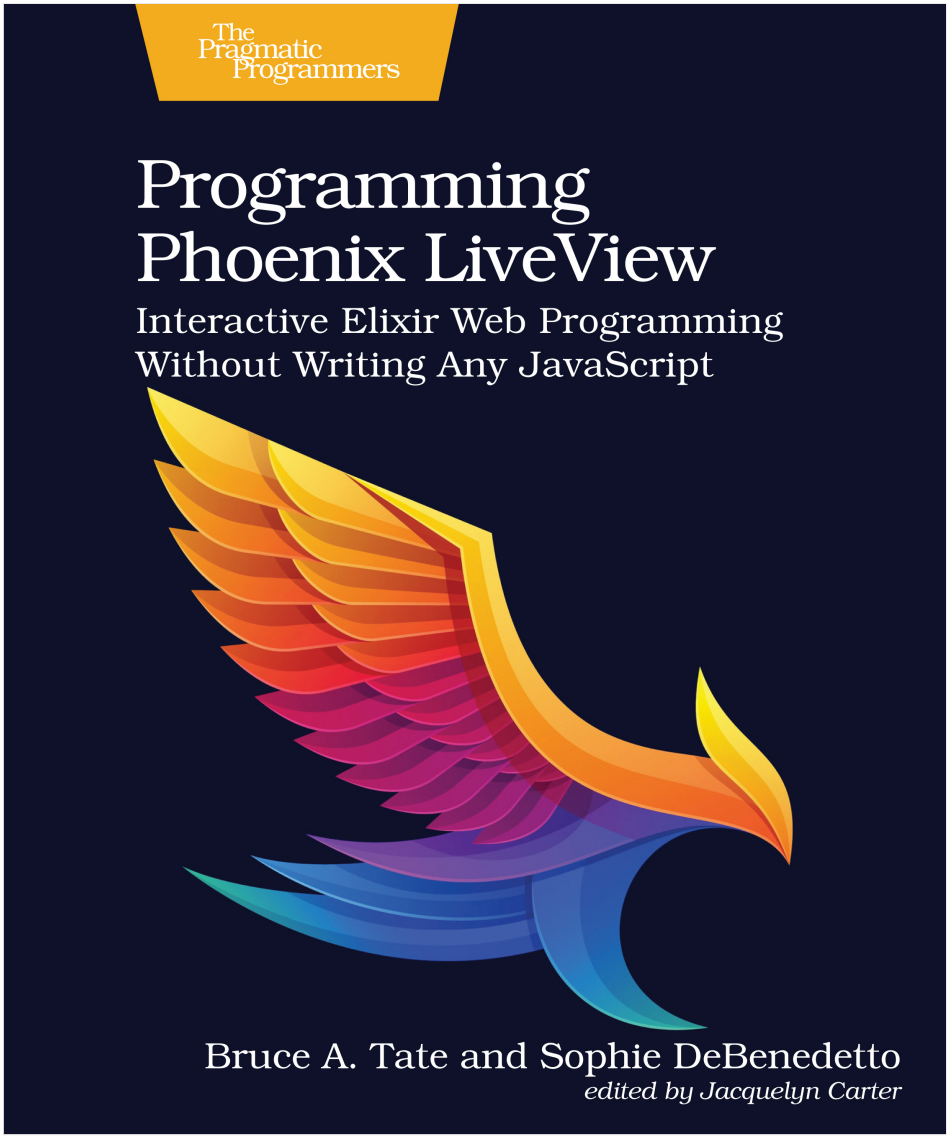


This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

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

Copyright © The Pragmatic Programmers, LLC.

CHAPTER 1

Get To Know LiveView

The nature of web development is changing. For many years, programmers needed to build web programs as many tiny independent requests with responses. Most teams had some developers dedicated to writing programs to serve requests, and a small number of team members dedicated to design. Life was simple. Applications were easy to build, even though the user experience was sadly lacking. Frankly, for most tasks, simple request-response apps were *good enough*.

Time passed until yesterday's *good enough* didn't quite cut it, and users demanded more. In order to meet these demands, web development slowly evolved into a mix of tools and frameworks split across the client and server. Take any of these examples:

- Instead of loading content page by page, modern Twitter and Facebook feeds load more content as a user scrolls down.
- Rather than having an inbox for an email client with a "refresh" button, users want a page that adds new emails to their inbox in real-time.
- Search boxes auto complete based on data in the database.

These kinds of web projects are sometimes called *single-page apps* (SPAs), though in truth, these kinds of applications often span multiple pages. Many different technologies have emerged to ease the development of SPAs. JavaScript frameworks like React make it easier to change web pages based on changing data. Web frameworks like Ruby's Action Cable and our own Phoenix Channels allow the web server to keep a running conversation between the client and the server. Despite these improvements, such tools have a problem. They force us into the wrong mindset—they don't allow us to think of SPAs as distributed systems.

