

The
Pragmatic
Programmers

Programming Phoenix LiveView

Interactive Elixir Web Programming
Without Writing Any JavaScript



Bruce A. Tate and Sophie DeBenedetto
edited by Jacquelyn Carter



Under Construction: The book you're reading is still under development. As part of our Beta book program, we're releasing this copy well before a normal book would be released. That way you're able to get this content a couple of months before it's available in finished form, and we'll get feedback to make the book even better. The idea is that everyone wins!

Be warned: The book has not had a full technical edit, so it will contain errors. It has not been copyedited, so it will be full of typos, spelling mistakes, and the occasional creative piece of grammar. And there's been no effort spent doing layout, so you'll find bad page breaks, over-long code lines, incorrect hyphenation, and all the other ugly things that you wouldn't expect to see in a finished book. It also doesn't have an index. We can't be held liable if you use this book to try to create a spiffy application and you somehow end up with a strangely shaped farm implement instead. Despite all this, we think you'll enjoy it!

Download Updates: Throughout this process you'll be able to get updated ebooks from your account at pragprog.com/my_account. When the book is complete, you'll get the final version (and subsequent updates) from the same address.

Send us your feedback: In the meantime, we'd appreciate you sending us your feedback on this book at pragprog.com/titles/liveview/errata, or by using the links at the bottom of each page.

Thank you for being part of the Pragmatic community!

The Pragmatic Bookshelf

Programming Phoenix LiveView

Interactive Elixir Web Programming
Without Writing Any JavaScript

Bruce A. Tate
Sophie DeBenedetto

The Pragmatic Bookshelf

Raleigh, North Carolina



Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit <https://pragprog.com>.

For sales, volume licensing, and support, please contact support@pragprog.com.

For international rights, please contact rights@pragprog.com.

Copyright © 2021 The Pragmatic Programmers, LLC.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

ISBN-13: 978-1-68050-821-5

Encoded using the finest acid-free high-entropy binary digits.

Book version: B4.0—April 29, 2021

Contents

	Change History	vii
	Introduction	ix
1.	Get To Know LiveView	1
	Single-Page Apps are Distributed Systems	2
	LiveView Makes SPAs Easy	5
	Program LiveView Like a Professional	8
	Install Elixir, Postgres, Phoenix, and LiveView	9
	Create a Phoenix Project	10
	The LiveView Lifecycle	13
	Build a Simple Live View	18
	LiveView Transfers Data Efficiently	23
	Your Turn	26
	Part I — Code Generation	
2.	Phoenix and Authentication	31
	CRC: Constructors, Reducers, and Converters	33
	Phoenix is One Giant Function	37
	Generate The Authentication Layer	41
	Explore Accounts from IEx	45
	Protect Routes with Plugs	48
	Authenticate The Live View	51
	Your Turn	55
3.	Generators: Contexts and Schemas	57
	Get to Know the Phoenix Live Generator	58
	Run the phx.live Generator	59
	Understand The Generated Core	63

Understand The Generated Boundary	70
Boundary, Core, or Script?	77
Your Turn	80
4. Generators: Live Views and Templates	83
Application Inventory	84
Mount and Render the Product Index	87
Handle Change for the Product Edit	93
LiveView Layers: The Modal Component	97
LiveView Layers: The Form Component	105
Your Turn	110

Part II — LiveView Composition

5. Forms and Changesets	115
Model Change with Changesets	115
Model Change with Schemaless Changesets	117
Use Schemaless Changesets in LiveView	119
LiveView Form Bindings	126
Live Uploads	129
Your Turn	140
6. Stateless Components	143
The Survey	144
Organize Your LiveView with Components	148
Build The Survey Context	150
Organize The Application Core and Boundary	155
Build The Survey Live View	159
Build The Demographic Form	165
Your Turn	170
7. Stateful Components	173
Make the Demographic Component Stateful	173
Show a Demographic	179
Build The Ratings Components	181
List Ratings	184
Show a Rating	187
Show the Rating Form	189
Your Turn	197

Part III — Extend LiveView

8.	Build an Interactive Dashboard	201
	The Plan	202
	Define The AdminDashboardLive LiveView	204
	Represent Dashboard Concepts with Components	206
	Fetch Survey Results Data	207
	Initialize the SurveyResultsLive Component State	209
	Render SVG Charts with Context	211
	Add Filters to Make Charts Interactive	217
	Refactor Chart Code with Macros	227
	Your Turn	231
9.	Build a Distributed Dashboard	233
	LiveView and Phoenix Messaging Tools	233
	Track Real-Time Survey Results with PubSub	235
	Track Real-Time User Activity with Presence	241
	Display User Tracking	248
	Your Turn	254
10.	Test Your Live Views	255
	What Makes CRC Code Testable?	256
	Unit Test Test Survey Results State	259
	Integration Test LiveView Interactions	267
	Verify Distributed Realtime Updates	277
	Your Turn	281

Part IV — Graphics and Custom Code Organization

11.	Build The Game Core	285
	The Plan	286
	Represent a Shape With Points	288
	Group Points Together in Shapes	303
	Track and Place a Pentomino	306
	Track a Game in a Board	311
	Your Turn	314
12.	Render Graphics With SVG	317
13.	Establish Boundaries and APIs	319
	Bibliography	321

Change History

The book you're reading is in beta. This means that we update it frequently. Here is the list of the major changes that have been made at each beta release of the book, with the most recent change first.

B4.0 April 29th, 2021

- Added [Chapter 11, Build The Game Core, on page 285.](#)
- Addressed errata.

B3.0 April 2nd, 2021

- Added [Chapter 10, Test Your Live Views, on page 255.](#)
- Addressed errata.

B2.0 March 11th, 2021

- Added [Chapter 9, Build a Distributed Dashboard, on page 233.](#)
- Addressed errata.

B1.0: February 24, 2021

- Initial beta release.

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 techniques¹ 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 WebSockets² 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 Morpdom 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 31

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 57

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 83

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 115](#)

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, Stateless Components, on page 143](#)

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, Stateful Components, on page 173

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 201

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 233

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 255

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 285

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.

the (as yet) unwritten Chapter 12, Render Graphics With SVG,

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.

the (as yet) unwritten Chapter 13, Establish Boundaries and APIs,

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>

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.

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 has an advertisement, Google analytics tracking, and 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.

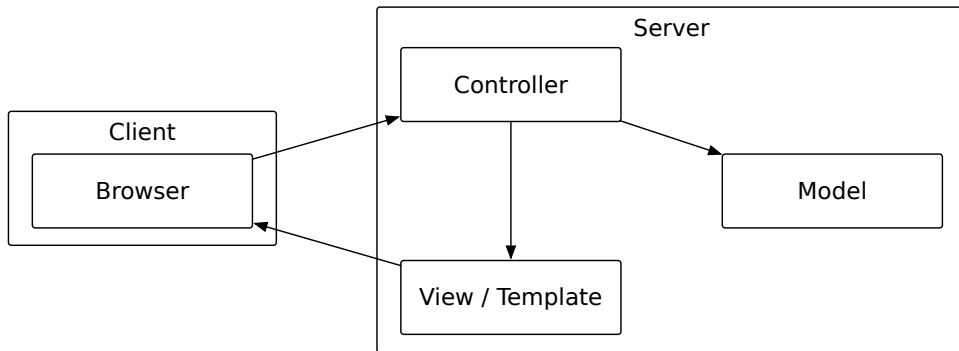
This simple hypothetical SPA breaks the mold of the traditional web application in which the user sends one request and the server sends one response representing a static page. The SPA would need JavaScript to detect when the selection in a field has changed, more code to send the data to your server, 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 server.

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.

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.

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.

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 discreet 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 *change your state*.
- After you change your state, LiveView will re-render *only the portions of the page that are affected by the changed 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 WebSockets¹ 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 Morpdom 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.

Install Elixir, Postgres, Phoenix, and LiveView

The first step is to install Phoenix, Erlang, Elixir, Node, and Postgres. Elixir is the language we'll be using, Erlang is the language it's built on, Phoenix is the web framework, Node supports the system JavaScript that LiveView uses, and PostgreSQL is the database our examples will use. If you've already done this, *you can skip this topic*.

Rather than give you a stale, error-prone procedure, we'll direct you to the Install Phoenix documentation² on the hexdocs page. It's excellent. Make sure you get the right version of Elixir (≥ 1.10 as of this writing), Erlang (≥ 21), and Phoenix (1.5). You'll also pull down or possibly update Node.js and PostgreSQL.

If you have trouble installing, use this experience as an opportunity to learn about the ecosystem that will support you when things go wrong. Go to the message board support for the framework that's breaking, or ask politely in the Elixir message boards.³ For more immediate help, you might use Elixir's Slack channels. Get the most recent support options on the Elixir slack community⁴ page on Hex. There's usually someone around to offer immediate assistance. When you ask for help, do your homework and honor those who are supporting you.

With the installation done, you're ready to create your project and set up LiveView. We'll use Mix to do so.

Create a Phoenix Project

With all of our dependencies installed, we're ready to start building the Pento app. We'll begin by setting up a new Phoenix project.

Open up an operating system shell and navigate to the parent directory for your project. Then, type:

```
mix phx.new pento --live

...
[pp_liveview] → mix phx.new pento --live
* creating pento/config/config.exs
* creating pento/config/dev.exs
* creating pento/config/prod.exs

...

Fetch and install dependencies? [Yn] Y
* running mix deps.get

...
```

The `mix phx.new` command runs the Phoenix installer for a standard Phoenix project, and the `--live` switch adds the library dependencies, configuration, and assets we'll need to build live views.

2. <https://hexdocs.pm/phoenix/installation.html>
3. <https://elixirforum.com>
4. <https://elixir-slackin.herokuapp.com>

As we work through this book, we'll point out the bits that this `--live` switch adds to our project, and we'll examine the directory structure in detail over time. For now, know that backend code goes in the `lib/pento` directory, the web-based assets like `.css` and `.js` files go in `assets`, and the web-based code all goes in the `lib/pento_web` directory.

Create The Database and Run The Server

At the bottom of the installation output, you'll find a few extra instructions that look something like this:

...

We are almost there! The following steps are missing:

```
$ cd pento
$ mix deps.get
$ cd assets && npm install && node node_modules/webpack/bin/webpack.js \
  --mode development
```

Then configure your database in `config/dev.exs` and run:

```
$ mix ecto.create
```

Start your Phoenix app with:

```
$ mix phx.server
```

You can also run your app inside IEx (Interactive Elixir) as:

```
$ iex -S mix phx.server
```

Note that you might see slightly different output depending on your Phoenix version.

Let's follow those instructions now by performing the following actions. First, make sure you have Postgres installed and running on localhost, accessible with the default username `postgres` and password `postgres`. See the PostgreSQL Getting Started⁵ guide for help.

Then, change to the `pento` directory, and create your database:

```
[intro] → cd pento/
[pento] → mix ecto.create
Compiling 13 files (.ex)
Generated pento app
The database for Pento.Repo has been created
```

Now, start the web server:

```
[pento] → mix phx.server
```

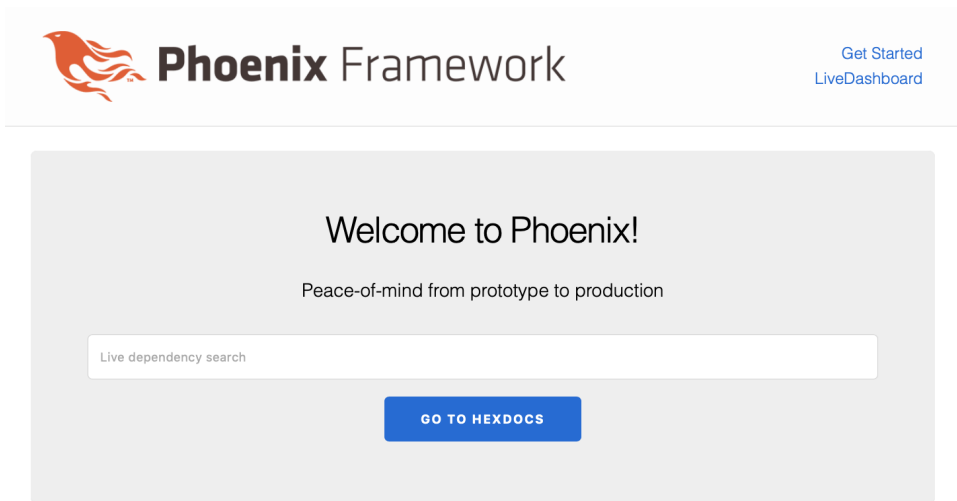
5. <https://www.postgresqltutorial.com/postgresql-getting-started/>

```
[info] Running PentoWeb.Endpoint with cowboy 2.8.0 at 0.0.0.0:4000 (http)
[info] Access PentoWeb.Endpoint at http://localhost:4000

webpack is watching the files...
```

```
...
```

Point your browser to localhost:4000/ and if you've installed correctly, you'll see the following image.



We're up and running! Let's see what the Phoenix generator did for us.

View Mix Dependencies

You just installed both Elixir and Phoenix, and created the application skeleton. The `--live` switch injected a few extra dependencies in our application. Let's take a closer look at them now.

Mix installed the libraries LiveView will need as *Mix dependencies*. Every Phoenix application uses the underlying mix tool to fetch and manage dependencies. The `mix.exs` file contains the instructions for which dependencies to install and how to run them. Crack it open and take a look:

```
intro/pento/mix.exs
```

```
defp deps do
  [
    {:bcrypt_elixir, "~> 2.0"},
    {:phoenix, "~> 1.5.3"},
    {:phoenix_ecto, "~> 4.1"},
    {:ecto_sql, "~> 3.4"},
    {:postgrex, "=> 0.0.0"},
    {:phoenix_live_view, "~> 0.15.1"},
  ]
end
```

```

{:floki, ">= 0.27.0", only: :test},
{:phoenix_html, "~> 2.11"},
{:phoenix_live_reload, "~> 1.2", only: :dev},
{:phoenix_live_dashboard, "~> 0.4.0"},
{:telemetry_metrics, "~> 0.4"},
{:telemetry_poller, "~> 0.4"},
{:gettext, "~> 0.11"},
{:jason, "~> 1.0"},
{:plug_cowboy, "~> 2.0"},
]
end

```

The `mix.exs` file ends with `.exs`, so it's an Elixir script. Think of this script as the configuration details for your app. Each line in the `deps` list is a dependency for your app. You may have noticed that Phoenix fetched the dependencies on this list when you ran `mix deps.get`. These dependencies are not hidden in some archive. You can actually see them and look at the code within each one. They are in the `deps` directory:

```
[pento] → ls deps
```

connection	html_entities	plug
cowboy	jason	plug_cowboy
cowlib	mime	plug_crypto
db_connection	phoenix	postgrex
decimal	phoenix_ecto	ranch
ecto	phoenix_html	telemetry
ecto_sql	phoenix_live_dashboard	telemetry_metrics
file_system	phoenix_live_reload	telemetry_poller
floki	phoenix_live_view	
gettext	phoenix_pubsub	

Those are the dependencies we've already installed. You might see a slightly different list based on your version. The LiveView dependencies are `phoenix_live_view`, `phoenix_live_dashboard` for system monitoring, and `floki` for tests. We also have a few dependencies our LiveView dependencies require.

Now that you understand how LiveView integrates into your Phoenix app as a Mix dependency, we're almost ready to write our first LiveView code. First, you need to understand the LiveView lifecycle— how it starts up and how it runs to handle user events and manage the state of your single-page app.

The LiveView Lifecycle

Before we build our first live view, let's take a deeper dive into the LiveView lifecycle we touched upon earlier when we discussed the LiveView loop. We'll walk you through how LiveView manages the state of your single-page app in a data structure called a socket, and how LiveView starts up, renders the

page for the user and responds to events. Once you understand the LiveView lifecycle, you'll be ready to build out this lifecycle and use it to manage the state of your own live views.

We'll begin by examining how LiveView represents state via `Phoenix.LiveView.Socket` structs. Understanding how the socket struct is constructed and updated will give you the tools you need to establish and change the state of your live views.

Hold State in LiveView Sockets

When all is said and done, live views are about state, and LiveView manages state in structs called sockets. The module `Phoenix.LiveView.Socket` creates these structs. Whenever you see one of these socket structs as a variable or an argument within a live view, you should immediately recognize it as the data that constitutes the live view's state.

Let's take a closer look at a socket struct now.

Go to the `pento` directory, and open up an IEx session for your application with `iex -S mix`. Then, request help:

```
iex> h Phoenix.LiveView.Socket
Phoenix.LiveView.Socket
```

The LiveView socket for Phoenix Endpoints.

That wasn't too helpful. That's OK. Elixir gives us more tools for understanding code than this one. Let's build a new socket:

```
iex(5)> Phoenix.LiveView.Socket.__struct__
#Phoenix.LiveView.Socket<
  assigns: %{},
  changed: %{},
  endpoint: nil,
  id: nil,
  parent_pid: nil,
  root_pid: nil,
  router: nil,
  view: nil,
  ...
>
```

That's better. Here, you can see the basic structure of a socket struct and start to get an idea of how socket structs represent live view state. The socket struct has all of the data that Phoenix needs to manage a LiveView connection, and the data contained in this struct is mostly private. The most important key, and the one you'll interact with most frequently in your live views, is

assigns: %{}). That's where you'll keep all of a given live view's custom data describing the state of your SPA.

That's the first lesson. Every running live view keeps data describing state in a socket. You'll establish and update that state by interacting with the socket struct's `:assigns` key.

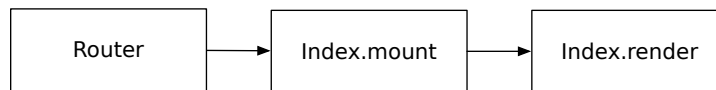
Let's talk about how LiveView establishes that state in the first place and renders it.

Render the Live View

In order to render a live view, we need to understand how the LiveView lifecycle starts.

The LiveView lifecycle begins in the Phoenix router. That is where you will define a special type of route called a “live route”. A live route maps an incoming web request to a specified live view so that a request to that endpoint will start up the live view process. That process will initialize the live view's state by setting up the socket in a function called `mount/3`. Then, the live view will render that state in some markup for the client.

The following figure tells the story:



That's simple enough. In the router, we define the route and map it to a given LiveView module. For LiveView routes, or live routes, defined with the `live/3` function, the router calls the `mount` function on the module we specify.

Let's take a look at an example live route definition. The following live route was generated for us when we created our new Phoenix application with the `--live` flag via the command line.

```

intro/pento/lib/pento_web/router.ex
scope "/", PentoWeb do
  pipe_through :browser

  live "/", PageLive, :index

```

Here, we see a LiveView route defined with the `live/3` function. This will map the given incoming web request to the provided LiveView module. The `live/3` function takes a final optional argument called a *live action*. Don't worry about what it does for now. This code means that the LiveView module that will handle requests to `/` is called `PentoWeb.PageLive`.

