreds of thousands of 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 doesn't need to juggle the past details of a conversation, 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.

Building applications without these guarantees is usually possible but 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.

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 monitored 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 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. Erlang's supervisors can have supervisors too, so that your whole application will have a tree of supervisors.

Here's the kicker. 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 a 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.

Now that we've shown you some of the advantages of Phoenix, let's decide whether this book is right for you.

Is This Book for You?

If you've followed Phoenix for any period of time, you already know that this book is the definitive resource for Phoenix programming. If you're using Phoenix or are seriously considering doing professional Elixir development, you're going to want this book. It's packed with insights from the team that created it. Find just one tip in these pages, and the book will pay for itself many times over. This section seeks to answer a different question, though. Beyond folks who've already decided to make an investment in Phoenix, who should buy this book?

Programmers Embracing the Functional Paradigm

Every twenty years or so, new programming paradigms emerge. The industry is currently in the midst of a shift from object-oriented programming to functional programming. If you've noticed this trend, you know that a half dozen or so functional languages are competing for mindshare. The best way to understand a programming language is to go beyond basic online tutorials to see how to approach nontrivial programs.

With *Programming Phoenix*, we don't shy away from difficult problems such as customizing authentication, designing for scale, or interactive web pages. As you explore the language, you'll learn how the pieces fit together to solve difficult problems and how functional programming helps us do it elegantly. When you're done, you might not choose Phoenix, but you'll at least understand the critical pieces that make it popular and if those pieces are likely to work for you.

Rails Developers Seeking Solutions

If you follow the Rails community closely, you know that it has experienced some attrition. Bear in mind that this team believes that Ruby on Rails was great for our industry. Rails still solves some problems well, and for those problems it can be a productive solution. The problem for Rails developers is that the scope of problems it's best able to solve is rapidly narrowing.

In fact, the early growth of Elixir is partially fueled by Rails developers like you. The similar syntax provided an attractive base for learning the language, but the radically improved programming paradigms, introspectable runtime, and concurrency models all provide the solid foundation that those who push Rails the hardest find lacking.

Phoenix measures response times in microseconds, and it has been shown to handle millions of concurrent WebSocket connections on a single machine without sacrificing the productivity we've come to appreciate.

If you're pushing Rails to be more scalable or more interactive, you're not alone. You're going to find Phoenix powerful and interesting.

Dynamic Programmers Looking for a Mature Environment

Like the authors of this book, you may be a fan of dynamic languages like Ruby, Python, and JavaScript. You may have used them in production or even contributed to those ecosystems. Many developers like us are looking for similar flexibility but with a more robust runtime experience. We may love the programming experience in those languages, but we often find ourselves worn out by the many compromises we have to make for performance, concurrency, and maintainability. Phoenix resonates with us because many of the creators of this ecosystem built it to solve these problems.

Elixir is a modern dynamic language built on the three-decades-old, battle-tested Erlang runtime. Elixir macros bring a lot of the flexibility that Ruby, Python, and JavaScript developers came to love, but those dynamic features are quarantined to compile time. With Elixir, during runtime, you have a consistent system with great type support that's generally unseen in other dynamic languages.

Mix these features with the concurrency power, and you'll see why Phoenix provides such excellent performance for everything on the web, and beyond.

Java Developers Seeking More

When Java emerged twenty years ago, it had everything a frustrated C++ community was missing. It was object-oriented, secure, ready for the Internet, and simple, especially when compared to the C++ alternatives at the time. As the Java community flourished and consolidated, the tools and support came. Just about everyone supported Java, and that ubiquity led to a language dominance that we'd never seen before.

As Java has aged, it's lost some of that luster. As the committees that shaped Java compromised, Java lost some of the edge and leadership that the small leadership team provided in early versions. Backward compatibility means that the language evolves slowly as new solutions emerge. All of that early ubiquity has led to a fragmented and bloated ecosystem that moves too slowly and takes years to master, but delivers a fraction of the punch of emerging languages. The Java concurrency story places too much of a burden on the developer, leaving libraries that may or may not be safe for production systems.

New languages are emerging on the JVM, and some of those are rich in terms of features and programming models. This team respects those languages tremendously, but we didn't find the same connection there that we found elsewhere. We also had a hard time separating the good from the bad in the Java ecosystem.

If you're a Java developer looking for where to go next, or a JVM-language developer looking for a better concurrency story, Phoenix would mean leaving the JVM behind. Maybe that's a good thing. You'll find a unified, integrated story in Phoenix with sound abstractions on top. You'll see a syntax that provides Clojure-style metaprogramming on syntax that we think is richer and cleaner than Scala's. You'll find an existing ecosystem from the Erlang community that has a wide range of preexisting libraries, but ones built from the ground up to support not only concurrency, but also distributed software.

Erlang Developers Doing Integrated Web Development

Curiously, we're not seeing a heavy proliferation of Erlang developers in the Elixir community so far. We expect that to change. The toolchain for Phoenix is spectacular, and many of the tools that exist for Erlang can work in this ecosystem as well. If you're an Erlang developer, you may want to take advantage of Mix's excellent scripting for the development, build, and testing workflow. You may like the package management in Hex, or the neat composition of concerns in the Plug library. You may want to use macros to extend the language for your business, or test with greater leverage. You'll have new programming features like protocols or structs.

If you do decide to embrace Elixir, that doesn't mean you need to leave Erlang behind. You'll still be able to use the Erlang libraries you enjoy today, including the Erlang process model and full OTP integration. You'll be able to access your OTP `GenServers` directly from the Elixir environment, and directly call libraries without the need for extra complex syntax. If these terms aren't familiar to you, don't worry. We'll explore each of them over the course of the book.

Heat Seekers

If you need raw power supported by a rich language, we have a solution and the numbers to back it up. You'll have to work for it, but you'll get much better speed and reliability when you're done. We've run a single chat room on one box supporting two million users. That means that each new message had to go out two million times. We've run benchmarks among the best in the industry, and our numbers seem to be improving as more cores are added. If you need speed, we have the tonic for what ails you.

Others

Certainly, this book isn't for everyone. We do think that if you're in one of these groups, you'll find something you like here. We're equally confident that folks that we haven't described will pick up this book and find something valuable. If you're one of those types, let us know your story.

Online Resources

The apps and examples shown in this book can be found at the Pragmatic Programmers website for this book.[3] You'll also find the community forum and the errata-submission form, where you can report problems with the text or make suggestions for future versions.

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!

3. http://pragprog.com/book/phoenix/programming-phoenix