• Click HERE to purchase this book now. discuss

Single-Page Apps are Distributed Systems

In case that jarring, bolded title wasn't enough to grab your attention, let the reality sink in. *A SPA is a distributed system!*

Don't believe us? Consider a typical SPA. This imaginary SPA contains a form with several fields. The first field is a select for choosing a country. Based on that country, we want to update the contents of a second field, a list of states or provinces. Based on the selected state, we update a yet another element on the page to display a tax amount.

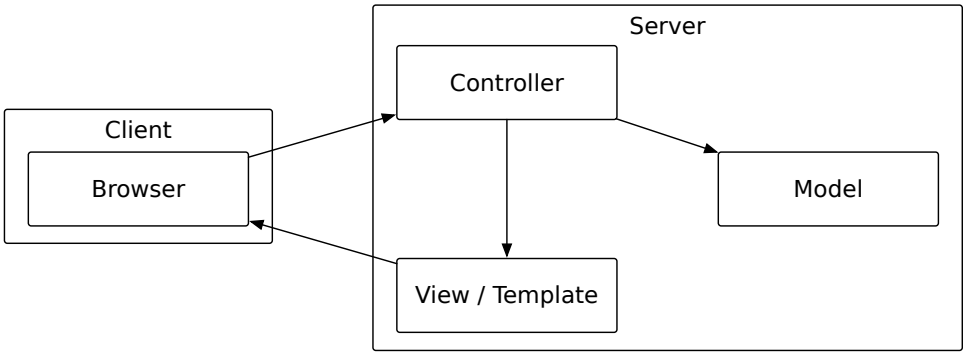
This simple hypothetical SPA breaks the mold of the traditional web application in which the user sends one request and the server sends one response representing a static page. The SPA would need JavaScript to detect when the selection in a field has changed, more code to send the data to your server, and still more server-side code to return the right data to the client. While these features aren't tough to build, they *are* tedious and error prone. You have several JavaScript elements with multiple clients on your browser page, and the failure of the JavaScript in any one of them can impact the others.

This SPA, and all SPAs, must coordinate and manage the state of the page across the client and the server. This means that single-page apps are distributed systems.

Distributed Systems are Complex

Distributed systems are software apps whose separate parts reside on multiple nodes of a network. In a distributed system, those separated parts communicate and orchestrate actions such that a single, coherent system is presented to the end-user.

Throughout much of its history, most of what we call web development has dodged the distributed systems label because the web server masked much of the complexity from us by handling all of the network communication in a common infrastructure, as in the following figure:



As you can see in this figure, everything is happening on the server:

- Our server-side system receives a request
- and then runs a server-side program, often through a layer called a *controller*
- which then possibly accesses a database, often through a layer called a *model*,
- that builds a response, often through a layer called a *view* with a *template*,
- and delivers the result back to the web browser.

Every bit of that program is contained within a single server and we rarely have to think about code that lives down on the client.

But as we've just discussed, this "request-response" mindset is no longer sufficient for conceptualizing the complex modern SPA.

If you're building a SPA with custom Javascript and some server-side layer, you can no longer claim this beautiful, simplified isolation. Web apps are now often multi-language distributed systems with JavaScript and HTML on the client, and some general purpose application language on the server.

This had made SPA development much more challenging and time-consuming than it needs to be.

SPAs are Hard

That's another bold, screaming headline, but we're willing to bet it's one that won't get much push back. It's likely that you're living with the consequences of dividing your team or your developer's mind across the client and server. These consequences include slow development cycle times, difficulty observing and remediating bugs, and much more. But it doesn't have to be this way. The typical SPA is complex because of the way we've been thinking about SPA development and the tools we've been using.

• Click HERE to purchase this book now. discuss

In truth, we can't even show a single diagram of a typical SPA because *there are no typical SPAs!* On the client side alone, JavaScript has become frighteningly complex, with many different frontend frameworks applying very different approaches.

Meanwhile, server-side code to deal with requests from client components is still often written with the old, insufficient, request-response mindset. As a result, traditional SPA tooling forces you to think about building each interaction, piece by piece. The mechanisms might vary, but most current approaches to building SPAs force us to think in terms of *interactions*—events that initiate tiny requests and responses that independently change a page in some way. The hardest part of the whole process is splitting our development across the client and server. That split has some very serious consequences.

By splitting our application development across the client and server boundary, we enable a whole class of *potential security breaches*, as a mistake in any single interaction leaves our whole page vulnerable.

By splitting our teams across the client and server, we surrender to a *slower and more complex development cycle*.

By splitting our design across the client and server, we commit to *slower and more complex bug remediation cycles.* By introducing a custom boundary between our browser and server, we dramatically complicate testing.