When your Phoenix app receives a request to the / route, the PageLive live view will start up, and LiveView will invoke that module's mount/3 function. The mount/3 function is responsible for establishing the initial state for the live view by populating the socket assigns. Let's take a look at that function now:

```
intro/pento/lib/pento_web/live/page_live.ex
def mount(_params, _session, socket) do
  {:ok, assign(socket, query: "", results: %{})}
end
```

Remember, the socket contains the data representing the state of the live view, and the :assigns key, referred to as the “socket assigns”, holds custom data. Setting values in maps in Elixir can be tedious, so the LiveView helper function assign/2 simply adds key/value pairs to a given socket assigns. This code is setting up our socket assigns with a search query string and an empty results map.

That means our initial socket looks something like this:

```
%Socket{
  assigns: %{
    query: "",
    results: %{}
  }
}
```

The mount function returns a result tuple. The first element is either :ok or :error, and the second element has the initial contents of the socket.

After the initial mount finishes, LiveView then passes the value of the socket.assigns key to a render function. If there's no render function, LiveView looks for a template to render based on the name of the view. Don't worry about these details now. Just know that initially, LiveView calls mount, and then render with those results.

Since there's no render function in this live view, we'll look for a template. Sure enough, there's one in live/page_live.html.leex. Here's a bit of that template:

```
<input
  type="text"
  name="q"
  value="<%= @query %>"
  placeholder="Live dependency search"
  list="results"
  autocomplete="off"/>
```

Notice the <%= @query %> expression. LiveView will populate this code with the value of socket.assigns.query, which we set in mount.

When LiveView finishes calling `mount` and then `render`, it returns the initial web page to the browser.

For a traditional web page, the story would end there. With LiveView, we're just getting started. After the initial web page is rendered in the browser, LiveView establishes a persistent WebSocket connection and awaits events over that connection. Let's look at this next part of the LiveView lifecycle now.

The Initial Render Respects SEO

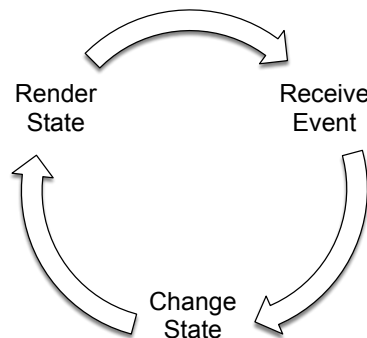
We should take a quick moment to point out a striking benefit of LiveView. The initial render works just like the render for a static page. For the purposes of search engine optimization, your initial page will show Google the same thing it tells your users!

Run the LiveView Loop

So far, we've seen LiveView receive request, set up the initial data, and render it. We've not addressed any of the technology that makes a live view interactive. We'll do that now.

When Phoenix processes a LiveView request, two things happen. First, Phoenix processes a plain HTTP request. The router invokes the LiveView module, and that calls the `mount/3` function and then `render/1`. This first pass renders a static, SEO-friendly page that includes some JavaScript. That page then opens a persistent connection between the client and the server using WebSockets.

After Phoenix opens the WebSocket connection, our LiveView program will call `mount/3` and `render/1` *again*. At this point, the LiveView lifecycle starts up the *LiveView loop*. The live view can now receive events, change the state, and render the state again. This loop repeats whenever live view receives a new event, like this figure shows:



Code structured in-line with this flow is simple to understand and easy to build. We don't have to worry about how events get sent to a live view or how markup is re-rendered when state changes. While we do have to implement our own event handler functions, and teach them how to change state, LiveView does the hard work of detecting events, such as form submits or clicks on a link, and invokes those handlers for us. Then, once our handlers have changed the state, LiveView triggers a new render based on those changes. Finally, LiveView returns to the top of the loop to process more events.

What you have is a pure, functional render function to deal with the complexities of rendering the user interface, and an event loop that receives events that change the state. Most of the hard problems—like delivering an event from the client to the server, detecting state changes, and re-rendering the page—stay isolated in the infrastructure, where they belong.

That's enough talking for now. It's time to put what you've learned into practice!

Build a Simple Live View

The Pento app's first live view will be a simple game called “You're Wrong!”. It's one you can give to your kids to keep them busy for hours. It will ask them to guess a number, and then tell them they're wrong.

We'll have to do three things—build a route, render a page, and handle the events. Let's go!

Define the Route

LiveView routes, called live routes, are defined in your Phoenix application's router. Open up `lib/pento_web/router.ex` and add the live route for `"/guess"`:

```
scope "/", PentoWeb do
  pipe_through :browser

  live "/", PageLive, :index
  live "/guess", WrongLive
end
```

Most Phoenix route definitions use HTTP methods,⁶ primarily get and post. A live view route is a little different. Live view routes are defined with a call to the live macro (more on that later), and point to a long-running live view. The initial HTTP request and response will flow through this route. After that, the WebSocket will handle the communication.

6. <https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods>

Now that we've defined our route, let's build the `WrongLive` module's initial render/1 function.

Render the Live View

Let's shift to the code that implements our live view. Create a new file inside `lib/pento_web/live` called `wrong_live.ex` and key this in. It's short, but the concepts are foundational so we'll address it in three parts.

First, we define the live view module:

```
intro/pento/lib/pento_web/live/wrong_live.ex
defmodule PentoWeb.WrongLive do
  use PentoWeb, :live_view
```

The `use PentoWeb, :live_view` macro makes available the special sigil we'll use within the render function. More on that in a bit.

Next, remember what happens when a live view is initialized. The request comes in through the router, and then the router calls the `mount/3` function to establish initial state:

```
intro/pento/lib/pento_web/live/wrong_live.ex
def mount(_params, _session, socket) do
  {
    :ok,
    assign(
      socket,
      score: 0,
      message: "Guess a number."
    )
  }
end
```

The `mount/3` function takes some initial information about the request in `_session`, the input parameters called `_params`, and a socket which is a simple struct.

The socket is nothing more than a representation of the state for a live view, and the `assign` function puts custom data into the `socket.assigns` map. We put the value of 0 into `assigns.score`, and the message "Guess a number." into `assigns.message`.

Now, the socket looks something like this:

```
%{
  assigns: %{
    score: 0,
    message: "Guess a number."
  }
}
```

We have what we need to display the initial web page for our game. Let's build some markup and fill in the details:

```
intro/pento/lib/pento_web/live/wrong_live.ex
```

```
def render(assigns) do
  ~L"""
  <h1>Your score: <%= @score %></h1>
  <h2>
    <%= @message %>
  </h2>
  <h2>
    <%= for n <- 1..10 do %>
      <a href="#" phx-click="guess" phx-value-number="<%= n %>"><%= n %></a>
    <% end %>
  </h2>
  """
end
```

In PageLive, you saw the template form of render. For WrongLive, we defined an explicit render/1 function that takes an argument of socket.assigns and renders some markup that will be sent down to the client. Here's how it works.

The ~L"" and "" bracket the HTML code. The ~L is a sigil. That means there's a function called sigil_L, and one of the features of that function is the processing of template replacements. Everything between the <%= and %> expressions is a template replacement string in a language called *LEEx*.⁷ Remember, the @message and @score expressions are actually keys from the socket.assigns map.

LiveView will render all of the *static markup* in the template exactly once, since that code will never change. It will replace any template replacement string *only when the value changes*.

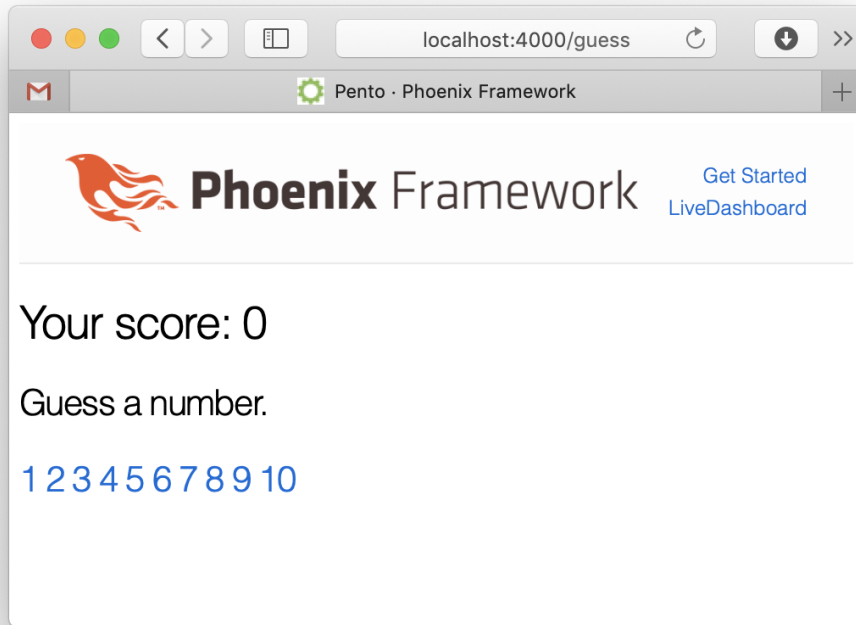
The code in our render function will show a message, and then some links. Let's look at one of these links now. The for comprehension will iterate through numbers 1 to 10, filling in the value n for each of the links. We're left with something like this:

```
<a href="#" phx-click="guess" phx-value-number="1">1</a>
```

That's a link that leads to nowhere, but it has two values, a phx-click and a phx-value-number. We'll use that data when it's time to process events. The page will have similar links for n=2 all the way up through n=10.

Tack on an end at the end of the module and make sure you've started your server with mix phx.server. Next, point your browser to localhost:4000/guess. You'll see something like the following:

7. https://hexdocs.pm/phoenix_live_view/Phoenix.LiveView.Engine.html



That's the user interface for the game. As expected, we see the message we put into assigns, and links for each of the 10 integers. Now, click on one of the links.

And... it fails. There's good news too, though. The application came back up! That's one of the perks of running on Elixir.

Flip on over to the console, and you'll see the message:

```
[error] GenServer #PID<0.2147.0> terminating
** (UndefinedFunctionError) function PentoWeb.WrongLive.handle_event/3
    is undefined or private
    PentoWeb.WrongLive.handle_event("guess", %{"number" => "10"},
      #Phoenix.LiveView.Socket<...>)
```

You can see that our program received a message it wasn't ready to handle. When the event came in, LiveView called the function `handle_event("guess", some-map-data, our-socket)`, but no one was home—no such function is implemented by the `WrongLive` module. Let's fix that.

Handle Events

Finishing off our game isn't going to take as much effort as you might expect because we won't be building routes for our links, or building controllers, or

templates, or models—all of our data will flow over the same socket and be handled by one live view module. We'll simply build a handler for our inbound event.

The tricky part is matching the inbound data. Remember those extra data elements to our `<a>` links? These will come into play now. As you saw, the inbound data will trigger the function `handle_event/3` with three arguments.

The first is the message name, the one we set in `phx-click`.

The second is a map with the metadata related to the event.

The last is the state for our live view, the socket.

Let's implement the `handle_event/3` function now:

```
intro/pento/lib/pento_web/live/wrong_live.ex
def handle_event("guess", %{"number" => guess}=data, socket) do
  IO.inspect data
  message = "Your guess: #{guess}. Wrong. Guess again. "
  score = socket.assigns.score - 1

  {
    :noreply,
    assign(
      socket,
      message: message,
      score: score)
  }
end
```

Look at the function head first. It uses Elixir's pattern matching to do the heavy lifting. You can see that we match only function calls where the first argument is `"guess"`, and the second is a map with a key of `"number"`. Those are the arguments we set in our `phx-click` and `phx-value` link attributes.

The job of this function is to change the live view's state based on the inbound event, so we need to transform the data within `socket.assigns`. We knock one point off of the score, and set a new message. Then, we set the new data in the `socket.assigns` map. Finally, we return a tuple in the shape that LiveView expects—`{:noreply, socket}`. This update to `socket.assigns` triggers the live view to re-render by sending some changes down to the client over the persistent WebSocket connection.

Now you can play the game for yourself. If your will isn't strong, be careful. The game is strangely addictive:



Phoenix Framework

Your score: -113

Your guess: 3. Wrong. Guess again.

1 2 3 4 5 6 7 8 9 10

If LiveView still seems a little mysterious to you, that's okay. We're ready to fill in a few more details.

LiveView Transfers Data Efficiently

Now you know how LiveView operates by handling an initial HTTP request, then opening a WebSocket connection and running an event loop over that connection. Understanding how LiveView transfers data to the client over the WebSocket connection is the final piece of the LiveView puzzle.

You know that LiveView re-renders the page by sending UI changes down to the client in response to state changes. What you might not know however, is that LiveView sends these changes in a manner that is highly efficient. LiveView applications can therefore be faster and more reliable than similar alternatives composed completely from scratch in lower level frameworks such as Phoenix or Rails.

We can examine the network traffic in our browser to illustrate exactly how LiveView sends diffs and see just how efficient it is for ourselves. In fact, we recommend getting into the habit of inspecting this network traffic when you're developing your live views to ensure that you're not asking LiveView to transfer too much data.

This section uses the Safari browser client to inspect network traffic, but you can use almost any modern web browser to get similar information.

Examine Network Traffic

Open up the developer tools for your browser and navigate to the network tab. Now, click a number on the page, and then click on either of the two websocket entries under the open network tab. In one of them, you should see the actual data that's sent up to the browser when the user clicks a number.

```
[ "4", "5", "lv:phx-1Yf0NAIF",
  "event", { "type": "click", "event": "guess",
    "value": { "number": "8" } } ]
```

The data here is formatted with some line breaks, but it's otherwise left intact. Other than a small bit of data in a header and footer, this data is information about the mouse click, including whether certain keys were pressed, the location of the cursor, and the like. We'll get data packets like this only for the links and key presses that we request.

Next, let's look at the data that goes back down to the client. Clicking on the other websocket entry should show you something like this:

```
[ "4", "5", "lv:phx-1Yf0NAIF", "phx_reply",
  { "response": {
    "diff": {
      "0": "-1",
      "1": "Your guess: 8. Wrong. Guess again. "
    }
  },
  "status": "ok" }
] 1579361038.5015142
```

Here is the data that LiveView has sent over the WebSocket connection in response to some state change. This payload only contains a small header and footer, along with *changes to the web page*, including the score and message we set in the `handle_event/3` function.

Look at the critical part of the message, the `diff`. It represents the changes in the view *since the last time we rendered!* You can see that LiveView sends the smallest possible payload of diffs to the client—only the information describing what changed in state, and therefore what needs to change on the page, is communicated. Keeping data payloads as small as possible helps ensure LiveView's efficiency.

Now, let's see how LiveView actually detects the changes to send down to the client.

Send Network Diffs

Let's explore how exactly LiveView knows what diffs to send and when. We'll update our view to include a clock.

Update your render function to include the time, like this:

```
<h2>
  <%= @message %>
  It's <%= time() %>
</h2>
```

Add this function right below the render:

```
def time() do
  DateTime.utc_now |> to_string
end
```

And take a look at your reloaded browser:

```
Your score: 0
Guess a number. It's 2020-01-18 15:53:40.209764Z
1 2 3 4 5 6 7 8 9 10
```

So far so good. You can see the time in the initial page load, 15:53:40.

Now, make a guess:

```
Your score: -1
Your guess: 5. Wrong. Guess again. It's 2020-01-18 15:53:40.209764Z
1 2 3 4 5 6 7 8 9 10
```

Even though the page updated, the time is *exactly the same*. The problem is that we didn't give LiveView any way to determine that the value should change and be re-rendered.

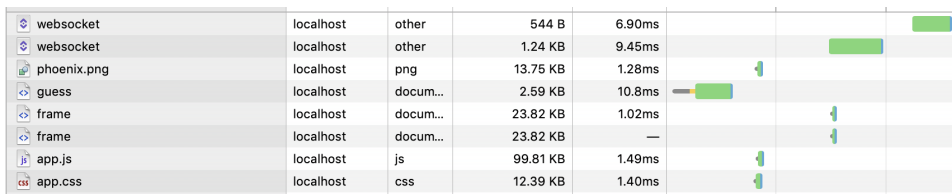
When you want to track changes, make sure to use socket assigns values such as `@score` in your templates. LiveView keeps track of the data in socket assigns and any changes to that data instruct LiveView to send a diff down to the client. Diffs describe only what changed in the socket assigns and LiveView re-renders only the portions of the page impacted by those changes.

So, although LiveView re-rendered the page when it handled the click event, LiveView did *not* consider the portion of the template including the invocation of the `time/0` function to have changed. Therefore that portion of the template was not re-rendered, the `time/0` was not re-invoked and the time did not update on the page.

We can fix this by assigning a time to the socket when we mount, rendering that value in the template, and changing that value when we receive events. We'll leave those changes as an exercise for the reader.

LiveView's Efficiency is SEO Friendly

If you refresh the page and then check out your network tab again in your inspector, you'll see that the initial page load *looks like any other page load*. Phoenix sends the main page and the assets just as it normally would for any HTTP request/response, as you can see in the following image of the browser's network tab. Pay close attention to the guess request line, which shows that the response is a simple HTML document.



Resource	Host	Type	Size	Time	Status
websocket	localhost	other	544 B	6.90ms	200
websocket	localhost	other	1.24 KB	9.45ms	200
phoenix.png	localhost	png	13.75 KB	1.28ms	200
guess	localhost	docum...	2.59 KB	10.8ms	200
frame	localhost	docum...	23.82 KB	1.02ms	200
frame	localhost	docum...	23.82 KB	—	200
app.js	localhost	js	99.81 KB	1.49ms	200
app.css	localhost	css	12.39 KB	1.40ms	200

Many one-page applications render pages that can't be used for SEO (search engine optimization). Because those apps must render the page in parts, Google just can't tell what's on the whole page.

Before LiveView, solving this problem was inevitably expensive. With LiveView, the initial page load looks like any other page to a search engine. Only after the initial page load completes does LiveView establish the WebSocket-backed LiveView loop in which your live view listens for events, updates state, and efficiently re-renders *only the portions of the page described in the network diff events*. You get SEO right out of the box, without impacting the efficiency of LiveView.

Now, you understand the basics of LiveView. It's time to put what you know to use.

Your Turn

LiveView is a library for building highly interactive single-page web flows called live views, without requiring you to write JavaScript. A live view:

- Has internal state
- Has a render function to render that state
- Receives events which change state
- Calls the render function when state changes
- Only sends changed data to the browser

- Needs no custom JavaScript

When we build live views, we focus on managing and rendering our view's state, called a socket. We manage our live view's state by assigning an initial value in the `mount/3` function, and by updating that value using several handler functions. Those functions can handle input from processes elsewhere in our application, as well as manage events triggered by the user on the page, such as mouse clicks or keystroke presses. After a handler function is invoked, LiveView renders the changed state with the `render/1` function.

