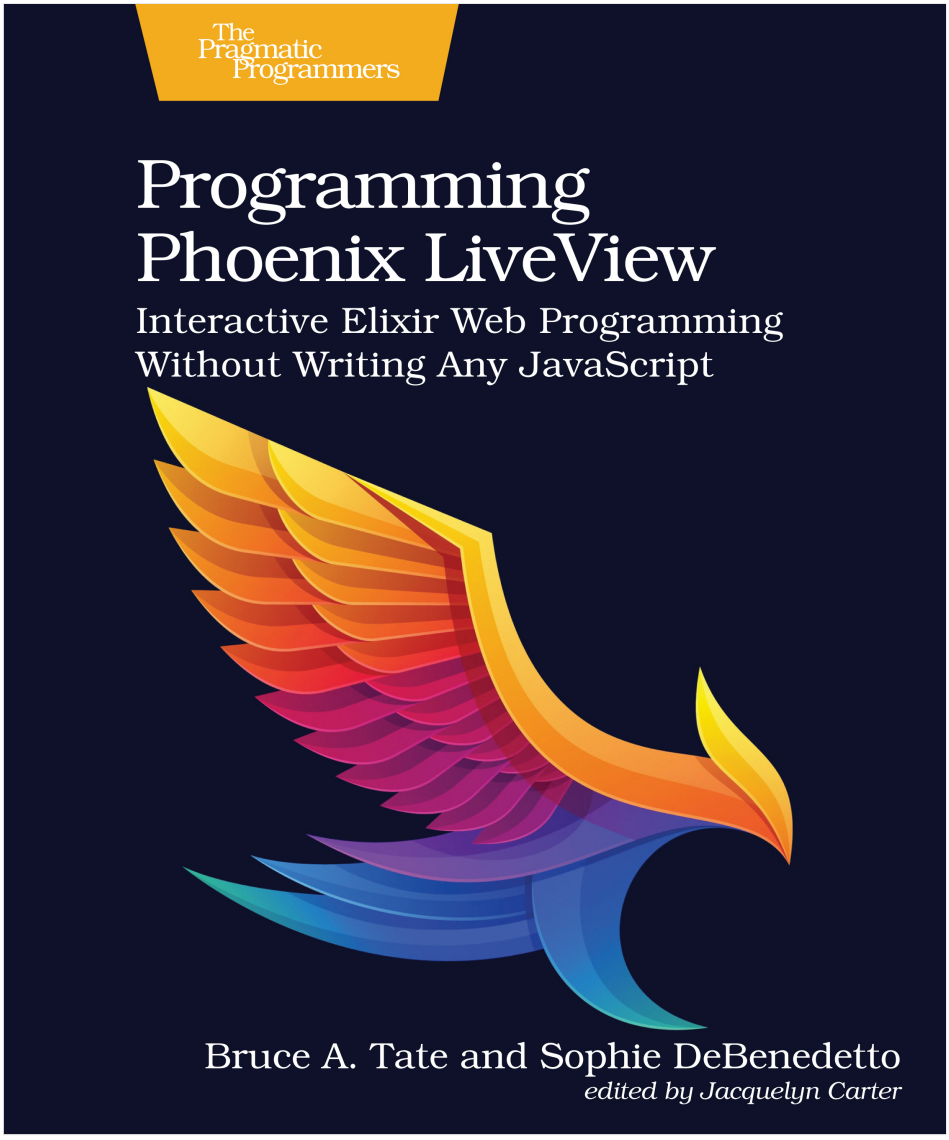

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

Introduction

If you haven't been following closely, it might seem like LiveView came suddenly, like a new seedling that breaks through the soil surface overnight. That narrative lacks a few important details, like all of the slow germination and growth that happens out of sight.

Chris McCord, the creator of Phoenix, worked on Ruby on Rails before coming over to the Elixir community. More and more often, his consultancy was asked to use Ruby on Rails to build dynamic single-page apps (SPAs). He tried to build a server-side framework on top of the Ruby on Rails infrastructure, much like LiveView, that would allow him to meet these demands for interactivity. But Chris recognized that the Ruby infrastructure was not robust enough to support his idea. He needed better reliability, higher throughput, and more even performance. He shopped around for a more appropriate language and infrastructure, and found Elixir.

When Chris moved from Ruby to Elixir, he first learned the metaprogramming techniques[1] he'd need to implement his vision. Then, he began building the Phoenix web development framework to support the infrastructure he'd need to make this vision a reality.