Want proof? If you've looked for a web development job lately, it's no great wonder that the requirements have grown so quickly. There's a single job, "full stack developer", that addresses this bloat. Developers become the proverbial frogs in their own pot of boiling water, a pot of escalating requirements without relief. Managers have boiling pots of their own, a pot of slowing development times, escalating developer salaries, and increasing requirements.

In this book, we'd like to introduce an idea. SPAs are hard because we've been thinking about them the wrong way. They're hard because we build custom solutions where common infrastructure would better serve. SPAs are hard because we think in terms of *isolated interactions* instead of *shared, evolving state.*

To make this new idea work, we need infrastructure to step into the breach between the client and server. We need tooling that lets us focus strictly on server-side development, and that relies on common infrastructure to keep the client up to date.

We need LiveView.

LiveView Makes SPAs Easy

Phoenix LiveView is a framework for building single-page flows on the web. It's an Elixir library that you will include as a dependency in your Phoenix app, allowing you to build interactive, real-time LiveView flows as part of that Phoenix application. Compared to the traditional SPA, these flows will have some characteristics that seem almost magical to many developers:

- The apps we build will be stunningly interactive.
- The apps we write will be shockingly resistant to failure.
- These interactive apps will use a framework to manage JavaScript for us, so we don't have to write our own client-side code.
- The programming model is simple, but composes well to handle complexity.
- The programming model keeps our brain in one place, firmly on the server.

All of this means that SPAs built with LiveView will be able to easily meet the interactive demands of their users. Such SPAs will be pleasurable to write and easy to maintain, spurring development teams to new heights of productivity.

This is because LiveView lets programmers make distributed applications by relying on the *infrastructure* in the LiveView library rather than forcing them to write their own custom code between the browser and server. As a result, it's no surprise that LiveView is making tremendous waves throughout the Elixir community and beyond.

LiveView is a compelling programming model for beginners and experts alike, allowing users to think about building applications in a different, more efficient way. As this book unfolds, you'll shift away from viewing the world in terms of many independent request-response interactions. Instead, you'll conceive of a SPA as a holistic state management system.

By providing functions to render state, and events to change state, LiveView gives you the infrastructure you need to build such systems. Over the course of this book, we'll acquaint you with the tools and techniques you'll use within LiveView to render web pages, capture events, and organize your code into templates and components. In other words, everything you'll need to build a distributed state management system, aka a SPA, with LiveView.

Though this is a book about a user interface technology, we'll spend plenty of time writing pure Elixir with a layered structure that integrates with our views seamlessly.

• Click HERE to purchase this book now. discuss

LiveView vs Live Views

Phoenix LiveView is one of the central actors in this book. It's the library written in the Elixir language that plugs into the Phoenix framework. *Live views* are another actor. A *live view* is comprised of the routes, modules and templates, written using the LiveView library, that represents a SPA. In this book, We'll focus more on live views than LiveView. That means we won't try to take you on a feature-by-feature grand tour. Instead, we'll build software that lasts using practical techniques with the LiveView library.

The LiveView Loop

The LiveView loop, or flow, is the core concept that you need to understand in order to build applications with LiveView. This flow represents a significant departure from the request-response mindset you might be used to applying to SPAs. This shift in mindset is one of the reasons why you'll find building SPAs with LiveView to be a smooth, efficient and enjoyable process.

Instead of thinking of each interaction on your single-page app as a discrete request with a corresponding response, LiveView manages the state of your page in a long-lived process that loops through a set of steps again and again. Your application receives events, changes the state, and then renders the state, over and over. This is the LiveView flow.

Breaking it down into steps:

- LiveView will *receive events*, like link clicks, key presses, or page submits.
- Based on those events, you'll write functions to *transform your state*.
- After you change your state, LiveView will re-render *only the portions of the page that are affected by the transformed state*.
- After rendering, LiveView again waits for events, and we go back to the top.

That's it. Everything we do for the rest of the book will work in the terms of this loop. Await events, change state, render the state, repeat.

LiveView makes it easy to manage the state of your SPA throughout this loop by abstracting away the details of client/server communication. Unlike many existing SPA frameworks, LiveView shields you from the details of distributed systems by providing some common infrastructure between the browser and the server. Your code, and your mind, will live in one place, on the server-side, and the infrastructure will manage the details.

If that sounds complicated now, don't worry. It will all come together for you. This book will teach you to think about web development in the terms of the LiveView loop: get an event, change the state, render the state. Though the examples we build will be complicated, we'll build them layer by layer so that no single layer will have more complexity than it needs to. And we'll have fun together.

Now you know what LiveView is and how it encourages us to conceive of our SPAs as a LiveView flow, rather than as a set of independent requests and responses. With this understanding under your belt, we'll turn our attention to the Elixir and OTP features that make LiveView the perfect fit for building SPAs.