This is the LiveView lifecycle in a nutshell. As we build live views that handle increasingly complex interactive features over the course of this book, you'll see how the LiveView framework allows you to be amazingly productive at building single-page apps. By providing an infrastructure that manages client/server communication in a manner that is reliable and scalable, LiveView frees you up to focus on what really matters—shipping features that deliver value to your users.

Give It a Try

Now that you've seen a basic LiveView “game”, you can tweak the game so that the user can actually win. You'll need to:

- Assign a random number to the socket when the game is created, one the user will need to guess.
- Check for that number in the `handle_event` for guess.
- Award points for a right guess.
- Show a winning message when the user wins.
- Show a restart message and button when the user wins. Hint: you might want to check out the `live_patch/2`⁸ function to help you build that button.

Next Time

In the next chapter, we're going to start work on the Pento application's infrastructure, beginning with the authentication layer. We'll build out this layer using a code generator. Along the way, we'll take the opportunity to explore how Phoenix requests work, and we'll show you how to use the generated authentication service to authenticate users. Lastly, you'll use the service to authenticate the guessing game live view you just built.

We're just getting started. Let's get to work.

8. https://hexdocs.pm/phoenix_live_view/Phoenix.LiveView.Helpers.html#live_patch/2

Part I

Code Generation

Both LiveView and the greater Phoenix ecosystem have outstanding support for code generation and you'll use generators often to build a solid foundation for your Phoenix LiveView apps. We'll use a code generator to build a secure and user-friendly authentication scheme that we'll use throughout the rest of the book. Then, we'll use a generator to build a LiveView frontend for creating and managing a database of products. Along the way, we'll use the generated code to illustrate core code organizational concepts.

Phoenix and Authentication

In this chapter, we're going to use a code generator to build an authentication layer for our Phoenix application. This is an approach that you'll often take when building out a new Phoenix LiveView app. You'll start with a web app essential, authentication, and reach for a tried and tested generator to get up and running quickly.

Let's look a little closer at the role authentication will play in Pento.

While authentication is not a LiveView concern per se, it will still serve an important purpose for us. On the web, users do things. Authentication services tell us which users are doing which things by tying the id of a user to a session.¹ More specifically, authentication allows us to:

Manage Users

One important feature of our authentication service is the ability to store users and tokens, lookup users by password, and so on.

Authenticate Requests

As requests come in, we need a way to check if the user that made the request is logged in or not so our application knows which page to show. A logged out user might get the sign-in page; a logged in user might get a game, and so on.

Manage Sessions

Our application will need to track *session data*, including information about the logged in user and the expiration of that login, if any. We'll manage this data in cookies, just as web applications built in other frameworks do.

1. <https://developer.mozilla.org/en-US/docs/Web/HTTP/Session>

You don't need to know every detail of how these services work, but you do need to understand in broad strokes what's happening. Because our live views will need to know which user is logged in, we'll rely on these critical responsibilities enacted by the authentication service throughout our LiveView code.

For example, our system will support surveys. We'll use authentication to force users to sign in before taking the survey, and to make the signed in user available to the live view. So, we're going to start the work of building our application with authentication—the act of attaching a user's conversation through browser requests to a user in your system.

We're also going to look at how plain old boring Phoenix works with traditional requests and responses. Every LiveView must start in the browser as a traditional HTTP request. Then, the request will flow through many Phoenix services, culminating in the router where we'll redirect unauthenticated users and attach a user ID to the session before LiveView ever gets involved. That means you need to understand how the Phoenix endpoints and routers work to do even the most basic of tasks.

Before we write any code, let's plan our trip. Let's look at the basic application we've already generated. We'll walk through what happens when a fresh request comes into Phoenix and trace it through the various layers. That journey will take us through an *endpoint* and into the *router*, and finally into the various modules that make up our custom application.

Then, we're going to implement our authentication code. We'll generate the bulk of our code with the `phx.gen.auth` generator, and then we'll tweak that code to do what we want. This generator is by far the best solution for Phoenix authentication.

After we generate the code, we'll work through the code base to explore the main authentication service APIs and we'll demonstrate how the generated code can be used to authenticate a live view.

By the end of this chapter, you'll understand how Phoenix handles web requests, and you'll be able to recognize that same pattern at play in LiveView code later on. You'll experience the recommended way to build and use authentication in your Phoenix app and be able to integrate authentication into your live views.

Let's get to work, starting with a common pattern called CRC.

CRC: Constructors, Reducers, and Converters

Pipelines and functional composition play a big role in Elixir. One pattern called CRC plays a huge role in many different Elixir modules. Its roots are closely entwined with the common function `Enum.reduce/3`. Let's take a closer look.

Web frameworks in functional languages all use variations of a common pattern. They use data represented by a common data type, and use many tiny, focused functions to change that data, step by step. For example, in the JavaScript world, the state reducer pattern² by Kent Dodds uses many of the same strategies. Clojure has a similar framework called Ring.³

In Phoenix, the Plug framework follows the same pattern. Let's explore this pattern in more detail.

In Elixir, many modules are associated with a *core type*. The `String` module deals with strings, `Enum` deals with enumerables, and so on. As often as possible, experienced Elixir developers strive to make a module's public functions relate to its core type. Constructors create a term of the core type from convenient inputs. Reducers transform a term of the core type to another term of that type. Converters convert the core type to some other type. Taken together, we'll call this pattern CRC.

So far, CRC might seem abstract, so let's take a simple tangible example. Let's build a module that has one of each of these functions:

```
iex(1)> defmodule Number do
... (1)>   def new(string), do: Integer.parse(string) |> elem(0)
... (1)>   def add(number, addend), do: number + addend
... (1)>   def to_string(number), do: Integer.to_string(number)
... (1)> end
```

Notice that this tiny module works with *integers*, and has three kinds of functions. All of them deal with integers as an input argument, output, or both. The `new/1` function is a constructor, and it's used to create a term of the module's type from a `String` input. The `to_string/1` function is a converter that takes an integer input and produces output of some other type, a `String` in our case. The `add/2` reducer takes an integer as both the input and output.

Let's put it to use in two different ways. first, let's use the `reduce/3` function with our three functions.

2. <https://kentcdodds.com/blog/the-state-reducer-pattern-with-react-hooks>

3. <https://github.com/ring-clojure/ring>


```

iex(2)> list = [1, 2, 3]
[1, 2, 3]
iex(3)> total = Number.new("0")
0
iex(4)> reducer = &Number.add(&2, &1)
#Function<13.126501267/2 in :erl_eval.expr/5>
iex(5)> converter = &Number.to_string/1
&Number.to_string/1
iex(6)> Enum.reduce(list, total, reducer) |> converter.()
"6"

```

We take a list full of integers and a string that we feed into our constructor that produces an integer we can use with our reducer. Since `Enum.reduce/3` takes the accumulator as the second argument, we build a `reducer/2` function that flips the first two arguments around. Then, we call `Enum.reduce/3`, and pipe that result into the converter.

It turns out that the same kinds of functions that work in reducers also work in pipes, like this:

```

iex(7)> [first, second, third] = list
[1, 2, 3]
iex(16)> "0" |> Number.new \
... (16)> |> Number.add(first) \
... (16)> |> Number.add(second) \
... (16)> |> Number.add(third) \
... (16)> |> Number.to_string
"6"

```

Perfect! The backslash at the end of each line tells IEx to delay execution because we have more to do. The functions in this `Number` module show an example of CRC, but it's not the only one. This pattern is great for taking something complicated, like breaking down the response to a complex request, down into many small steps. It also lets us build tiny functions that each focus on one thing.

CRC in Phoenix

Phoenix processes requests with the CRC pattern. The central type of many Phoenix modules is a *connection* struct defined by the `Plug.Conn` module. The connection represents a web request. We can then break down a response into a bunch of smaller reducers that each process a tiny part of the request, followed by a short converter. Here's what the program looks like:

```

connection
|> process_part_of_request(...)
|> process_part_of_request(...)
|> render()

```

You can see CRC in play. Phoenix itself serves as the constructor. It builds a common piece of data that has both request data and response data. Initially, the request data is populated with information about the request, but the response data is empty. Then, Phoenix developers build a response, piece by piece, with small reducers. Finally, Phoenix converts the connection to a response with the `render/1` converter.

Let's make this example just a little more concrete. Say we wanted to have our web server build a response to some request, piece by piece. We might have some code that looks like this:

```
iex(4)> connection = %{request_path: "http://mysite.com/"}
%{request_path: "http://mysite.com/"}
iex(5)> reducer = fn acc, key, value -> Map.put(acc, key, value) end
#Function<19.126501267/3 in :erl_eval.expr/5>
iex(6)> connection |> reducer.(status, 200) |> reducer.(body, ok)
%{body: :ok, request_path: "http://mysite.com/", status: 200}
```

Notice the two main concepts at play. First is the common data structure, the connection. The second is a function that takes an argument, called `acc` for accumulator, that we'll use for our connection, and two arguments. Our function is called a *reducer* because we can reduce an accumulator and a few arguments into a single accumulator.

Now, with our fictional program, we can string together a narrative that represents a web request. For our request, we take the connection, and then we pass that connection through two reducers to set the status to 200 and the body to `:ok`. After we've built a map in this way, we can then give it back to our web server by passing it to our `render/1` converter to send the correct body with the correct status down to the client.

Now that we have a high-level understanding of how Phoenix strings together a series of functions to respond to a web request, let's look at the specifics. As we go, pay attention to the plugs. Each one is a reducer that accepts a `Plug.Conn` as an input, does some work, and returns a transformed `Plug.Conn`.

The Plug.Conn Common Data Structure

Plug is a framework for building web programs, one function at a time. Plugs are either Elixir functions, or tiny modules that support a small function named `call`. Each function makes one little change to a connection—the `Plug.Conn` data structure. A web server simply lets developers easily string many such plugs together to define the various policies and flows that make up an application. Chris McCord took the Plug toolkit and used it to build Phoenix.

You don't have to guess what's inside. You can see it for yourself. Type `iex -S mix` to launch interactive Elixir in the context of your Phoenix application. Key in an empty `Plug.Conn` struct and hit enter. You should see these default values:

```
iex> %Plug.Conn{}
%Plug.Conn{
  ...
  host: "www.example.com",
  method: "GET",
  ...
  resp_body: nil,
  resp_headers: [{"cache-control", "max-age=0, private, must-revalidate"}],
  status: nil
  ...
}
```

We've cut out most of the keys, but left a few in place for context. Some are related to the inbound *request*, including the host, the request method,⁴ and so on. Some are related to the *response*. For example, the response headers are pieces of data to control caching, specify the response type, and more. The response status is the standardized http status.⁵

So that's the “common data structure” piece of the equation. Next, we'll look at the reducer.

Reducers in Plug

Now, you've seen `Plug.Conn`, the data that stitches Phoenix programs together. You don't need to know too much to understand many of the files that make up a Phoenix application beyond three main concepts:

- Plugs are reducer functions
- They take a `Plug.Conn` struct as the first argument
- They return a `Plug.Conn` struct.

When you see Phoenix configuration code, it's often full of plugs. When you see lists of plugs, imagine a pipe operator between them. For example, when you see something like this:

```
plug Plug.MethodOverride
plug Plug.Head
plug Plug.Session, @session_options
plug PentoWeb.Router
```

you should mentally translate that code to this:

4. <https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods>

5. <https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>

```

connection
|> Plug.MethodOverride.call()
|> Plug.Head.call()
|> Plug.Session.call(@session_options)
|> PentoWeb.Router.call()

```

Said another way, lists of plugs are composed with *pipelines*, plus a small amount of sugar to handle failure.

Now, with that background, we're going to look at the heart of your Phoenix infrastructure, and even if you have only a small amount of experience with Phoenix, you'll be able to understand it. Keep in mind that this information will come in handy because it will help you understand exactly what happens when a live view runs.

Phoenix is One Giant Function

In order to understand how Phoenix handles web requests, and therefore how LiveView handles web requests, you can think of Phoenix requests as simply one big function broken down into smaller plugs. These plugs are stitched together, one after another, as if they were in one big pipeline.

The main sections of the giant Phoenix pipeline are the endpoint, the router, and the application. You can visualize any Phoenix request with this CRC pipeline:

```

connection_from_request
|> endpoint
|> router
|> custom_application

```

Each one of these pieces is made up of tiny functions. The `custom_application` can be a Phoenix controller, a Phoenix channels application, or a live view. We'll spend most of the book on live views. For now, let's take a few short sections discussing the first two parts of this pipeline, the endpoint and router.

The Phoenix Endpoint

If Phoenix is a long chain of reducer functions called plugs, the endpoint is the constructor at the very beginning of that chain. The endpoint is a simple Elixir module in a file called `endpoint.ex`, and it has exactly what you would expect—a pipeline of plugs.

You might not ever change your `endpoint.ex` file, so we won't read through it in detail. Instead, we'll just scan through it to confirm that every Phoenix request goes through an explicit list of functions called plugs. There's no magic.

Open up `endpoint.ex`, and you'll notice that it has a bit of configuration followed by a bunch of plugs. That configuration defines the *socket* that will handle the communication for all of your live views, but the details are not important right now.

After those sockets, you see a list of plugs, and every one of them transforms the connection in some small way. Don't get bogged down in the details. Instead, scan down to the bottom. Eventually, requests flow through to the bottom of the pipeline to reach the router at the bottom:

```
intro/pento/lib/pento_web/endpoint.ex
plug Plug.MethodOverride
plug Plug.Head
plug Plug.Session, @session_options
plug PentoWeb.Router
```

You don't have to know what these plugs do yet. Just know that requests, in the form of `Plug.Conn` connections, flow through the plugs and eventually reach the Router.

The esteemed router is next.

The Phoenix Router

Think of a router as a switchboard operator. Its job is to route the requests to the bits of code that make up your application. Some of those bits of code are common pieces of *policy*. A Policy defines how a given web request should be treated and handled. For example, browser requests may need to deal with cookies; API requests may need to convert to and from JSON, and so on. The router does its job in three parts.

- First, the router specifies chains of common functions to implement policy.
- Next, the router groups together common requests and ties each one to the correct policy.
- Finally, the router maps individual requests onto the modules that do the hard work of building appropriate responses.

Let's see how that works. Open up `lib/pento_web/router.ex`. You'll find more plugs, and some mappings between specific URLs and the code that implements those pages. Each grouping of plugs provides *policy* for one or more routes. Here's how it works.

Pipelines are Policies

A *pipeline* is a grouping of plugs that applies a set of transformations to a given connection. The set of transformations applied by a given plug represents

a policy. Since you know that every plug takes in a connection and returns a connection, you also know that the first plug in a pipeline takes a connection and the last plug in that pipeline returns a connection. So, a plug pipeline works exactly like a single plug! This browser pipeline implements the policy your application needs to process a request from a browser:

```
auth/pento/lib/pento_web/router.ex
pipeline :browser do
  plug :accepts, ["html"]
  plug :fetch_session
  plug :fetch_live_flash
  plug :put_root_layout, {PentoWeb.LayoutView, :root}
  plug :protect_from_forgery
  plug :put_secure_browser_headers
  plug :fetch_current_user
end
```

This bit of code says we're going to accept only HTML requests, and we'll fetch the session, and so on. This api pipeline implements the policy for an API:

```
intro/pento/lib/pento_web/router.ex
pipeline :api do
  plug :accepts, ["json"]
end
```

It has a single plug that means associated routes will accept only JSON⁶ requests.

Now that we know how to build a policy, the last thing we need to do is to tie a particular URL to a policy, and then to the code responsible for responding to the request for the particular URL.

Scopes

A scope block groups together common kinds of requests, possibly with a policy. Here's a set of common routes in a scope block.

```
scope "/", PentoWeb do
  pipe_through :browser
  ... individual routes here...
end
```

This tiny block of code does a lot. The scope expression means the provided block of routes between the do and the end applies to *all routes* because all routes begin with /. The pipe_through :browser statement means every matching

6. <https://www.json.org/json-en.html>

request in this block will go through all of the plugs in the `:browser` pipeline. We'll handle the routes next.

Routes

The last bit of information is the individual routes. Let's list our route one more time for clarity.

```
live "/", PageLive, :index
```

Every route starts with a route type, a URL pattern, a module, and options. LiveView routes have the type `live`.

The URL pattern in a route is a pattern matching statement. The `"/"` pattern will match the url `/`, and a pattern of `"/bears"` will match a URL like `/bears`, and so on.

The next bit of information is the `PageLive` module, which implements the code that responds to the request. The type of route will determine what kind of code does the responding. Since our route is a live route, the `PageLive` module will implement a live view.

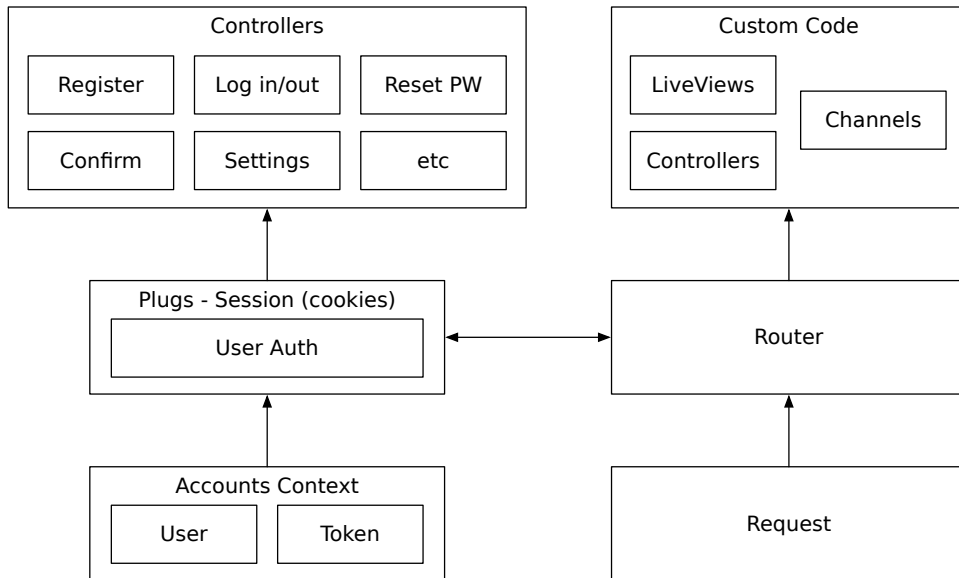
The last option is the `:index` live action. It's just a bit of metadata about the request. As we go, we'll offer more information about routes. For now, let's move on.

Plugs and Authentication

Now we need to think about authentication. Web applications almost always need to know who's logged in. Authentication is the service that answers the question *Who is logging in?*. Only the most basic applications can be secure without authentication, and since malicious actors have worked for decades breaking authentication systems, it's best to use a service built by someone who knows what they are doing.

Our authentication service will let in only those who have accounts on our game server. Since we plan to have pages only our registered users should see, we will need to secure those pages. We must know who is logging in before we can decide whether or not to let them in.

Now, let's put all of that conversation about plugs into action. Let's discuss a plan for authentication. We will build our authentication system in layers, as demonstrated in this figure.