At that time, José Valim began helping Chris write idiomatic Elixir abstractions relying on OTP. OTP libraries have powered many of the world's phone switches, offering stunning uptime statistics and near realtime performance, so it played a critical role in Phoenix. Chris introduced a programming model to Phoenix called *channels*. This service uses HTTP WebSockets[2] and OTP to simplify interactions in Phoenix. As the Phoenix team fleshed out the programming model, they saw stunning performance and reliability numbers. Because of OTP, Phoenix would support *the concurrency, reliability, and performance that interactive applications demand.*

1. https://pragprog.com/titles/cmelixir/metaprogramming-elixir/
2. https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API

In functional programming, Chris found cleaner ways to tie his ideas together than object orientation offered. He learned to compose functions with Elixir pipelines and the plugs. His work with OTP taught him to think in the same composable steps we'll show you as this book unfolds. His work with metaprogramming and macros prepared him to build smooth features beyond what basic Elixir provided. As a result, in Phoenix LiveView, users would find a *pleasant, productive programming experience*.

As the web programming field around him grew, frameworks like React and languages like Elm provided a new way to think about user interface development in layers. Chris took note. Some frameworks like Morphdom popped up to allow seamless replacement of page elements in a customizable way. The Phoenix team was able to build JavaScript features into LiveView that automate the process of changing a user interface on a socket connection. In LiveView, programmers would find a *beautiful programming model based on tested concepts*, and one that provided JavaScript infrastructure so *developers didn't need to write their own JavaScript.*

In a nutshell, that's LiveView. We'll have plenty of time to go into more detail, but now, let's talk about you.

Is This Book for You?

This book is for advanced beginners and intermediate programmers who want to build web applications using Phoenix LiveView. In it, you'll learn the basic abstractions that make LiveView work, and you'll explore techniques that help you organize your code into layers that make sense. We will try not to bore you with a tedious feature-by-feature march. Instead, we'll help you grasp LiveView by building a nontrivial application together.

We think this book is ideal for these readers:

You Want to Build Something with LiveView

In this book, you'll learn the way the experts do. You'll write programs that communicate the most important LiveView concepts. You'll take four passes through the content.

- You'll start with a trivial example.
- Then, you'll generate some working code, and walk through it step by step with the authors.
- After that, you'll extend those programs while tacking on your own code.
- Finally, you'll code some complex programs from scratch.

When you're done, you'll know the base abstractions of Phoenix LiveView, you'll know how to build on them, and you'll be able to write code from scratch because you'll know what code goes where.

You Are Having a Hard Time Getting Started

Phoenix LiveView is a brilliant programming model, but it's not always an easy model to grasp. Sometimes, you need a guide. In this book, we break down the basics in small examples like this one:

```
mount() |> render() |> handle_event()
```

Of course, LiveView is a bit more complicated, but this short example communicates the overarching organization underneath every single LiveView program. We'll show you how this example makes it easier to understand the LiveView layer, and we'll show you tools you can use to understand where to place the other bits of your program.

When you're done, you'll know how LiveView works. More importantly, you'll know how it works with the other layers in your Phoenix application.

You Want to Know Where Your Layers Go

LiveView is just one part of a giant ecosystem. Along the way, you will encounter concepts such as Ecto, OTP, Phoenix, templates, and components. The hard part about coding LiveView isn't building code that works the first time.

If you want code that lasts, you'll need to break your software into layers, the way the experts do. We'll show you how Phoenix developers organize a core layer for predictable concepts, and to manage uncertainty in a boundary layer. Then, you'll explore how to apply some of the same concepts in the user interface. We'll show you how to break off major components, and also how to write functions that will be primed for reuse.

If you are seeking organizational guidance, you'll be able to fit the concepts in this book right into your mental bookshelf. You won't just know what to do; you'll know why to do it that way.

You Like to Play

If you want to program just for the joy of it, LiveView is going to be great for you. The programming model keeps your brain firmly on the server, and lets you explore one concept at a time. Layering on graphics makes this kind of exploratory programming almost magical. If this paragraph describes you,

LiveView will give your mind room to roam, and the productivity to let your fingers keep up.

This Book Might Not Be For You

While most LiveView developers will have something to learn from us, two groups might want to consider their purchase carefully. Advanced Elixir developers might find this book too basic, and early stage beginners might find it too advanced. Let us explain.

If you've never seen Elixir before, you'll probably want to use other resources to learn Elixir, and come back later. If you don't yet know Elixir, we'll provide you with a few resources you might try before coming back to this book.