LiveView, Elixir, and OTP

LiveView gives us the infrastructure we need to develop interactive, real-time, distributed web apps quickly and easily. This infrastructure, and the LiveView flow we just outlined, is made possible because of the capabilities of Elixir and OTP. Understanding just what Elixir and OTP lend LiveView illustrates why LiveView is perfectly positioned to meet the growing demand for interactivity on the web.

OTP libraries have powered many of the world's phone switches, offering stunning uptime statistics and near realtime performance. OTP plays a critical role in Phoenix, in particular in the design of Phoenix channels. Channels are the programming model in Phoenix created by Chris McCord, the creator of Phoenix. This service uses HTTP WebSockets[1] and OTP to simplify client/server interactions in Phoenix. Phoenix channels led to excellent performance and reliability numbers. Because of OTP, Phoenix, and therefore LiveView, would support *the concurrency, reliability, and performance that interactive applications demand.*

LiveView relies heavily on the use of Phoenix channels—LiveView infrastructure abstracts away the details of channel-based communication between the client and the server. Let's talk a bit about that abstraction, and how Elixir made it possible to build it.

Chris's work with OTP taught him to think in terms of the reducer functions we'll show you as this book unfolds. Elixir allowed him to string reducer functions into pipelines, and these pipelines underlie the composable nature of LiveView. At the same time, Elixir's metaprogramming patterns, in partic-

1. https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API

ular the use of macros, support a framework made up of clean abstractions. As a result of these Elixir language features, users would find a *pleasant, productive programming experience* in Phoenix LiveView.

LiveView doesn't owe all of its elegance and capability to Elixir, however. JavaScript plays a big role in the LiveView infrastructure. As the web programming field grew, frameworks like React and languages like Elm provided a new way to think about user interface development in layers. Meanwhile, frameworks like Morphdom popped up to allow seamless replacement of page elements in a customizable way. Chris took note, and the Phoenix team was able to build JavaScript features into LiveView that automate the process of changing a user interface on a socket connection. As a result, 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.*

By this point, you already know quite a bit about LiveView—what it is, how it manages state at a high level via the LiveView loop, and how its building blocks of Elixir, OTP, and JavaScript make it reliable, scalable, and easy to use. Next up, we'll outline the plan for this book and what you'll build along the way. Then you'll get your hands dirty by building your very first live view.

Program LiveView Like a Professional

LiveView meets all of the interactivity and real-time needs of your average single-page app, while being easy to build and maintain. We firmly believe that the future of Phoenix programming lies with LiveView. So, this book provides you with an on-ramp into not just LiveView, but also Phoenix. We'll cover some of the essential pieces of the Phoenix framework that you need to know in order to understand LiveView and build Phoenix LiveView apps, the right way.

We'll approach this book in the same way you'd approach building a new Phoenix LiveView app from scratch, in the wild. This means we'll walk you through the use of generators to build out the foundation of your Phoenix app, including an authentication layer. Having generated a solid base, we'll begin to customize our generated code and build new features on top of it. Finally, we'll build custom LiveView features, from scratch, and illustrate how you can organize complex LiveView applications with composable layers. This generate, customize, build-from-scratch approach is one you'll take again and again when building your own Phoenix LiveView apps in the future.

Along the way, you'll learn to use LiveView to build complex interactive applications that are exceptionally reliable, highly scalable, and strikingly easy to maintain. You'll see how LiveView lets you move fast by offering elegant patterns for code organization, and you'll find that LiveView is the perfect fit for SPA development.

Here's the plan for what we're going to build and how we're going to build it.

We're going to work on a fictional business together, a game company called Pento. Don't worry, we won't spend all of our time, or even most of our time, building games. Most of our work will focus on the back office.

In broad strokes, we'll play the part of a small team in our fictional company that's having trouble making deadlines. We'll use LiveView to attack important isolated projects, like building a product management system and an admin dashboard, that provide value for our teams. Then, we'll wrap up by building one interactive game, Pentominoes.

We'll approach this journey in four parts that mirror how you'll want to approach building your own Phoenix LiveView applications in real life. In the first part, we'll focus on using code generators to build a solid foundation for our Phoenix LiveView app, introducing you to LiveView basics as we go. In the second part, we'll shift gears to building our own custom live views from the ground up, taking you through advanced techniques for composing live views to handle sophisticated interactive flows. In the third part, we'll extend LiveView by using Phoenix's PubSub capabilities to bring real-time interactivity to your custom live views. Then, you'll put it all together in the final part to build the Pentominoes game.

Before we can do any of this work, though, we need to install LiveView, and it's past time to build a basic, functioning application. In the next few sections, we'll install the tools we need to build a Phoenix application with LiveView. Then, we'll create our baseline Phoenix app with the LiveView dependency. Finally, we'll dive into the LiveView lifecycle and build our very first live view.

Enough talking. Let's install.

• Click HERE to purchase this book now. discuss