On the left side is the *infrastructure*. This code will use a variety of services to store long-term user data in the database, short-term session data into cookies, and it will provide user interfaces to manage user interactions.

On the right side, the Phoenix router will send appropriate requests through authentication plugs within the router, and these plugs will control access to custom live views, channels, and controllers.

We'll go into each of these layers in detail throughout the rest of the chapter. Suffice to say we're not going to build this service ourselves. Instead, we'll generate it from an existing dependency. Let's get to work!

Generate The Authentication Layer

`phx.gen.auth` is an application generator that builds a well-structured authentication layer for Phoenix requests. This generator is rapidly becoming the authentication standard for all Phoenix applications. Though `phx.gen.auth` is not a LiveView framework, all live views begin as standard web requests. That means a plug-based approach suits our purposes just fine to authenticate our users in LiveView, as well as throughout the rest of our application. Using this generator, we'll be able to generate, rather than hand-roll, a solution that's mostly complete, and adapt it to meet any further requirements we might have.

In the following sections, you'll learn how to use the generator to build an authentication layer, you'll see how the pieces of generated code fit together

to handle the responsibilities of authentication, and you'll even see how LiveView uses the authentication service to identify and operate on the logged in user.

We'll start by installing and running the generator.

Add the Dependency

To use this authentication generator, we need to add `phx_gen_auth` to our application's dependency list. Navigate to the `mix.exs` file and add the `phx_gen_auth` library:

```
auth/pento/mix.exs
{:jason, "~> 1.0"},
{:plug_cowboy, "~> 2.0"},
{:phx_gen_auth, "~> 0.4", only: [:dev], runtime: false},
```

Notice we're using version 0.4 as of this writing. If you'd like, you can use a later version instead. Most of the approaches will still work, though you might have to tweak some of the code here or there. Elixir Mix projects separate the code that we write from the dependencies containing code others write. When we list a dependency, we sometimes provide an exact version to use, and other times we give Mix some flexibility to choose. In this case, we're asking for version 0.4, but we're allowing any minor versions.

The next step is to fetch all of the code for our dependencies, *and the code their mix projects depend on*, into our local project. Fetch the dependencies with `mix deps.get`, and you'll see a result similar to this one:

```
Unchanged:
  connection 1.0.4
  cowboy 2.8.0
  cowlib 2.9.1
...
New:
  phx_gen_auth 0.4.0
```

If you take a peek into the `deps` directory, you'll find a new subdirectory:

```
[pento] → ls deps/phx_gen_auth/
CHANGELOG.md      hex_metadata.config mix.exs
README.md         lib                  priv
```

Nice. Our dependency isn't authentication code itself. It's the code that will *generate* the authentication layers. We'll need to run that generator to get a working authentication layer. Let's let the generator fly.

Run the Generator

Before we generate our code, we'll take advantage of a common pattern for many mix tasks: running them without arguments will give you guidance about any options you'll need. Run `mix phx.gen.auth` without any arguments to trigger a small bit of help, like this:

```
[pento] → mix phx.gen.auth
==> phx_gen_auth
Compiling 6 files (.ex)
Generated phx_gen_auth app
==> pento
** (Mix) Invalid arguments

mix phx.gen.auth expects a context module name, followed by
the schema module and its plural name (used as the schema
table name).
```

For example:

```
mix phx.gen.auth Accounts User users
...

```

Don't worry about the vocabulary. We'll cover contexts, schemas, and the like in more detail later. For now, know that running this generator creates a module called a context and another module called a schema. Look at a context as an API for a service, and a schema as a data structure describing a database table. This generator is giving us the command to build an authentication layer. It would generate a context called `Accounts` and a schema called `User` with a plural of `users`. Check out [Designing Elixir Systems with OTP \[IT19\]](#) for more detail about building software in layers if you are hungry for more.

The generator's defaults seem reasonable, so let's take that advice. Now we can let it fly.

```
[pento] → mix phx.gen.auth Accounts User users
Compiling 14 files (.ex)
Generated pento app
* creating priv/repo/migrations/20200720144910_create_users_auth_tables.exs
* creating lib/pento/accounts/user_notifier.ex
* creating lib/pento/accounts/user.ex
* creating lib/pento/accounts/user_token.ex
...
* injecting lib/pento/accounts.ex
* creating test/pento/accounts_test.exs
* injecting test/pento/accounts_test.exs
* creating test/support/fixtures/accounts_fixtures.ex
```

```
* injecting test/support/fixtures/accounts_fixtures.ex
...
```

The last few instructions on the page are not shown. They tell us to fetch dependencies, and run migrations. It may seem a little strange to have to fetch dependencies after we just added `phx.gen.auth`, but our freshly generated code has its own set of requirements. So, dutifully, we fetch them again:

```
[pento] → mix deps.get
Resolving Hex dependencies...
Dependency resolution completed:
Unchanged:
...
New:
  bcrypt_elixir 2.2.0
  comeonin 5.3.1
  elixir_make 0.6.0
```

You'll notice the generator fetched dependencies to encrypt passwords, along with password hashing libraries. Also, one of these dependencies requires `elixir_make`. We don't need to know why.

Run Migrations

Elixir separates the concepts of working with database *records* from that of working with database *structure*. Our generator gave us the “database structure” code in the form of a set of Ecto migrations for creating database tables. Ecto is the framework for dealing with databases within Elixir, and migrations are the part of Ecto that create and modify database entities. Before your application can work with a database table, your migrations will need to be run to ensure that the database table exists, has the right structure for the data you'll put in it, and has the right set of indexes for performance. Check out the excellent advice in [Programming Ecto \[WM19\]](#) for more details.

Fortunately, along with the rest of the authentication code, `phx.gen.auth` built some migrations for us. We need only run them. Head over to your terminal and execute the migrate command shown here:

```
[pento] → mix ecto.migrate
10:51:27.763 [info] == Running 20200720144910
...CreateUsersAuthTables.change/0 forward
10:51:27.763 [info] execute "CREATE EXTENSION IF NOT EXISTS citext"
10:51:27.851 [info] create table users
10:51:27.857 [info] create index users_email_index
10:51:27.858 [info] create table users_tokens
10:51:27.864 [info] create index users_tokens_user_id_index
10:51:27.868 [info] create index users_tokens_context_token_index
10:51:27.870 [info] == Migrated 20200720144910 in 0.1s
```

Perfect. We made sure the case insensitive extension exists, and we created the tables for users and tokens. Along the way, we created a few indexes for performance as well.

Before we dive in too deeply, let's make sure the overall service is working, end to end. Tests would be a great way to do so.

Test the Service

To make sure everything works, run the tests like this:

```
[pento] (auth *) → mix test
==> connection
Compiling 1 file (.ex)
Generated connection app
...
Compiling 33 files (.ex)
Generated pento app
.....
.....
Finished in 0.7 seconds
101 tests, 0 failures
Randomized with seed 541418
```

Everything works just fine. We're ready to do some code spelunking!

Explore Accounts from IEx

When you're working with big applications with huge available libraries, it pays to have a few tools in your tool box for exploration. Reading code is one technique you can use, and another is looking at the public functions in IEx. To do so you'll use a function called `exports`.

Most experienced programmers strive to separate complex code into layers. We'll have plenty of opportunities to explore these layers from the inside in [Chapter 3, Generators: Contexts and Schemas, on page 57](#). In this chapter, rather than focusing on how the code works, we'll look at the various things that it can *do*—starting in this section with the Accounts context. The generated Accounts *context* is the layer that we will use to create, read, update, and delete users in the database. It provides an API through which all of these database transactions occur.

The Accounts context will handle a few more responsibilities beyond basic CRUD interactions for a user. When a user logs in, we'll need a bit of code that looks up a user. We'll need to store an intermediate representation called a *token* in our database to keep our application secure. We'll also need a way for our

user to securely update their email or password. We'll do all of these things in the Accounts context.

View Public Functions

We can get a good idea of what the Accounts context does by looking at its public functions. Luckily, IEx makes this easy. Open up IEx with `iex -S mix`, alias the context, and get a look at the exports, like this:

```
iex> alias Pento.Accounts
Pento.Accounts
iex> exports Accounts
```

You'll see a ton of functions. We're going to look at them in chunks. The first few functions work with new users. When you expose an application on the web that sends email to users, it's your responsibility to make sure the person on the other end of that email is real, and has actually asked to be included. Confirmation proves a person actually owns the email address they've used to register:

```
...
register_user/1
confirm_user/1
...
```

The `register_user/1` function creates a user and `confirm_user/1` confirms a user. See the [hexdocs documentation](#)⁷ for details about user confirmation.

Moving on to another responsibility of the code generated by the `phx.gen.auth` package—managing the user's session. Session management is handled by adding a tiny token to the session stored in a user's cookie when a user signs in, and deleting it when they sign out. These generated functions create and delete the session token:

```
...
delete_session_token/1
generate_user_session_token/1
...
```

Next up are a few functions that let us look up users in various ways:

```
...
get_user!/1
get_user_by_email/1
get_user_by_email_and_password/2
get_user_by_reset_password_token/1
get_user_by_session_token/1
```

7. https://hexdocs.pm/phx_gen_auth/overview.html#confirmation

...

Sessions will have tokens, so we'll be able to look up a logged in user using those tokens. We'll also be able to find our user by email and password when a user logs in, and so on.

In addition, our context provides a few functions for changing users. Here are the most important ones:

```
...
reset_user_password/2
update_user_password/3
update_user_email/2
...
```

We can start the password reset process if a user forgets their password, updates a password, or updates an email.

These functions make up the bulk of the Accounts API. The remaining functions let us validate new and existing users, integrate custom email services, and the like. We have what we need to continue our exploration. Let's put the Accounts API through its paces.

Create a Valid User

While the console is open, let's create a user. All we need are the parameters for a user, like this:

```
iex> params = %{email: "mercutio@grox.io", password: "R0sesBy0therNames"}
%{email: "mercutio@grox.io", password: "R0sesBy0therNames"}
iex> Accounts.register_user(params)
[debug] QUERY OK ...
SELECT TRUE FROM "users" AS u0 WHERE (u0."email" = $1) LIMIT 1 ...
[debug] QUERY OK db=3.0ms
INSERT INTO "users" ("email","hashed_password","inserted_at","updated_at") ...
[debug] QUERY OK db=4.2ms
commit []
{:ok,
 #Pento.Accounts.User<
  __meta__: #Ecto.Schema.Metadata<:loaded, "users">,
  confirmed_at: nil,
  email: "mercutio@grox.io",
  hashed_password: "$2b$12$jv...",
  id: 1,
  inserted_at: ~N[2020-07-20 20:12:32],
  updated_at: ~N[2020-07-20 20:12:32],
  ...
>}
```

Under the hood, the Accounts context created a changeset, and seeing valid data, it inserted an account record into the database. Notice the result is an `{:ok, user}` tuple, so Mercutio rides!

Try to Create an Invalid User

Now, let's try to create an invalid user. If the parameters are invalid, we'd get an error tuple instead:

```
iex> Accounts.register_user(%{})
{:error,
 #Ecto.Changeset<
  action: :insert,
  changes: %{},
  errors: [
    password: {"can't be blank", [validation: :required]},
    email: {"can't be blank", [validation: :required]}
  ],
  data: #Pento.Accounts.User<>,
  valid?: false
 >}
```

Since the operation might fail, we return a result tuple. We'll get `{:ok, user}` on success and `{:error, changeset}` upon error. You'll learn later that a changeset represents change. Invalid changesets say why they are invalid with a list of errors. Don't get bogged down in the details. We'll go more in depth later.

Now that you've seen how our new context works, let's move on to the code that will let web requests in or keep them out. That happens in the router. We'll look at the authentication service and you'll see how it uses plugs that call on Accounts context functions to manage sessions and cookies.

Protect Routes with Plugs

The authentication service integrates with the Phoenix stack to provide infrastructure for session management including plugs that we can use in the router to control access to our routes.

The authentication service is defined in the file `lib/pento_web/controllers/user_auth.ex`. We could open up the code base, but instead, let's do a quick review in IEx to see what the public API looks like.

If IEx isn't opened, fire it up with `iex -S mix`, and key this in:

```
iex> exports PentoWeb.UserAuth
fetch_current_user/2
log_in_user/2
log_in_user/3
```

```
log_out_user/1
redirect_if_user_is_authenticated/2
require_authenticated_user/2
```

All of these functions are plugs. The first fetches an authenticated user and adds it into the connection. The next three log users in and out. The last two plugs direct users between pages based on whether they are logged in or not. Let's first examine `fetch_current_user/2`.

Fetch the Current User

Remember, plugs are reducers that take a `Plug.Conn` as the first argument and return a transformed `Plug.Conn`. Most of the plugs we'll use are from the `UserAuth` module. `fetch_current_user/2` will add the current user to our `Plug.Conn` if the user is authenticated. You don't have to take this on faith. Though you might not understand all of the code, you already know enough to get the overall gist of what's happening. Let's take a closer look.

Most of `Plug.Conn` contains private data we can't change, but application developers have a key pointing to a dedicated map called `assigns` that we *can* use to store custom application data.

The `fetch_current_user/2` function plug will add a key in `assigns` called `current_user` if the user is logged in. You can see that the code generator added this plug to our browser pipeline in the router, like this:

```
auth/pento/lib/pento_web/router.ex
pipeline :browser do
  plug :accepts, ["html"]
  plug :fetch_session
  plug :fetch_live_flash
  plug :put_root_layout, {PentoWeb.LayoutView, :root}
  plug :protect_from_forgery
  plug :put_secure_browser_headers
  plug :fetch_current_user
end
```

Now, whenever a user logs in, any code that handles routes tied to the browser pipeline will have access to the `current_user` in `conn.assigns.current_user`.

You may not know it yet, but our pento web app is already taking advantage of this feature. Open up `lib/pento_web/templates/layout/_user_menu.html.eex`:

```
<ul>
<%= if @current_user do %>
  <li><%= @current_user.email %></li>
  ...
  <li><%= link "Log out", to: Routes.user_session_path(@conn, :delete),
    method: :delete %></li>
```



```
<% else %>
...
<% end %>
</ul>
```

The new layout's user menu uses the `current_user`, stored in the connection's assigns and accessed in the template via `@current_user`, to print the email for the logged-in user. We know the `current_user` will be present if they are logged in.

Authenticate a User

Remember, Phoenix works by chaining together plugs that manipulate a session. The `log_in_user/3` function is no exception. Let's check out the details for logging in a user, like this:

```
iex> h PentoWeb.UserAuth.log_in_user

      def log_in_user(conn, user, params \\ %{})
```

Logs the user in.

It renews the session ID and clears the whole session to avoid fixation attacks. See the `renew_session` function to customize this behaviour.

It also sets a `:live_socket_id` key in the session, so LiveView sessions are identified and automatically disconnected on log out. The line can be safely removed if you are not using LiveView.

Notice that the function also sets up a unique identifier for our LiveView sessions. That ID will come in handy later. We can expect to see this function called within the code that logs in a user. In fact, that code is within the `lib/pento_web/controllers/user_session_controller`:

```
auth/pento/lib/pento_web/controllers/user_session_controller.ex
def create(conn, %{ "user" => user_params }) do
  %{ "email" => email, "password" => password } = user_params

  if user = Accounts.get_user_by_email_and_password(email, password) do
    UserAuth.log_in_user(conn, user, user_params)
  else
    render(conn, "new.html", error_message: "Invalid e-mail or password")
  end
end
```

Short and sweet. We pluck the email and password from the inbound params sent by the login form. Then, we use the context to check to see whether the user exists and has provided a valid password. If not, we render the login page again with an error. If so, we'll execute the `log_in_user/3` function implement by the `UserAuth` module, passing our connection:

```
auth/pento/lib/pento_web/controllers/user_auth.ex
def log_in_user(conn, user, params \\ %{}) do
```

```

token = Accounts.generate_user_session_token(user)
user_return_to = get_session(conn, :user_return_to)

conn
|> renew_session()
|> put_session(:user_token, token)
|> put_session(:live_socket_id, "users_sessions:#{Base.url_encode64(token)}")
|> maybe_write_remember_me_cookie(token, params)
|> redirect(to: user_return_to || signed_in_path(conn))
end

```

We build a token and grab our redirect path from the session. Then, we renew the session for security's sake, adding both the token and a unique identifier to the session. Next, we create a `remember_me` cookie if the user has selected that option, and finally redirect the user. This beautiful code practically weaves a plain English narrative for us. Later, you'll learn how to use this token to identify the authenticated user in a live view.

With those out of the way, let's look at the plugs that will let us use all of the infrastructure we've generated. We're ready to tweak our router just a bit to make sure users are logged in. With this, we'll have put together all of the pieces of the generated authentication code.

Authenticate The Live View

Let's integrate our toy `wrong_live` view with the authentication infrastructure. This quick test will let us make sure our infrastructure is working. When we're done, you'll understand how to protect authenticated LiveView routes and you'll know how to identify the authenticated user in a live view.

We'll start in the router by putting our live route behind authentication. Then, we'll update our live view's `mount/3` function to use the token from the session to find the logged in user.

Protect Sensitive Routes

When we ran the generator earlier, a scope was added to our router containing the set of routes that require a logged-in user. The scope pipes requests to such routes through two pipelines—the browser pipeline, which establishes the policy for web requests from browsers, and the generated `UserAuth.require_authenticated_user/2` function plug, which ensures that a current user is present, or else redirects to the sign in page.