Alternative Resources

If you are new to functional programming and want to learn it with a book, try *Learn Functional Programming with Elixir. [Alm18]* For a book for programmers that ramps up more quickly, try *Programming Elixir. [Tho18]* For a multimedia approach, check out Groxio.[3]

Similarly, this book might move a bit slowly for if you are an advanced programmer, so you have a difficult decision to make since there aren't many LiveView books out yet. We won't be offended if you look elsewhere. If you are building APIs in Phoenix, but not single-page apps, this book is not for you, though you will probably enjoy what *Programming Phoenix [TV19]* has to say. If you want an advanced book about organizing Elixir software, check out *Designing Elixir Systems with OTP. [IT19]*

If you're willing to accept a book that's paced a bit slowly for advanced developers, we're confident that you will find something you can use.

About this Book

Programmers learn by writing code, and that's exactly how this book will work. We'll work on a project together as if we're a fictional game company. You'll write lots of code, starting with small tweaks of generated code and building up to major enhancements that extract common features with components.

As you build the application, you'll encounter more complexity. A distributed dashboard will show a real time view of other users and processes. You'll even

3. https://grox.io/language/elixir/course

build a game from scratch because that's the best way to learn how to layer the most sophisticated LiveView applications.

Let's take a more detailed look at the plan.

Part I: Code Generation

We'll use two different code generators to build the foundational features of the Pento web app—a product database with an authenticated LiveView admin interface.

We won't treat our generated code as black boxes. Instead, we'll trace through the generated code, taking the opportunity to learn LiveView and Phoenix design and best practices from some of the best Elixir programmers in the business. We'll study how the individual pieces of generated code fit together and discuss the philosophy of each layer. We'll show you when to reach for generators and what you'll gain from using them.

Chapter 2, Phoenix and Authentication, on page ?

The phx.gen.auth authentication layer generator is a collaboration between the DashBit company and the Phoenix team. This code doesn't use LiveView, but we'll need this generator to authenticate users for our applications. You'll generate and study this layer to learn how Phoenix requests work. Then, you'll use the generated code to authenticate a live view.

Chapter 3, Generators: Contexts and Schemas, on page ?

The phx.gen.live generator creates live views with all of the code that backs them. We'll use this generator to generate the product CRUD feature-set. Since the code created by the phx.gen.live generator contains a complete out-of-the-box set of live views backed by a database, we'll spend two chapters discussing it. This chapter will focus on the two backend layers—the *core* and the *boundary*. The boundary layer, also referred to as the *context*, represents code that has uncertainty, such as database interfaces that can potentially fail. The context layer will allow our admin users to manage products through an API. The core layer contains code that is certain and behaves predictably, for example, code that maps database records and constructs queries.

Chapter 4, Generators: Live Views and Templates, on page ?

The phx.gen.live generator also generates a set of web modules, templates, and helpers that use the database-backed core and boundary layers detailed in the previous chapter. This chapter will cover the web side of this generator, including the LiveView, templates, and all of the supporting user interface code. Along the way, we'll take a detailed look at the gener-

ated LiveView code and trace how the pieces work together. This walk-through will give you a firm understanding of LiveView basics.

With the LiveView basics under your belt, you'll know how to generate code to do common tasks, and extend your code to work with forms and validations. You'll be ready to build your own custom live views using components.

Part II: LiveView Composition

LiveView lets you compose complex change management behavior with layers. First, we'll look at how LiveView manages change with the help of changesets and you'll see how you can compose change management code in your live views. Then, we'll take a deep dive into LiveView components. Components are a mechanism for compartmentalizing live view behavior and state. A single live view can be comprised of a set of small components, each of which is responsible for managing a specific part of your SPA's state. In this part, you'll use components to build organized live views that handle sophisticated interactive features by breaking them down into smaller pieces. Let's talk about some of those features now.

With our authenticated product management interface up and running, our Pento admins will naturally want to know how those products are performing. So, we'll use LiveView, and LiveView components, to do a bit of market research.

We'll build a survey feature that collects demographic information and product ratings from our users. We'll use two LiveView component features to do this work.

Chapter 5, Forms and Changesets, on page ?

After we've generated a basic LiveView, we'll take a closer look at forms. Ecto, the database layer for Phoenix, provides an API, called changesets, for safely validating data. LiveView relies heavily on them to present forms with validations. In this chapter, we'll take a second pass through basic changesets and form tags for database-backed data. Then, we'll work with a couple of corner cases including changesets without databases and attachment uploads.

Chapter 6, Function Components, on page ?

We'll use stateless components to start carving up our work into reusable pieces. These components will work like partial views. We'll use them to build the first pieces of a reusable multi-stage poll for our users. In the first stage, the user will answer some demographic questions. In the next

stage, the user will rate several products. Along the way, you'll encounter the techniques that let LiveView present state across multiple stages.