In order to authenticate our `wrong_live` view, we'll move the live view route into this pipeline.

```
auth/pento/lib/pento_web/router.ex
```

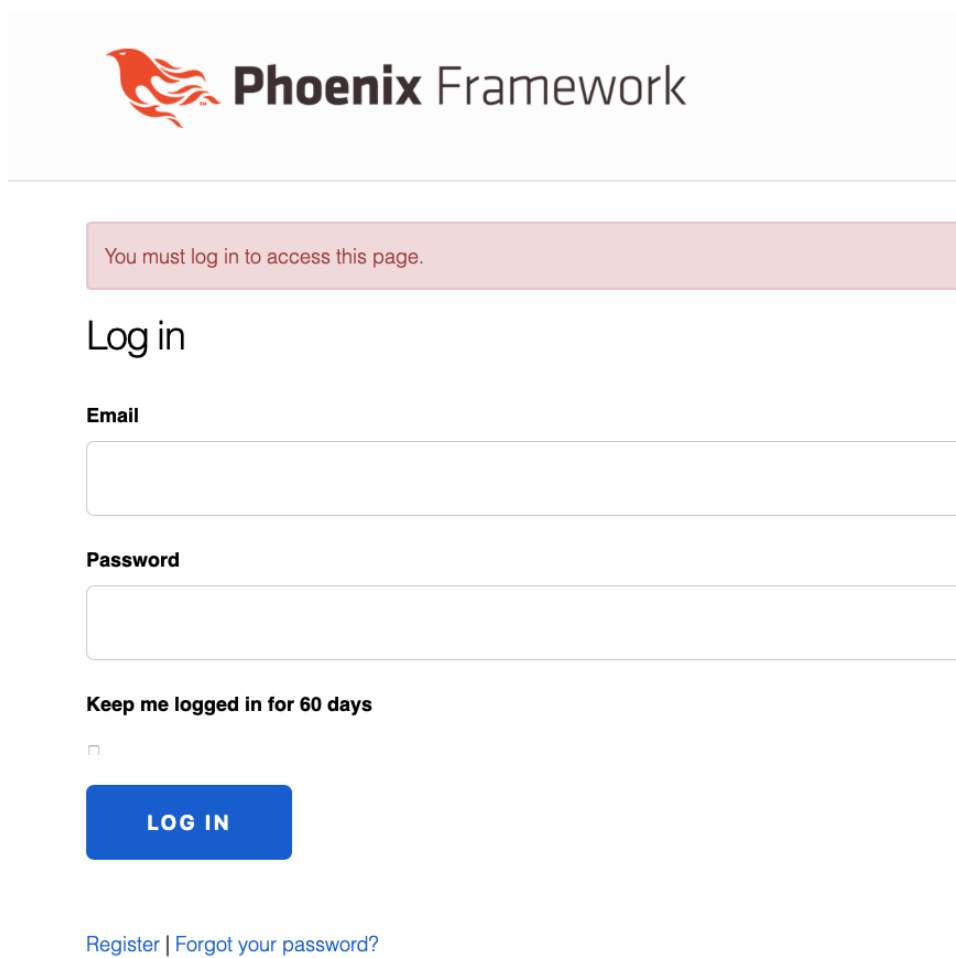
```
scope "/", PentoWeb do
  pipe_through [:browser, :require_authenticated_user]

  live "/guess", WrongLive
```

Believe it or not, that's all we have to do to restrict our route to logged in users! Let's take it for a spin.

Test Drive the LiveView

Start up your web server with `mix phx.server`, and point your browser to `localhost:4000/guess` to see the following image:



The image shows a web page for the Phoenix Framework. At the top, there is a logo consisting of a red phoenix icon and the text "Phoenix Framework". Below the logo, there is a light red rectangular box containing the text "You must log in to access this page." Underneath this box, the text "Log in" is displayed in a large, black, sans-serif font. Below "Log in", there are two input fields. The first is labeled "Email" and the second is labeled "Password". Both labels are in a bold, black, sans-serif font. Below the password field, there is a checkbox and the text "Keep me logged in for 60 days". At the bottom of the form, there is a blue rectangular button with the text "LOG IN" in white, uppercase, sans-serif font. Below the button, there are two links: "Register" and "Forgot your password?", both in a blue, sans-serif font.

The plug fires, and redirects you to the login page. You can click register:



Phoenix Framework

[Get Started](#)
[LiveDashboard](#)
[Register](#)
[Log in](#)

Register

Email

bruce@example.com

Password

.....



REGISTER

[Log in](#) | [Forgot your password?](#)

And once registered, you're logged in!



Phoenix Framework

[Get Started](#)
[LiveDashboard](#)
 bruce@example.com
[Settings](#)
[Log out](#)

Your score: 0

Guess a number.

12345678910

The logged in user appears in the title bar.

Before we wrap up, let's do a tiny bit of exploration of how LiveView uses the authentication service to identify the signed-in user. Let's teach our live view to grab the user and live view ID from the session.

Access Session Data

Our session has both a token and a live view socket id, and the session is made available to the live view as the second argument given to the mount/3 function. From there, it's a small matter of using a function provided by our Accounts context to find the user who belongs to the token.

```
auth/pento/lib/pento_web/live/wrong_live.ex
def mount(_params, session, socket) do
  {
    :ok,
    assign(
      socket,
      score: 0,
      message: "Guess a number.",
      user: Pento.Accounts.get_user_by_session_token(session["user_token"]),
      session_id: session["live_socket_id"]
    )
  }
end
```

Here, we add two more keys to the socket.assigns. To set the :session_id key, we copy the session ID directly. Then, we use Accounts.get_user_by_session_token/1 to set the :user key. To make sure things are working, let's just render these assignments. We can do so by accessing the values of the @user and @session_id assignments in the markup returned by the render/1 function:

```
auth/pento/lib/pento_web/live/wrong_live.ex
def render(assigns) do
  ~L"""
  <h1>Your score: <%= @score %></h1>
  <h2>
    <%= @message %>
  </h2>
  <h2>
    <%= for n <- 1..10 do %>
      <a href="#" phx-click="guess" phx-value-number="<%= n %>"><%= n %></a>
    <% end %>
  </h2>
  <pre>
    <%= @user.email %>
    <%= @session_id %>
  </pre>
  """
end
```

Now, if you refresh the page at /guess, you'll see a few extra items:

bruce@example.com

```
users_sessions:qDiTcmf1o0V22eYYLr1VojmpFm0Lgtz-5ffzniGlc4=
```

The extra information slides into place, just like we planned it! We demonstrated a nice start to an authentication service, and you can see how LiveView integrates with that service.

It's been a long and intense chapter, so it's time to wrap up.

Your Turn

Rather than using libraries for authentication, a good strategy is to generate your code with the `phx.gen.auth` code generator. The code that this generator creates checks all of the must-have boxes for an authentication service, especially satisfying the OWASP standards, and saves us the tedious work of building out authentication ourselves. When you're building your own Phoenix LiveView apps in the wild, you'll reach for this generator to quickly add a tried and tested authentication solution to your web app.

Once you install and run the generator, you'll be able to maintain the code as if it were your own. The code comes with a context for long term persistence of users, passwords, and session tokens, and a short-term solution for adding authenticated tokens representing users to a session. There are controllers to handle registration, logging in, confirming users, and resetting passwords, as well as plugs that you will use in your router to apply authentication policies to certain routes.

You saw exactly how Phoenix uses plugs to respond to web requests by constructing pipelines of small functions, each of which applies some transformation to a common connection struct. Later, you'll see that this is the same pattern individual live views will use to respond to both initial web requests and user interactions with a live view page.

With all of this under your belt, it's time to put what you've learned into practice.

Give It a Try

These problems deal with small tweaks to the existing generated code.

- If you already have an email service, try plugging it in to the generated authentication service so that it will really send the user an email when they register for an account. Did you have to add additional arguments to the existing functions?

- Add a migration and a field to give the User schema a username field, and display that username instead of the email address when a user logs in. Did you require the username to be unique?
- If a logged in user visits the / route, make them redirect to the /guess route.

Next Time

After a long chapter of Phoenix configuration, you may want a break from the detailed concepts. With the authentication chapter behind us, we're ready to play. In the next chapter, we're going to start building out the specific functionality of our application. We'll begin with a product management system—we want to be able to persist a list of products, and provide simple admin pages to maintain them. Let's keep it rolling!

Generators: Contexts and Schemas

So far, we've focused our efforts on briefly building some intuition for how Phoenix LiveView works, and building an authentication layer. While we did create our own custom live view in order to explore LiveView forms, we haven't yet written any serious LiveView code. In this chapter, that changes.

The next two chapters will build a product catalog into our application. Rather than write the code by hand, we'll use the Phoenix generators to build the bulk of what we need.

You might wonder why we're planning to generate code in a book dedicated to teaching you to write your own LiveView code. We do so because Phoenix's Live generator is a powerful tool that will increase your productivity as a LiveView developer. With just one command, you can generate a full CRUD feature for a given resource, with all of the seamless real-time interactions that LiveView provides. You will reach for the Phoenix Live generator whenever you need to build a basic CRUD feature, saving yourself the time and effort of implementing this common functionality. Beyond that, the generated code provides a strong, organized foundation on which to build additional features when you do need to go beyond CRUD.

The Phoenix Live generator is just one more way that Phoenix empowers developers to be highly productive, while bringing the real-time capabilities of LiveView to the table to meet the increasingly interactive demands of the modern web. While you won't use the Phoenix Live generator every time you build a LiveView application, you will reach for it when building common, foundational web app functionality. This helps you cut down on coding time, making it a valuable tool in your toolbox.

Let's make a brief plan. First, we'll run the generator. Some of the code we generate will be backend database code, and some will be frontend code. In

this chapter, we'll focus on the backend code, and in the next chapter, we'll take a deep dive into the generated frontend code. The Phoenix generators will separate backend code into two layers. The schema layer describes the Elixir entities that map to our individual database tables. It provides functions for interacting with those database tables. The API layer, called a context, provides the interface through which we will interact with the schema, and therefore the database.

The generated code was built and shaped by experts, and we believe it reflects one of the best ways to build LiveView applications. In these two chapters, we'll trace through the execution of our generated code and show you why it represents the best way to build and organize LiveView. When you're done, you'll know how to leverage the powerful generator tool to create full-fledged CRUD features, you'll have a strong understanding of how that generated code ties together, and you'll start to appreciate the best practices for organizing LiveView code.

These two chapters will be demanding, but fun. It's time to get to work.

Get to Know the Phoenix Live Generator

The Phoenix Live generator is a utility that generates code supporting full CRUD functionality for a given resource. This includes the backend schema and context code, as well as the frontend code including routes, LiveView, and templates. Before we dive into using the generator, it's worth discussing just what's so great about this generated code in the first place. In order to do that, we'll address the elephant in the room—the not-uncommon skepticism of code generators.

Let's be honest. Code generators have a checkered past. The potential land mines are many. In some environments, generated code is so difficult to understand that application developers can't make reliable changes. In others, generated code does not follow the best practices for a given ecosystem, or is too simplistic to serve as a meaningful foundation for custom, non-generated code.

Furthermore, in Elixir, code generation is not as essential to our productivity as it is in some other languages. This is in part because Elixir supports macros. Since macros are code that writes code, macros often replace code that must be generated in other environments. In fact, we'll see that our generated code will take advantage of macros to pull in key pieces of both backend and frontend functionality.

Code generators are still critical in one area however: the creation of generic application code. No macro can satisfy the needs of a generic application, so sometimes the best approach is to generate the tedious, simple code as a foundation. Then, the developer can rely on that foundation to build the rest of their application.

Foundations only work if they are right, and the Phoenix team worked hard to make sure the abstractions within the generated code are right, and that the encapsulated ideas are accessible. The whole Phoenix team placed serious emphasis on refactoring the generated code, bit by bit, until it was right.

So, the Phoenix Live generator provides us with a quick and easy way to build CRUD features, taking over the often tedious and repetitive work of building out this common functionality. It does so in a way that is adherent to best-practices for organizing Phoenix code in general, and LiveView code specifically, making it easy for developers to build on top of, and customize, the generated code. The Phoenix Live generator is just one of many reasons why Phoenix and LiveView developers can be so highly productive.

Now that you understand what the Phoenix Live generator is and what it does for you at a high level, we're ready to use it.

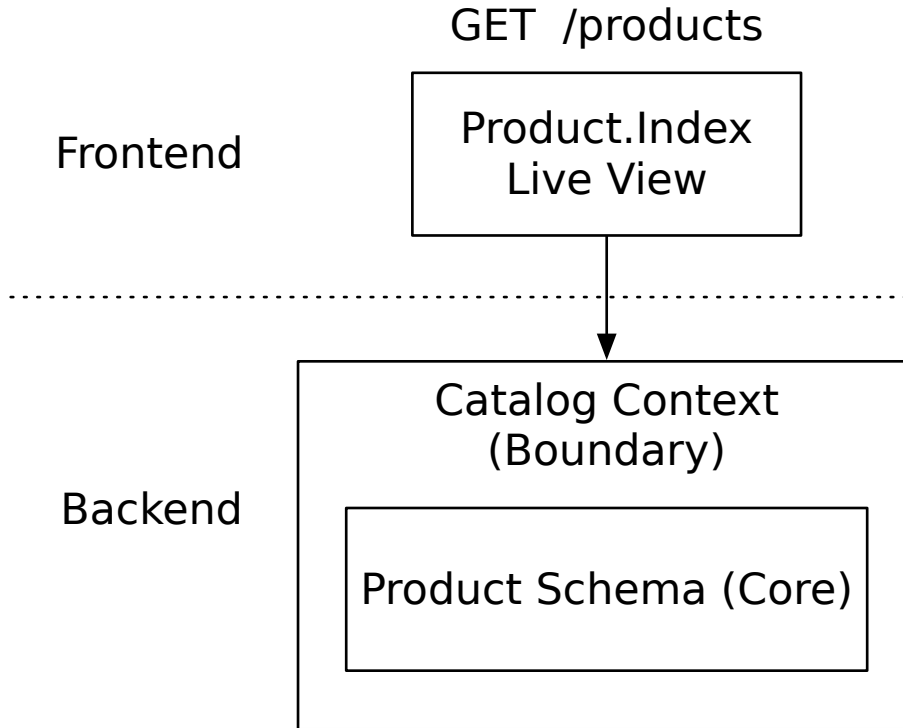
Let's get started.

Run the phx.live Generator

We'll use the Phoenix Live generator to create a feature for managing products. If you are familiar with web development with Phoenix or even other languages, you know many web libraries and frameworks have a concept called a *resource*. In common terms, a resource is a collection of like entities. Product will be our resource.

Running the generator will give us all of the code needed to support the CRUD interactions for this resource. The generated frontend code, including the live views, will reside in `lib/pento_web`. Backend code, on the other hand, will live in `lib/pento`. It will deal with database interactions via the schema and provide an API through which to manage those interactions, called the context.

When we're done, we'll have a schema for a Product, a Catalog context, along with live views for managing a product. As this figure demonstrates, all of these pieces of generated code will work together to make up the CRUD interactions for the Product resource.



At a high level, you can see that an HTTP request, one for the `/products` route for example, will be routed to and handled by a live view. These are the frontend concerns. The live view will in turn rely on the context, which wraps the schema, to interact with product records in the database. Together, the context and schema make up the backend concerns. We'll learn more about the context and schema and how they work in the following sections.

Let's fire up the generator!

Learn How To Use the Generator

When we generate the code for our resource, we'll need to specify both a context and a schema. We'll also need to tell Phoenix which fields to support for the resource's corresponding database table. In order to learn exactly how to structure the generator command, we'll use the generator tool's "help" functionality. As with most Elixir tooling, the documentation and help is excellent.

The first way to get help for a tool is to use it without required options. Run the generator without options, like this:

```
$ mix phx.gen.live
```

...compiling...

** (Mix) Invalid arguments

`mix phx.gen.html`, `mix phx.gen.json`, `mix phx.gen.live`, and `mix phx.gen.context` expect a context module name, followed by singular and plural names of the generated resource, ending with any number of attributes.

For example:

```
mix phx.gen.html Accounts User users name:string
mix phx.gen.json Accounts User users name:string
mix phx.gen.live Accounts User users name:string
mix phx.gen.context Accounts User users name:string
```

The context serves as the API boundary for the given resource.

Multiple resources may belong to a context and a resource may be split over distinct contexts (such as `Accounts.User` and `Payments.User`).

The command to run the Phoenix Live generator is `mix phx.gen.live`. Since we executed the command without any options, it provides some help for us. Specifically, it offers us some examples of how to use Phoenix generators more generally. The third example down on the indented list of examples illustrates how to use the `mix phx.gen.live` command in order to generate a hypothetical `Accounts` context and `User` schema. Let's dig into this example a bit so that we can understand how to structure our own generator command for the `Product` resource.

Here's the example from the help output:

```
mix phx.gen.live Accounts User users name:string
```

The first argument given to `mix phx.gen.live` is the *context*—here called `Accounts`. The second argument, `User`, is the name of the *resource* and *schema*, while the attributes that follow are the names and types of the fields our schema will support. The generator will take these arguments and use it to generate an `Accounts` context and a `User` schema that maps the provided fields to database columns. Let's use the guidance provided by this example to write our own generator command for the `Product` resource now.

Generate a Resource

Run the generator again, this time filling in the blanks for the context, resource, and fields.

We'll construct the generator command such that it will generate a `Catalog` context with a schema for `Product`, corresponding to a `products` database table. A product will have `name`, `description`, `unit_price`, and `SKU` fields, like this:

```
[pento] → mix phx.gen.live Catalog Product products name:string \
description:string unit_price:float sku:integer:unique
```

```
* creating lib/pento_web/live/product_live/show.ex
* creating lib/pento_web/live/product_live/index.ex
...
```

Add the live routes to your browser scope in

lib/pento_web/router.ex:

```
live "/products", ProductLive.Index, :index
live "/products/new", ProductLive.Index, :new
live "/products/:id/edit", ProductLive.Index, :edit

live "/products/:id", ProductLive.Show, :show
live "/products/:id/show/edit", ProductLive.Show, :edit
```

Phoenix generated a bunch of files, and left some instructions for us. Let's copy-paste these routes into our router within the "/" scope, like this:

`generators/pento/lib/pento_web/router.ex`

```
scope "/", PentoWeb do
  pipe_through [:browser, :require_authenticated_user]
  # Generated product routes start here:
  live "/products", ProductLive.Index, :index
  live "/products/new", ProductLive.Index, :new
  live "/products/:id/edit", ProductLive.Index, :edit

  live "/products/:id", ProductLive.Show, :show
  live "/products/:id/show/edit", ProductLive.Show, :edit
```

Notice that we've added our routes to the browser scope that pipes requests through the `:require_authenticated_user` plug. This will ensure that only logged-in users can see the products pages. This will become important later on in this book.

As you saw in [Chapter 1, Get To Know LiveView, on page 1](#), for live views, these routes tie URL patterns to the module that implements them. Let's look at one of these routes in more detail.

```
live "/products/new", ProductLive.Index, :new
```

The `live` macro instructs Phoenix that this request will start a live view. The `ProductLive.Index` argument is the module that implements the live view. The `:new` argument is the *live action*. As you'll see later, Phoenix will put the `:new` live action into the socket when it starts the live view. We'll take a closer look at this macro in the next chapter.

Now it's time to shift our attention to the backend—the context and schema. Let's look at the backend code the generator created, and how that code works together to support the CRUD features for products.

Understand The Generated Core

In [Designing Elixir Systems with OTP \[IT19\]](#), we separate the concerns for each resource into two layers, the *boundary* and the *core*. Our generated backend code is also separated in this way. The Catalog context represents the boundary layer, it is the API through which external input can make its way into the application.

The Product schema, on the other hand, represents the application's core. The generated migrations are also part of the core. The core is the home of code that is certain and predictable—code that will always behave the same way given the same inputs. The core is responsible for managing and interacting with the database. You'll use code in the core to create and maintain database tables, and prepare database transactions and queries. Later, you'll see how LiveView uses some of this code, through the API provided by the context, to manage product records. Before we get to that though, it's important for you to understand how the core handles these responsibilities and how the context and core work together to expose an API for database interactions to the rest of the application.

Context vs. Boundary



A *Phoenix Context* is a module in your Phoenix application that provides an API for a service or resource. It is responsible for managing uncertainty, external interfaces, and process machinery. The context implements the *boundary layer* of your application. In this book, we'll refer to the context to denote such a module, and the boundary to describe the role that a context plays in your application's architecture.

Let's walk through the generated core code—the migration file and the Product schema. Then, we'll take a deep dive into the Catalog context.

The Product Migration

We don't need to understand the whole user interface before we put our backend code to work, but we do need to tweak the database so it supports our new products table. Fortunately, the generator created a migration file for us to do exactly that.

Open up the migration in a file that looks something like `pento/priv/repo/migrations/20200910122000_create_product.exs`. The filename, including the timestamp, was generated for us when we ran our generator command, so yours won't match exactly because the timestamp was built into the file name.

The migration file defines a database table, `products`, along with a set of fields for that table. The generator took the table name and the field name and type specifications from the generator command and used them to inform the content of this file.

```
generators/pento/priv/repo/migrations/20200910122000_create_products.exs
defmodule Pento.Repo.Migrations.CreateProducts do
  use Ecto.Migration

  def change do
    create table(:products) do
      add :name, :string
      add :description, :string
      add :unit_price, :float
      add :sku, :integer

      timestamps()
    end

    create unique_index(:products, [:sku])
  end
end
```

Migration files allow us to build key changes to the database into code. Executing the files makes these changes to your database. Since these files need to be executed in a specific order, the filename should begin with a timestamp. You can, and likely will, build your own custom migration files, and/or customize generated migration files. Luckily for us however, the migration file that the generator command built already has exactly what we need to create the `products` table. All we need to do is execute the file.

Run the migration now by opening up your terminal and firing off the Mix command:

```
[pento] → mix ecto.migrate
08:20:32.843 [info] == Running 20200910122000
  Pento.Repo.Migrations.CreateProducts.change/0 forward
08:20:32.843 [info] create table products
08:20:32.846 [info] == Migrated 20200910122000 in 0.0s
```

Notice the `[info]` messages. As we expected, running the migration via `mix ecto.migrate` created the `products` database table.

Now that we have a shiny new table, it's time to turn our attention to the schema.

The Product Schema

Think of schemas as maps between two kinds of data. On the database side is the products table we generated with our migration. On the Elixir side, the Product schema knows how to translate between the products database table and the `Pento.Catalog.Product` Elixir struct. We don't have to write all of that translation code. Ecto will do that for us in the Product schema module.

The generator created that module and placed it in the `lib/pento/catalog/product.ex` file. Crack it open now.

```
generators/pento/lib/pento/catalog/product.ex
defmodule Pento.Catalog.Product do
  use Ecto.Schema
  import Ecto.Changeset

  schema "products" do
    field :description, :string
    field :name, :string
    field :sku, :integer
    field :unit_price, :float
    timestamps()
  end
end
```

Notice the `use Ecto.Schema` expression. The `use` macro injects code from the specified module into the current module. Here, the generated code is giving the Product schema access to the functionality implemented in the `Ecto.Schema` module. This includes access to the `schema/1` function.

The `schema/1` function creates an Elixir struct that weaves in fields from a database table. The generator knew what fields to specify here based on the field name and types that we gave the `mix phx.gen.live` command. The `timestamps` function means our code will also have `:inserted_at` and `:updated_at` timestamps.

We'll begin by examining the public API of our Product schema with the help of the `exports` function in `IEx`, like this:

```
iex> alias Pento.Catalog.Product
iex> exports Product
__changeset__/0    __schema__/1    __schema__/2    __struct__/0
__struct__/1      changeset/2
```

When you look at the public functions with `exports Product`, you can see the `__struct__` function. We didn't create that struct, but our schema macro did. You also see a few other functions Ecto created for us. We'll use structs to represent database rows in Elixir form.

Let's take a closer look at the Product struct:

```
iex> Product.__struct__
```



```
%Pento.Catalog.Product{
  __meta__: #Ecto.Schema.Metadata<:built, "products">,
  description: nil,
  id: nil,
  inserted_at: nil,
  name: nil,
  sku: nil,
  unit_price: nil,
  updated_at: nil
}
```

You can see that the `Product` struct contains all of the fields defined by the schema function, including the `:updated_at` and `:inserted_at` fields implemented by the use of the `timestamps()` function.

Let's use `__struct__/1` to create a new `Product` struct in IEx:

```
iex> Product.__struct__(name: "Exploding Ninja Cows")
%Pento.Catalog.Product{
  __meta__: #Ecto.Schema.Metadata<:built, "products">,
  description: nil,
  id: nil,
  inserted_at: nil,
  name: "Exploding Ninja Cows",
  sku: nil,
  unit_price: nil,
  updated_at: nil
}
```

The schema macro is not the only aspect of the `Product` module that helps us interact with the `products` database table. The `Product` schema has a function that we can use to validate unsafe input before we include it in a struct. Let's look at that next.

Changesets

Maintaining database integrity is the sacred duty of every application developer, according to the rules of our business. To keep data correct, we'll need to check every piece of data that our application creates or updates. Rules for data integrity together form change *policies* that need to be implemented in code.

Schemas are not limited to a single change policy. For example, admins may be able to make changes that other users can't, while users may not be able to change their email addresses without validation. In `Ecto`, *changesets* allow us to implement any number of change *policies*. The `Product` schema has access to `Ecto`'s changeset functionality, thanks to the call to import `Ecto.Changeset` in

the `Pento.Catalog.Product` module. The `import` function allows us to use the imported module's functions without using the fully qualified name.

Here's what our changeset looks like.

```
generators/pento/lib/pento/catalog/product.ex
def changeset(product, attrs) do
  product
  |> cast(attrs, [:name, :description, :unit_price, :sku])
  |> validate_required([:name, :description, :unit_price, :sku])
  |> unique_constraint(:sku)
end
```

This changeset implements the change policy for new records and updates alike. The piped syntax tells a beautiful story. The pipeline starts with the `Product` struct we want to change. The `Ecto.Changeset.cast/4` function filters the user data we pass into params. Our changeset allows the `:name`, `:description`, `:unit_price`, and `:sku` fields. Other fields are rejected.

The `cast/4` function also takes input data, usually as maps with atom keys and string values, and transforms them into the right types.

The next part of our change policy is to validate the data according to the rules. `Ecto` supports a long list of validations.¹ Our changeset requires all of our attributes to be present, and the `sku` to be unique

The result of our changeset function is a changeset struct. We'll try to interact with our database with changesets to keep both our database and our database administrators happy.

Test Drive the Schema

Now that we've run our migration and taken a closer look at the `Product` schema, let's open up `IEx` and see what we can do with our changeset.

First, initialize an empty `Product` struct:

```
iex> alias Pento.Catalog.Product
Pento.Catalog.Product

iex> product = %Product{}
%Pento.Catalog.Product{
  __meta__: #Ecto.Schema.Metadata<:built, "products">,
  description: nil,
  id: nil,
  inserted_at: nil,
  name: nil,
  sku: nil,
```

1. <https://hexdocs.pm/ecto/Ecto.Changeset.html#module-validations-and-constraints>

```

    unit_price: nil,
    updated_at: nil
  }

```

Now, establish a map of valid Product attributes:

```

iex> attrs = %{
  name: "Pentominoes",
  sku: 123456,
  unit_price: 5.00,
  description: "A super fun game!"
}
%{
  description: "A super fun game!",
  name: "Pentominoes",
  sku: 123456,
  unit_price: 5.0
}

```

Next, execute the `Product.changeset/2` function to create a valid Product changeset:

```

iex> Product.changeset(product, attrs)
#Ecto.Changeset<
  action: nil,
  changes: %{
    description: "A super fun game!",
    name: "Pentominoes",
    sku: 123456,
    unit_price: 5.0
  },
  errors: [],
  data: #Pento.Catalog.Product<>,
  valid?: true
>

```

We can take this valid changeset and insert it into our database with a call to the `Pento.Repo.insert/2` function:

```

iex> alias Pento.Repo
Pento.Repo
iex> Product.changeset(product, attrs) |> Repo.insert()
[debug] QUERY OK db=8.6ms decode=1.8ms queue=4.6ms idle=1783.9ms
INSERT INTO "product" ("description","name","sku","unit_price",
"inserted_at","updated_at") VALUES ($1,$2,$3,$4,$5,$6) RETURNING "id"
["A super fun game!", "Pentominoes", 123456, 5.0, ~N[2020-09-10 13:19:17],
~N[2020-09-10 13:19:17]]
{:ok,
 %Pento.Catalog.Product{
   __meta__: #Ecto.Schema.Metadata<:loaded, "product">,
   description: "A super fun game!",
   id: 1,
   inserted_at: ~N[2020-09-10 13:19:17],

```

```

    name: "Pentominoes",
    sku: 123456,
    unit_price: 5.0,
    updated_at: ~N[2020-09-10 13:19:17]
  }}

```

What happens if we create a changeset with a map of attributes that will not pass our validations? Let's find out:

```

iex> invalid_attrs = %{name: "Not a valid game"}
%{name: "Not a valid game"}
iex> Product.changeset(product, invalid_attrs)
#Ecto.Changeset<
  action: nil,
  changes: %{name: "Not a valid game"},
  errors: [
    description: {"can't be blank", [validation: :required]},
    unit_price: {"can't be blank", [validation: :required]},
    sku: {"can't be blank", [validation: :required]}
  ],
  data: #Pento.Catalog.Product<>,
  valid?: false
>

```

Nice! Our changeset has an attribute of `valid?: false`, and an `:errors` key that describes the problem in a generic way we can present to users. Later, Ecto will use the `valid?` flag to keep bad data out of our database, and Phoenix forms will use the error messages to present validation errors to the user.

Our generated schema already does so much for us, but we can build on it to customize our changeset validations. Let's add an additional validation to the changeset to validate that a product's price is greater than 0.

Add a new validation rule within `lib/pento/catalog/product.ex`, like this:

```

def changeset(product, attrs) do
  ...
  |> validate_number(:unit_price, greater_than: 0.0)
end

```

Now, let's see what happens when we create a changeset with an attribute map that contains an invalid `:unit_price`:

```

iex> recompile()
iex> invalid_price_attrs = %{
  name: "Pentominoes",
  sku: 123456,
  unit_price: 0.00,
  description: "A super fun game!"}
%{
  description: "A super fun game!",

```

```

    name: "Pentominoes",
    sku: 123456,
    unit_price: 0.0
  }
iex> Product.changeset(product, invalid_price_attrs)
#Ecto.Changeset<
  action: nil,
  changes: %{
    description: "A super fun game!",
    name: "Pentominoes",
    sku: 123456,
    unit_price: 0.0
  },
  errors: [
    unit_price: {"must be greater than %{number}",
      [validation: :number, kind: :greater_than, number: 0.0]}
  ],
  data: #Pento.Catalog.Product<>,
  valid?: false
>

```

Perfect! Our changeset's `valid?` flag is false, and the errors list describes the `unit_price` error.

Our application code won't work on the `Pento.Catalog.Product` schema directly. Instead, any interactions with it will be through the API, our `Catalog` context. This structure lets us protect the functional core, making sure that data is valid and correct *before* it gets to our database.

Now that we have a working schema, let's put it through the paces using the `Catalog` context.

Understand The Generated Boundary

We've spent a little time in the functional core, the land of certainty and beauty. We've seen that our schema and changeset code is predictable and certain—it behaves the same way given the same inputs, every time. In this section, we'll shift away from the core and into the places where the uncertain world intrudes on our beautiful assumptions. We've come to the `Phoenix` context.

Contexts represent the boundary for an application. As with all boundaries, it defines the point where our single purpose code meets the world outside. That means contexts are APIs responsible for taking un-sanitized, un-validated data and transforming it to work within the core, or rejecting it before it reaches the core.

The boundary code isn't just an API layer. It's the place we try to hold *all* uncertainty. Our context has at least these responsibilities:

Access External Services

The context allows a single point of access for external services.

Abstract Away Tedious Details

The context abstracts away tedious, inconvenient concepts.

Handle uncertainty

The context handles uncertainty, often by using result tuples.

Present a single, common API

The context provides a single access point for a family of services.

Based on what you're doing in your code, the boundary may have other responsibilities as well. Boundaries might handle process machinery. They might also transform correct outputs to work as inputs to other services. Our generated Phoenix context doesn't have those issues, though. Let's dig a little deeper into the context we've generated.

Access External Services

External services will always be accessed from the context. Accessing external services may result in failure, and managing this unpredictability is squarely the responsibility of the context.

Our application's database is an external service, and the Catalog context provides the service of *database access*. This access is enacted using Ecto code. Just like the rest of our application, Ecto code can be divided into core and boundary concerns. Ecto code that deals with the certain and predictable work of building queries and preparing database transactions belongs in the core. That is why, for example, we found the changeset code that sets up database transactions in the Product schema. Executing database requests, on the other hand, is unpredictable—it could always fail. Ecto implements the Repo module to do this work and any such code that calls on the Repo module belongs in the context module, our application's boundary layer.

Here are a few functions from the context module. Notice that each of them use the Repo module, so we know they're in the right place.

```
generators/pento/lib/pento/catalog.ex
def list_products do
  Repo.all(Product)
end
```

```
generators/pento/lib/pento/catalog.ex
def get_product!(id) do
  Repo.get!(Product, id)
end
```

We can get one product, or a list of them. Here's another one:

```
generators/pento/lib/pento/catalog.ex
def delete_product(%Product{} = product) do
  Repo.delete(product)
end
```

These functions perform some of the classic *CRUD* operations. *CRUD* stands for create, read, update, and delete. We've shown only a few functions here, but you get the idea. We don't want to get too bogged down in the Ecto details. If you need more Ecto information, check out the excellent hex documentation² or the definitive book on Ecto, *Programming Ecto [WM19]* for more details.

The last expression in each of these *CRUD* functions is some function call to `Repo`. Any function call to `Repo` can fail, so they come in one of two forms. By convention, if the function name ends in a `!`, it can throw an exception. Otherwise, the function will return a *result tuple*. These tuples will have either `:ok` or `:error` as their first element. That means it's up to the client of this context to handle both conditions.

If you can't do anything about an error, you should use the `!` form. Otherwise, you should use the form with a result tuple.

Abstract Away Tedious Details

Elixir will always use Ecto to transact against the database. But the work of using Ecto to cast and validate changesets, or execute common queries, can be repetitive. Phoenix contexts provide an API through which we can abstract away these tedious details, and our generated context is no different.

Let's walk through an example of this concept now.

First, we'll take a quick look at what we might have to do to use Ecto directly to insert a new record into the database. You don't have to type this right now; we're going to do it later the easy way:

```
iex> alias Pento.Catalog
Pento.Catalog
iex> alias Pento.Catalog.Product
Pento.Product
iex> alias Pento.Repo
```

2. <https://hexdocs.pm/ecto/Ecto.html>

```

Pento.Repo
iex> attrs = %{
    name: "Battleship",
    sku: 89101112,
    unit_price: 10.00,
    description: "Sink your opponent!"
}
%{
    description: "Sink your opponent!",
    name: "Battleship",
    sku: 89101112,
    unit_price: 10.0
}
iex> product = %Product{}
iex> changeset = Product.changeset(product, attrs)
iex> Repo.insert!(changeset)
{:ok, %Product{...}}

```

Changeset are part of the Ecto library, and as we can see here, working directly with them can be pretty verbose. We need to alias our Product module, build an empty Product struct, and build our changeset with some attributes. Only *then* can we insert our new record into the database.

Luckily, we don't have to get mired in all this drudgery because the Catalog context manages the ceremony for us. The Catalog context's API beautifully wraps calls to query for all products, a given product, and all the other CRUD interactions.

Instead of creating a changeset and inserting it into the database ourselves, we can leverage the generated Catalog context function `Catalog.create_product/1`:

```

generators/pento/lib/pento/catalog.ex
@doc """
Creates a product.

## Examples

    iex> create_product(%{field: value})
    {:ok, %Product{}}

    iex> create_product(%{field: bad_value})
    {:error, %Ecto.Changeset{}}

"""
def create_product(attrs \\ %{}) do
  %Product{}
  |> Product.changeset(attrs)
  |> Repo.insert()
end

```

Let's drop into IEx and try it out:


```

iex> alias Pento.Catalog
Pento.Catalog
iex> attrs = %{
  name: "Candy Smush",
  sku: 50982761,
  unit_price: 3.00,
  description: "A candy-themed puzzle game"
}
%{
  description: "A candy-themed puzzle game",
  name: "Candy Smush",
  sku: 50982761,
  unit_price: 3.0
}
iex> Catalog.create_product(attrs)
[debug] QUERY OK db=0.8ms idle=188.2ms
INSERT INTO "product" ("description","name","sku",
"unit_price","inserted_at","updated_at")
VALUES ($1,$2,$3,$4,$5,$6) RETURNING "id" ["A candy-themed puzzle game",
"Candy Smush", 50982761, 3.0,
~N[2020-09-10 13:28:50], ~N[2020-09-10 13:28:50]]
{:ok,
 %Pento.Catalog.Product{
  __meta__: #Ecto.Schema.Metadata<:loaded, "product">,
  description: "A candy-themed puzzle game",
  id: 3,
  inserted_at: ~N[2020-09-10 13:28:50],
  name: "Candy Smush",
  sku: 50982761,
  unit_price: 3.0,
  updated_at: ~N[2020-09-10 13:28:50]
 }}

```

It works just as advertised, all we had to do was provide a set of attributes with which to create a new product record. Let's move on to the next reason for being for our contexts, presenting a common API.

Present a Single, Common API

A tried and true approach of good software design is to funnel all code for related tasks through a common, unified API. Our schema has features that other services will need, but we don't want external services to call our schema directly. Instead, we'll wrap our `Product.changeset/2` API in a simple function.

```
generators/pento/lib/pento/catalog.ex
```

```

def change_product(%Product{} = product, attrs \\ %{}) do
  Product.changeset(product, attrs)
end

```

This code may seem a little pointless because it is a one-line function that calls an existing function implemented elsewhere. Still, it's worth it because now our clients won't have to call functions in our schema layer directly. That's the core; we want all external access to go through a single, common API.

Now, we can move on to the sticky topic of managing uncertainty.

Handle Uncertainty

One of the most important duties of the context is to translate unverified user input into data that's safe and consistent with the rules of our database. As you have seen, our tool for doing so is the changeset. Let's see how our context works in these instances:

```
generators/pento/lib/pento/catalog.ex
```

```
@doc """
Creates a product.

## Examples

iex> create_product(%{field: value})
{:ok, %Product{}}

iex> create_product(%{field: bad_value})
{:error, %Ecto.Changeset{}}

"""
def create_product(attrs \\ %{}) do
  %Product{}
  |> Product.changeset(attrs)
  |> Repo.insert()
end
```

```
generators/pento/lib/pento/catalog.ex
```

```
def update_product(%Product{} = product, attrs) do
  product
  |> Product.changeset(attrs)
  |> Repo.update()
end
```

This code uses the `changeset/2` function in the `Product` schema to build a changeset that we try to save. If the changeset is not valid, the database transaction executed via the call to `Repo.insert/1` or `Repo.update/1` will ignore it, and return the changeset with errors. If the changeset is valid, the database will process the request. This type of uncertainty belongs in our context. We don't know *what* will be returned by our call to the `Repo` module but it's the context's job to manage this uncertainty and orchestrate any downstream code that depends on these outcomes.