Chapter 7, Live Components, on page ?

As our components get more sophisticated, we'll need to increase their capability. We'll need them to capture events that change the state of our views. We'll use stateful components to let our users interact with pieces of our survey by tying common state to events.

By this point, you'll know when and how to reach for components to keep your live views manageable and organized.

Part III: Extend LiveView

In the next few chapters, you'll see how you can extend the behavior of your custom live view to support real-time interactions. We'll use communication between a parent live view and child components, and between a live view and other areas of your Phoenix app, to get the behavior we want. You'll learn how to use these communication mechanisms to support distributed SPAs with even more advanced interactivity.

Having built the user surveys, we'll need a place to evaluate their results. We'll build a modular admin dashboard that breaks out survey results by demographic and product rating. Our dashboard will be highly interactive and responsive to both user-triggered events and events that occur elsewhere in our application.

We'll approach this functionality in three chapters.

Chapter 8, Build an Interactive Dashboard, on page ?

Users will be able to filter results charts by demographic info and rating. We'll leverage the functions and patterns that LiveView provides for the event management lifecycle and you'll see how components communicate with the live view to which they belong.

Chapter 9, Build a Distributed Dashboard, on page ?

Our survey results dashboard won't just update in real-time to reflect state changes brought about by user interaction on the page. It will also reflect the state of the entire application by updating in real-time to include any new user survey results, as they are submitted by our users. This distributed real-time behavior will be supported by Phoenix PubSub.

Chapter 10, Test Your Live Views, on page ?

Once our dashboard is up and running, we'll take a step back and write some tests for the features we've built. We'll examine the testing tools

that LiveView provides and you'll learn LiveView testing best practices to ensure that your live views are robustly tested as they grow in complexity.

When we're done, you'll understand how to use components to compose even complex single-page behaviors into one elegant and easy-to-maintain live view. You'll also know how to track and display system-wide information in a live view. You'll have everything you need to build and maintain highly-interactive, real-time, distributed single-page applications with LiveView.

With all of that under our belts, we'll prototype a game.

Part IV: Graphics and Custom Code Organization

We know games aren't the best use case for LiveView. It's usually better to use a client-side technology to solve a pure client-side problem, but bear with us. We strongly believe that games are great teaching tools for the layering of software. They have well-understood requirements, and they have complex flows that often mirror problems we find in the real world. Building out our game will give you an opportunity to put together everything you've learned, from the basics to the most advanced techniques for building live views.

In this set of chapters, we'll prototype a proof-of-concept for a game. A quick proof of concept is firmly in LiveView's wheelhouse, and it can save tons of time and effort over writing games in less productive environments.

Our game will consist of simple puzzles of five-unit shapes called pentominoes. Here are the concepts we'll focus on. By this point, none of these concepts will be new to you, but putting them into practice here will allow you to master them.

Chapter 11, Build the Game Core, on page ?

We'll build our game in layers beginning with a layer of functions called the *core*. We'll review the reducer method and drive home why it's the right way to model functional software within functional cores. We'll use this technique to build the basic shapes and functions that will make up our game.

Chapter 12, Render Graphics With SVG, on page ?

We integrate the details of our game into a basic presentation layer. LiveView is great at working with text, and SVG is a text-based graphics representation. We'll use SVG to represent the game board and each pentomino within that board.

Chapter 13, Establish Boundaries and APIs, on page ?

As our software grows, we'll need to be able to handle uncertainty. Our code will do so in a *boundary* layer. Our boundary will implement the rules that effectively validate movements, limiting how far the pentominoes can move on the page. We'll also integrate the boundary layer into our live view.

These low-level details will perfectly illustrate how the different parts of Elixir work together in a LiveView application. When you're through with this part, you'll have practiced the techniques you'll need to build and organize your own complex LiveView applications from the ground up.

Online Resources

The apps and examples shown in this book can be found at the Pragmatic Programmers website for this book.[4] You'll also find the errata-submission form, where you can report problems with the text or make suggestions for future versions. If you want to explore more from these authors, you can read more of Sophie's fine work at Elixir School.[5] If you want to expand on this content with videos and projects to further your understanding, check out Groxio's LiveView course,[6] with a mixture of free and paid content.

When you're ready, turn the page and we'll get started. Let's build something together!

4. http://pragprog.com/titles/liveview/
5. https://elixirschool.com/blog/phoenix-live-view/
6. https://grox.io/language/liveview/course