Now that you understand how to use the context to interact with our application's database, let's put that knowledge to use.

Use The Context to Seed The Database

As we continue to develop our web application and add more features in the coming chapters, it will be helpful to have a quick and easy way to insert a set of records into the database. We'll create some seed data to populate our database and we'll use our context to do it.

Open up `priv/repo/seeds.exs` and key this in:

```
generators/pento/priv/repo/seeds.exs
alias Pento.Catalog

products = [
  %{
    name: "Chess",
    description: "The classic strategy game",
    sku: 5_678_910,
    unit_price: 10.00
  },
  %{
    name: "Tic-Tac-Toe",
    description: "The game of Xs and Os",
    sku: 11_121_314, unit_price: 3.00
  },
  %{
    name: "Table Tennis",
    description: "Bat the ball back and forth. Don't miss!",
    sku: 15_222_324,
    unit_price: 12.00
  }
]

Enum.each(products, fn product ->
  Catalog.create_product(product)
end)
```

Now, execute it:

```
[pento] → mix run priv/repo/seeds.exs
[debug] QUERY OK db=1.6ms decode=0.6ms queue=1.0ms idle=8.6ms
INSERT INTO "product" ("description","name","sku","unit_price","inserted_at",
"updated_at") VALUES ($1,$2,$3,$4,$5,$6) RETURNING "id"
["The classic strategy game", "Chess", 5678910, 10.0,
~N[2020-09-13 13:43:05], ~N[2020-09-13 13:43:05]]
[debug] QUERY OK db=0.8ms queue=1.1ms idle=21.3ms
INSERT INTO "product" ("description","name","sku",
"unit_price","inserted_at","updated_at")
VALUES ($1,$2,$3,$4,$5,$6) RETURNING "id"
```

```
[ "The game of Xs and Os", "Tic-Tac-Toe", 11121314, 3.0,
~N[2020-09-13 13:43:05], ~N[2020-09-13 13:43:05]]
[debug] QUERY OK db=0.9ms queue=0.8ms idle=23.4ms
INSERT INTO "product" ("description","name","sku","unit_price",
"inserted_at","updated_at") VALUES ($1,$2,$3,$4,$5,$6) RETURNING
"id" [ "Bat the ball back and forth. Don't miss!", "Table Tennis",
15222324, 12.0, ~N[2020-09-13 13:43:05], ~N[2020-09-13 13:43:05]]
```

Nice! The log shows each new row as Ecto inserts it. For bigger seed files, we could make this code more efficient by using batch commands. For these three records, it's not worth the time.

After looking at these layers, you might ask yourself “Where should new code go?” The next section has some advice for you as you organize your project.

Boundary, Core, or Script?

As you add new functions, you can think about them in this way: any function that deals with process machinery (think “input/output”) or uncertainty will go in the *boundary*, or context. Functions that have certainty and support the boundary go in the *core*. Scripts that support operational tasks, such as running tests, migrations, or seeds, live outside the `lib` codebase altogether. Let's dive a little deeper.

The Context API is With-land

As you saw in [Chapter 2, Phoenix and Authentication, on page 31](#), the context module is the API for a service. Now you know that the context module acts as the application's boundary layer. Boundary layers handle uncertainty. This is why one of the responsibilities of the context is to manage database interactions, for example—database requests can fail.

In Elixir, we can use `with` statements to manage code flow that contains uncertainty. The `with/1` function allows us to compose a series of function calls while providing an option to execute if a given function's return doesn't match a corresponding expectation. Reach for `with/1` when you can't pipe your code cleanly.

So, you can think of the boundary as *with-land*—a place where you want to leverage the `with/1` function, rather than the pipe operator, to compose code that deals with uncertainty. You might chafe a bit at this advice. Many Elixir developers fall in love with the language based on the beautiful idea of composing with pipes, but the pipe operator often falls short of our needs in the context, or boundary layer. Let's take a look at why this is the case.

We'll start by looking at an appropriate usage of the pipe operator in our application's core. Here's what a pipe that builds a query might look like:

```
defmodule Catalog.Product.Query do
  ...
  def base_product_query, do: Product
  def cheaper_than(query, price), do: from p in query, where...
  def cheap_product_skus(price)
    base_product_query()
    |> cheaper_than(price)
    |> skus
  end
  ...
end
```

Don't worry about how the individual functions work. Just know they build queries or transform them. If we've verified that price is correct, this code should not fail. In other words, the behavior of this code is certain. Pipes work great under these conditions.

When the outcome of a given step in a pipeline *isn't* certain, however, pipes are *not* the right choice. Let's look at what an *inappropriate* usage of the pipe operator in our application's boundary layer, the context, might look like.

Consider the following code:

```
# this is a bad idea!

defmodule Pento.Catalog do
  alias Catalog.Product.Query
  alias Pento.Repo

  def run_query(query) do
    {:ok, list} = Repo.all(query)
    list
  end

  def send_sale_skus(query) do
    query
    |> Product.cheap_product_skus(25.00)
    |> run_query
    |> Service.send_bargains
  end
end
```

This fictional code takes an input, builds a query, executes it, and then sends the result to an external service. It doesn't matter how you send the service request; it only matters that you must invoke the `Service.send_bargains/1` function with an argument of the product list returned from `run_query/1`. But the `run_query/1` function can fail! This means that `send_sale_skus/1` won't work reliably. Whenever

run_query/1 fails to return the list of products, our code will blow up. In fact, the result tuple we abstract away in the run_query/1 function is a hint that something might go wrong. The pipeline we built can handle the :ok case, but not the error case.

Instead of this code, we need to compose such statements with Elixir's with/1 function.³ Here's what a with example might look like:

```
defmodule Pento.Catalog do
  alias Catalog.Product.Query
  alias Pento.Repo

  def run_query(query) do
    Repo.all(query)
  end

  def get_sale_skus(query) do
    query
    |> Product.cheap_product_skus(25.00)
    |> run_query
  end

  def send_sale_skus(query) do
    with {:ok, products} <- get_sale_skus(query),
         {:ok, response} <- Service.send_bargains(products) do
      response
    else
      {:error, reason} ->
        IO.puts "Error sending sale skus: #{reason}"
    end
  end
end
```

Some Elixir programmers are frustrated when they encounter code that uses with, because it is more verbose than piped code. The truth is that code with uncertainty *needs to be* more verbose, because it *must* deal with failure.

If you find yourself mired in too much with, remember that with code properly belongs in the application's boundary layer, the context. Use with in boundary code; use the pipe operator, |>, in core code, and seek to move as much code as possible *from the boundary to the core*!

The Core is Pipe-land

The core has functions that support the boundary API. We'll learn to shape core code from scratch a bit later. For now, know that core code is *predictable*

3. <https://elixirschool.com/en/lessons/basics/control-structures/#with>

and *reliable* enough that we can compose our functions together with pipes. Let's look at a few examples.

Though executing queries might fail, building queries is completely predictable. If you're building code to compose complex queries, you'll put that code in the *core*. So you can remember this rule:

Build a query in the *core* and execute it in the *boundary*.

Schemas don't actually interact with the database. Instead, think of them as road maps that describe how to tie one *Elixir module* to a *database table*. The schema doesn't actually connect to the database; it just has the data that answers key questions about how to do so:

- What's the name of the table?
- What are the fields the schema supports?
- What are the relationships between tables?

Once you've debugged your code, the outcomes of schema definitions are *certain*. Put them in the core.

Working with data that comes from the database is *predictable and certain*, so code that constructs or validates database transactions can go in the core.

Operations Code

We've looked at boundary and core code. Sometimes, you need code to support common development, deployment, or testing tasks. Rather than compiling such operations code, Elixir places it in scripts. Migrations, other mix tasks, and code to add data to your database fit this model. Put such code in `/priv`. If it deals with the database, the code will reside in `/priv/repo`. Mix configuration will go in `mix.exs`. Configuration of your main environments goes in `/config`. In general, `.exs` scripts simply go where they are most convenient.

We've been working for a whole chapter, and we're still not done with the generated code! That's OK. It's time for a much-needed break.

Your Turn

Generating code is a useful technique for creating an early foundation you can freely customize. You'll use it when developing your own Phoenix LiveView apps, anytime you need to quickly build the CRUD functionality that so often forms the basis of more complex, custom features.

The Phoenix Live generator has a layering system, and the backend layers include core and boundary code. In the core, the schema contains information

to define a struct that ties Elixir data to fields in a database. Each struct represents a row of the database. Changesets implement change policies for those rows.

Phoenix Contexts represent the boundary layer of your application, with important responsibilities for protecting the core. Each context presents a common API for some problem domain, one that abstracts away tedious details. Contexts wrap services and handle unverified user input.

Now you get to put some of these ideas into practice.

Give It a Try

You'll have more of an opportunity to get your hands dirty with the exercises at the end of the next chapter. Until then, these tasks will give you some practice with writing core and boundary code.

- Create another changeset in the Product schema that only changes the `unit_price` field and only allows for a price decrease from the current price.
- Then, create a context function called `markdown_product/2` that takes in an argument of the product and the amount by which the price should decrease. This function should use the new changeset you created to update the product with the newly decreased price.

Next Time

In the next chapter, we'll cover the frontend generated code we've not yet touched. Don't stop now, we're just getting started!

Generators: Live Views and Templates

In the last chapter, we generated a context and live views for a service that will let us enter products into a user interface and save them into a database. Then, we started a careful exploration of the *context*, the application's boundary layer, along with the schema, our functional core. In this chapter, we're going to shift to an examination of the frontend code that the Phoenix Live generator built.

By taking a deep dive through the generated frontend code, you'll understand how LiveView works to support the CRUD functionality for the Product resource, you'll experience some of the best ways to organize LiveView code, and you'll be prepared to build custom LiveView functionality on top of this strong foundation.

Before we dive in, let's make a plan.

First, we'll start with the routes and use them to understand the views that our generator has made available to the user. Then, we'll take inventory of the files that the generator created. We'll look at these files and what each one does.

Finally, we'll walk through the main details of a live view and show you how things work. Along the way, you'll pick up a few new concepts you haven't seen before. We'll introduce live navigation and live actions and demonstrate how LiveView builds and handles routes. We'll explore ways to navigate between pages, without actually leaving a given live view. We'll illustrate how LiveView's lifecycle manages the presentation and state of a given view for a user. Lastly, we'll introduce LiveView components and lay out how to organize LiveView code properly.

When you are through, you won't *just* know about this generated code. You will understand how experts weave typical LiveView applications together,

and how well-structured code is layered. You'll be prepared to write your own LiveView applications, the right way.

Application Inventory

So far, we've spent all of our time on the backend service that manages products in our catalog. We were lucky because we could focus our exploration on a single API, the Catalog context.

In the live view, we're not so lucky. There are nearly a dozen files that we need to worry about. It would be nice to start with a common interface for our user-facing features.

It turns out we *do* have such an interface, but it's not an Elixir API. Instead, it's a list of the routes a user can access. That's right. The routes in `lib/pento_web/router.ex` are an API.

Let's take a look at the LiveView routes we generated in the last chapter now.

Route the Live Views

The router tells us all of the things a user can *do* with products. It represents a high-level overview of how each piece of CRUD functionality is backed by LiveView. Let's take a look at the routes we added to our application's router after we ran the generator in the previous chapter. We'll break down how Phoenix routes requests to a live view and how LiveView operates on those requests.

```
generate_web/pento/lib/pento_web/router.ex
live "/products", ProductLive.Index, :index
live "/products/new", ProductLive.Index, :new
live "/products/:id/edit", ProductLive.Index, :edit

live "/products/:id", ProductLive.Show, :show
live "/products/:id/show/edit", ProductLive.Show, :edit
```

That's exactly the API we're looking for. This list of routes describes all of the ways a user can interact with products in our application. Each of these routes starts with a macro defining the type of request, followed by three options. All of our routes are live routes, defined with the `live` macro. We'll take a brief look at where this function comes from. Then, we'll talk about what it does for us.

The `live/4` macro function is implemented by the `Phoenix.LiveView.Router` module. It is made available inside the router thanks to this line:

```
generate_web/pento/lib/pento_web/router.ex
use PentoWeb, :router
```

Here, we see another example of Phoenix’s reliance on macros to share code—even the generated code takes this approach! The `use` macro injects the `PentoWeb.router/0` function into the current module. If we look at that module now, we see that it in turn imports the `Phoenix.LiveView.Router` module.

```
generate_web/pento/lib/pento_web.ex
def router do
  quote do
    use Phoenix.Router

    import Plug.Conn
    import Phoenix.Controller
    import Phoenix.LiveView.Router
  end
end
```

For a closer look at exactly how `use`, and macros in general, work in Elixir, check out Chris McCord’s [Metaprogramming Elixir \[McC15\]](#).

For our purposes, it is enough to understand that the `live/4` macro function is available in our application’s router by way of the `Phoenix.LiveView.Router` module. Let’s move on to discuss what this function does.

The `live` macro generates a route that ties a URL pattern to a given `LiveView` module. So, when a user visits the URL in the browser, the `LiveView` process starts up and renders a template for the client.

The first argument to a `live` route is the URL pattern. This pattern defines what the URL looks like. Notice the colons. These represent *named parameters*. For example, if the user types the URL `products/7`, the router will match the pattern `"/products/:id"`, and prepare this map of params to be made available to the corresponding live view:

```
%{"id" => "7"}
```

The second argument to a `live` route is the `LiveView` module implementing our code. If you look closely at the list of routes, the first three all specify the `ProductLive.Index` module. This module represents an entire live view that will handle all of the “list products”, “create new product” and “edit existing product” functionalities. The next two routes specify the `ProductLive.Show` module. Notice that it takes *just these two* modules put together to implement our entire single-page app! As we’ll see throughout this chapter, `LiveView` offers a simple and organized solution for managing even complex single-page functionality without writing a large amount of code.

The final argument to `live/4` is called the *live action*. The action allows a given live view to manage multiple page states.

For example, as these routes indicate, you'll see that the `ProductLive.Index` view implements three different live actions: `:index`, `:new`, and `:edit`. This means that *one* live view, `ProductLive.Index`, will handle the `:index` (read all products), `:new` (create a product), and `:edit` (update a product) portions of the Product CRUD feature-set. That's because both the `:new` and `:edit` actions will build pop ups, or modal dialogs, that sit on top of a list of products, all within the single `ProductLive.Index` live view.

The `ProductLive.Show` live view implements two different actions: `:show` and `:edit`. This means that the `ProductLive.Show` live view handles *both* the `:show` (read one product) and `:edit` (update a product) functionality. Notice that this is the second appearance of the `:edit` action. Just like the `ProductLive.Index`, the `ProductLive.Show` live view *also* uses this action to build a pop up—this time placing it on top of the single product page. So, users will have two interfaces through which they can edit a product.

If this seems like a lot of detail right now, don't worry. We'll break it down later on in this chapter. For now, it's enough to understand that a single live view can handle multiple page states, and therefore multiple features, with the help of live actions.

With that first pass behind us, let's take a second look at the output from the generator in and familiarize ourselves with the generated files.

Explore The Generated Files

Before we dive into the generated LiveView code, let's briefly review the files that the Phoenix Live generator created. This will help us navigate around the code in the next few sections.

When we ran the `mix phx.gen.live` command, the code generator told us exactly which files it created. It's been a while, so we'll show them to you again. This is the portion of output from the generator describing the frontend files, though they're shown here in a different order:

```
* creating lib/pento_web/live/product_live/show.ex
* creating lib/pento_web/live/product_live/show.html.leex

* creating lib/pento_web/live/product_live/index.ex
* creating lib/pento_web/live/product_live/index.html.leex

* creating lib/pento_web/live/product_live/form_component.ex
* creating lib/pento_web/live/product_live/form_component.html.leex

* creating lib/pento_web/live/modal_component.ex
* creating lib/pento_web/live/live_helpers.ex
* injecting lib/pento_web.ex
```

```
* creating test/pento_web/live/product_live_test.exs
```

That's a lot of code! Let's break it down.

The `show.ex` file implements the `LiveView` module for a single product. It uses the `show.html.leex` template to render the HTML markup representing that product. Similarly, both `index.ex` and `index.html.leex` together implement a list of products.

The `form_component.ex` and `form_component.html.leex` both implement a new feature you've not seen yet called a *LiveView component*. Don't worry about the details of components quite yet. These two files together represent the form that we'll use to collect input for a new or changing product.

The rest of the files represent supporting files and tests. We'll get to them a bit later in this chapter.

Before we dive into the code, there's one more thing you need to know about—LiveView's two key workflows. There are two main workflows in the LiveView programming model—the mount and render workflow and the change management workflow.

We'll begin with the mount/render workflow for our Product Index feature. Then, we'll move on to the change management workflow and look at how it allows us to use the same live view to support the Product New and Product edit features.

Mount and Render the Product Index

The mount/render workflow describes the process in which a live view sets its initial state and renders it, along with some markup, for the client. The best way to understand the mount/render workflow is to see it in action. The Product Index feature is our entry-point into this workflow. We'll play around with adding specific data to that live view's socket in order to tweak what's rendered.

The easiest way to put data into the socket is via the `mount/3` function. Open up `lib/pento_web/live/product_live/index.ex` and look at the live view's `mount/3` function:

```
def mount(_params, _session, socket) do
  {:ok, assign(socket, :products, list_products())}
end

# ...

defp list_products do
  Catalog.list_products()
end
```

The generator has built us a `mount/3` function in which the socket assigns is updated with a key of `:products`, pointing a value of all of the products returned from the `list_products/0` helper function.

Let's update this `mount/3` function to add an additional key of `:greeting` to the socket assigns. We'll do so building a small pipeline of calls to the `assign/3` function, like this:

```
def mount(_params, _session, socket) do
  {:ok,
   socket
   |> assign(:greeting, "Welcome to Pento!") # add this line
   |> assign(:products, list_products())}
end
```

This call to the `Phoenix.Socket.assign/3` function adds a key/value pair to the socket struct's map of `:assigns`. As you'll remember from [Chapter 2, Phoenix and Authentication, on page 31](#), the `:assigns` map is where the socket struct holds state. Any key/value pairs placed in the socket's `:assigns` can be accessed in the live view's template via a call to the assignment. For example, you can now access the value of the `:greeting` key in this live view's corresponding template like this: `@greeting`. Go ahead and do that now.

Open up `lib/pento_web/live/product_live/index.html.leex` and add a header that renders the value of the `@greeting` assignment.

```
<h1><%= @greeting %></h1>
```

Now, start up the Phoenix server by executing the `mix phx.server` command in your terminal and point your browser at `localhost:4000/products`. You should see the Product Index page render with your greeting!

Products

Welcome to Pento!

Name	Description	Unit price	Sku	
Chess	The classic strategy game	10.0	5678910	Show Edit Delete

Let's break down what happens under the hood when you navigated to the `/products` URL. But first, you need to understand how the LiveView framework leverages Elixir's behaviours to enact the `mount/render` workflow.

Understand LiveView Behaviours

Let's peel back the onion one layer. Live views are called *behaviours*. (Yes, we spell them the proper British way, in honor of Joe Armstrong and the rest of the Swedish team that created OTP.) Think of consuming a behaviour as the opposite of using a library. When you use a library, your program is in control, and it calls library functions.

Live views don't work like that. Your code is not in control. The behaviour runs a specified application and calls your code according to a contract. The LiveView contract defines several callbacks.¹ Some are optional, and you must implement others.

When we talk about the LiveView lifecycle, we're talking about a specific program defined in the behaviour. This includes the `mount/3` function to set up data in the socket, the `render/1` function to return data to the client, the `handle_*` functions to change the socket, and an optional `terminate/2` function to shut down the live view.

When we say that `mount/3` happens before `render/1` in a live view, we don't mean `mount/3` actually calls `render/1`. We mean the *behaviour* calls `mount/3`, and then `render/1`.

It's time to take a closer look at how LiveView's behaviour works, starting with a live route and ending with the first render.

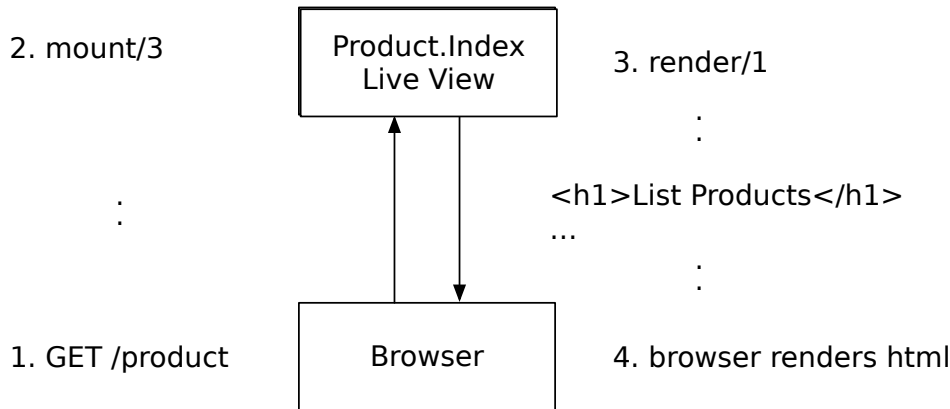
Route to the Product Index

The entry point of the `mount/render` lifecycle is the route. When you point your browser at the `/products` route, the router will match the URL pattern to the code that will execute the request, and extract any parameters. Next, the live view will start up. If you care about the details, the live view is actually an OTP GenServer. If this doesn't mean anything to you, don't be concerned. You only need to know what the different parts of the behaviour do.

The first call that the LiveView behaviour will make to our code is the `mount/3` function. Its job is to *set up* the initial data in the live view. Next, the live view will do the initial *render*. If we've defined an explicit `render/1` function, the behaviour will use it. If not, LiveView will render a template based on the name of the live view file. There's no explicit `render/1` function defined in the `ProductLive.Index` live view, so our live view will render the template in the `index.html.leex` file.

1. https://hexdocs.pm/phoenix_live_view/Phoenix.LiveView.html#callbacks

If you would rather not think about the behaviour, that's ok. You can think about it in simplistic terms instead. This diagram describes what's happening:



This flow represents a common pattern in Elixir and Phoenix programming that you'll see again and again as you work in LiveView. When the LiveView process starts up, the socket is *initialized* or *constructed*. The `mount/3` function further *reduces* over that socket to make any state changes. Then, the `render/1` function *converts* that socket state into markup which is delivered to the client in the browser. Keep this “construct, reduce, convert” pattern in mind as we dive deeper into LiveView. It will help you understand how LiveView manages the state of your single-page app and will prepare you to write clean, organized LiveView code in the coming chapters.

Now that you know what will happen after the route is matched, let's open up the code in our live view and trace through it line by line.

Establish Product Index State

The job of the `ProductLive.Index` live view is to manage all actions that deal with lists of products. Regardless of the URL pattern we match to get there, each path takes us first to the `mount/3` function:

```
generate_web/pento/lib/pento_web/live/product_live/index.ex
@impl true
def mount(_params, _session, socket) do
  {:ok, assign(socket, :products, list_products())}
end
```

You already know that a live view revolves around its state. The `mount/3` function sets up the initial state, in this case adding a list of products into the socket assigns. Here, it does so with the help of the `list_products()` function. Open up

the Product Index live view in `pento/lib/pento_web/live/product_live/index.ex` and take a look:

```
generate_web/pento/lib/pento_web/live/product_live/index.ex
defp list_products do
  Catalog.list_products()
end
```

The `Catalog.list_products/0` function is used to get a list of products. With that, you can see how the generated backend code we explored in the previous chapter supports the CRUD interactions on the frontend.

Now that the product list has been added to `socket assigns` in the `mount/3` function, the socket will look something like this:

```
%{
  ...some private stuff...,
  assigns: %{
    live_action: :index,
    products: %([...a list of products...]),
    ...other keys...
  }
}
```

Our LiveView's index state is complete and ready to be rendered! Since our live view doesn't implement a render function, the behaviour will fall back to the default `render/1` function and render the template that matches the name of the LiveView file, `pento/pento_web/live/index.html.leex`. It's time to discuss the template.

Render Product Index State

Before we take a closer look at how the template renders the products in socket assigns, let's look at just how these LEEEX templates function in a LiveView application.

LiveView's built-in templates use the `.leex` extension. *LEEX*, or *Live EEX*, is similar to *EEX* except that it is designed to minimize the amount of data sent down to the client over the WebSocket connection. Part of the job of these templates is to track state changes in the live view socket and *only* update portions of the template impacted by these state changes.

If you've ever worked with a web scripting language before, LEEEX will probably look familiar to you. The job of the `pento/pento_web/live/index.html.leex` template is simple. It has text and substitution strings in the form of eex tags.

Most of the file is pure text—usually HTML—that will be rendered one time upon the first render. The rest of the template has embedded Elixir snippets.

When the eex compiler encounters Elixir code within the `<%= %>` tags (notice the `=`), the compiler will compute the code and leave the result in place of the embedded Elixir. When the eex compiler encounters the `<% %>` tags, any Elixir code between them will be computed, but nothing will be rendered in their place.

LiveView makes the data stored within `socket.assigns` available for computations in LEEEx templates. When that data changes, the LEEEx template is re-evaluated, and the live view will keep track of any differences from one evaluation to the next. This allows the live view to *only* do the work of re-rendering portions of the template that have actually changed based on changes to the state held in `socket.assigns`. In this way, LEEEx templates are highly efficient.

After the first invocation of `mount/3`, the only thing we've added to `socket.assigns` is the `:products` key. Let's take a look at how we'll render those products:

```
generate_web/pento/lib/pento_web/live/product_live/index.html.leex
```

```
<table>
  <thead>
    <tr>
      <th>Name</th>
      <th>Description</th>
      <th>Unit price</th>
      <th>Sku</th>

      <th></th>
    </tr>
  </thead>
  <tbody id="product">
    <%= for product <- @products do %>
      <tr id="product-<%= product.id %>">
        <td><%= product.name %></td>
        <td><%= product.description %></td>
        <td><%= product.unit_price %></td>
        <td><%= product.sku %></td>
```

Here, we're iterating over the products in `socket.assigns.products`, available in the template as the `@products` assignment, and rendering them into an HTML table. You'll notice the `for` expression executing this iteration.

Take a look at the block of text follow the `<%= for product <- @products do %>` statement. It might not surprise you to learn that Phoenix will render all of this code for each product in the `@products` list. Let's take a closer look:

```
<tr>
<td><%= product.name %></td>
<td><%= product.description %></td>
<td><%= product.unit_price %></td>
<td><%= product.sku %></td>
```

```
...
</tr>
```

This code renders a separate table row for each product in the list, looking up the `product.name`, `product.description`, and so on for each column in the table.

What might surprise you is that after the first render, Phoenix will render each snippet *only when values change*!

And with that knowledge under your belt, you've seen the entire `mount/render` workflow in action. First, we set up the socket using `mount/3`, and then we render it in the `index.html.leex` template via an implicit `render/1` function.

Now, let's move on to some scenarios that actually *change* our socket through the use of params and event handlers. We're ready to dive into the change management workflow.

Handle Change for the Product Edit

The `ProductLive.Index` live view will also support the Product Edit and Product New features by using the change management workflow to alter socket state with event handlers. In this way, a single live view can easily handle multiple pieces of CRUD functionality.

We'll examine the change management workflow now, starting with the Product Edit functionality.

Route to the Product Edit

Let's trace the code that fires when you point your browser at the `/products/:id/edit` route, starting with the route definition.

The router contains the following generated route for the Product Edit feature:

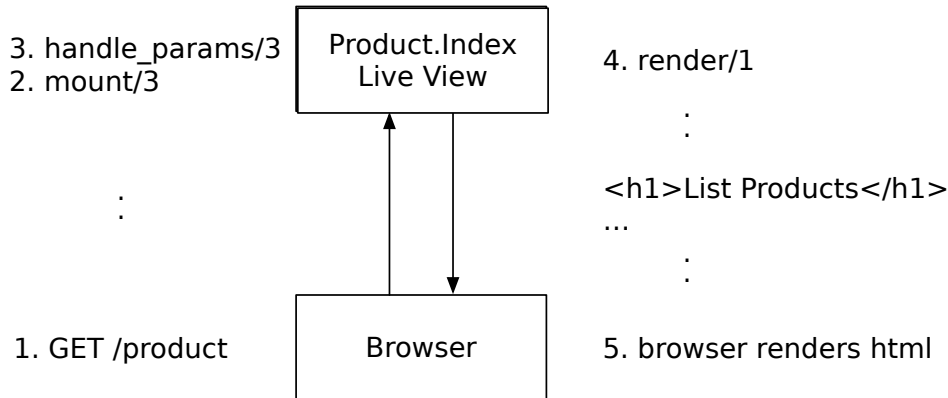
```
live "/products/:id/edit", ProductLive.Index, :edit
```

This maps the `/products/:id/edit` route to the same `ProductLive.Index` live view that we examined earlier, this time with a live action of `:edit`. By specifying a live action in the route definition, `LiveView` adds a key of `:live_action` to the live view's socket assigns, setting it to the value of the provided action.

In order to take advantage of this live action to change the live view's state, we'll hook into a slightly different `LiveView` lifecycle than we saw for `mount/render`.

When we navigate to the Product Index route, `/products`, the `LiveView` lifecycle that kicks-off first calls the `mount/3` lifecycle function, followed by `render/1`. If, however, we want to access and use the live action from socket assigns, we

must do so in the `handle_params/3` lifecycle function. This callback, if it is implemented, is called right after the `mount/3`. So, our adjusted LiveView lifecycle looks something like this:



Before we take a closer look at `handle_params/3`, it's important to understand that there is another way a user can navigate to the Product Edit view. In addition to pointing the browser directly to the Product Edit route at `products/:id/edit`, the generated Product Index template includes some code that builds a link to this route.

Live Navigation with `live_patch/2`

Open up the template at `lib/pento_web/live/product_live/index.html.leex` and take a look at this markup:

```
generate_web/pento/lib/pento_web/live/product_live/index.html.leex
<%= live_patch "Edit",
  to: Routes.product_index_path(@socket, :edit, product)%>
```

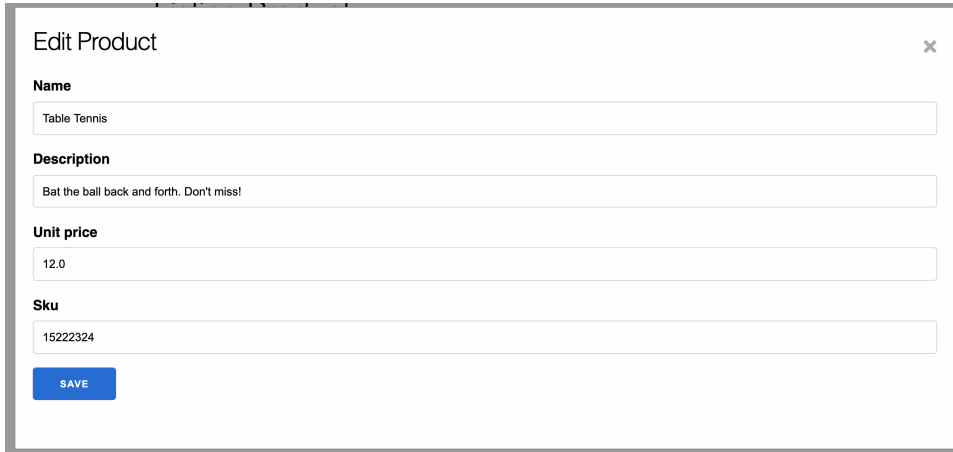
This markup generates an HTML link that the user can click to be taken to the Product Edit view. Open the element inspector in your browser and inspect the edit link. You'll see the following HTML generated by this markup:

```
<a data-phx-link="patch" data-phx-link-state="push" href="/products/1/edit">
  Edit
</a>
```

This is a special kind of link called a “live patch”, returned by the call to the `live_patch/2` function. A live patch link will “patch” the current live view. This means that clicking the link *will* change the URL in the browser bar, courtesy of a JavaScript feature called *push state navigation*. But it *won't* send a web request to reload the page. Instead, clicking this link will kick off LiveView's

change management workflow—the `handle_params/3` function will be invoked for the linked LiveView, followed by the `render/1` function.

So, when you click the edit link on the product index template, you'll see a modal pop up with the edit product form, like this:

A screenshot of a web application modal titled "Edit Product" with a close button (X) in the top right corner. The form contains four input fields: "Name" with the value "Table Tennis", "Description" with the value "Bat the ball back and forth. Don't miss!", "Unit price" with the value "12.0", and "Sku" with the value "15222324". At the bottom left of the form is a blue button labeled "SAVE".

and if you take a peek at the URL, you'll see that it has changed to read `/products/1/edit!`

Navigating with `live_patch/2` causes LiveView to behave a little bit differently than pointing your browser at `/products/1/edit`. While the lifecycle triggered by *navigating your browser* to a route with a live action first calls the `mount/3`, *then* `handle_params/3`, and finally `render/1`, the lifecycle triggered by the `live_patch` request skips the call to `mount/3`.

But, whether you click the edit link for the first product on the list or point your browser at the edit route for that product, the `ProductLive.Index` live view will call `handle_params/3`. The `handle_params/3` function will therefore be responsible for using these data points to update the socket with the correct information so that the template can render with the markup for editing a product.

It's time to take a closer look at how the `handle_params/3` function works to set the “edit product” state.

Establish Product Edit State

The change management workflow begins when the user enacts a change by clicking the edit link. Phoenix then calls `handle_params/3` with a live action of `:edit` and the params of `%{id: 1}`. It's time to apply the live action to our live view's socket state.

You can see that the generated `handle_params/3` function invokes a helper function, `apply_action/3` to do exactly that:

```
generate_web/pento/lib/pento_web/live/product_live/index.ex
defp apply_action(socket, :edit, %{ "id" => id }) do
  socket
  |> assign(:page_title, "Edit Product")
  |> assign(:product, Catalog.get_product!(id))
end
```

This code is a simple pipe, with each fragment taking and returning a socket.

Here, the code is setting a `:page_title` of "Edit Product". You can also see that pattern matching is being used to extract the `:id` from `params`. Then, the product ID is fed to `Catalog.get_product!/1` to extract the full product from the database. Finally, the product is added to `socket.assigns` under a key of `:product`. Since the socket has changed, LiveView pushes only the changed state to the client, which then renders those changes.

The `handle_params/3` function *changes* socket state in accordance with the presence of a live action and certain params. This is exactly in-line with the pattern we discussed earlier. The live view's socket is initialized, then the `handle_params/3` function *reduces* over that socket to change its state. Lastly, the socket state is *transformed* via the rendering of the template into markup for the client.

You can see now how LiveView uses live actions, params, and the `handle_params/3` callback to manage complex page state *within a single live view*. With the `handle_params/3` callback, LiveView provides an interface for managing change. As the state of your single-page app becomes more complex, and needs to accommodate changes brought on by additional user interaction like filling out a form, LiveView will continue to use this interface. In this way, LiveView scales beautifully to manage additional complexity.

Now, let's shift our attention to rendering, and see how LiveView will handle our new socket state.

Render Product Edit State

Let's look at the way LiveView renders the index template with the new `:live_action` and `:product` keys from `socket.assigns`. Remember, we're still in the `ProductLive.Index` live view, so we'll look at the `index.leex.html.leex` template:

```
generate_web/pento/lib/pento_web/live/product_live/index.html.leex
<%= if @live_action in [:new, :edit] do %>
  <%= live_modal @socket, Pentoweb.ProductLive.FormComponent,
    id: @product.id || :new,
```

```

    title: @page_title,
    action: @live_action,
    product: @product,
    return_to: Routes.product_index_path(@socket, :index)%>
<% end %>

```

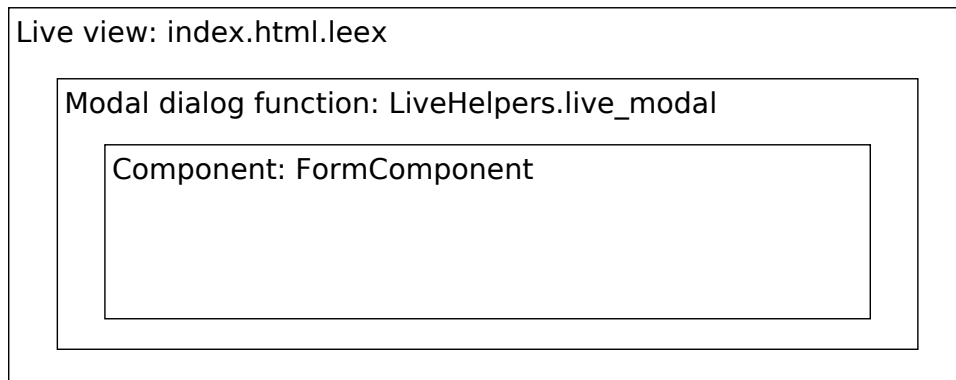
Notice the conditional logic that's based on the `@live_action` assignment. If `@live_action` is `:new` or `:edit`, we want to render a form with the `@product` assignment. That means we'll render a modal component.

It's time to dive into this modal component now. Along the way, you'll see how further change management workflows can be kicked off by user interactions on the page and handled by LiveView's `handle_event/3` callback. You'll see some additional types of live navigation and you'll learn how LiveView leverages components to organize code into layers.

LiveView Layers: The Modal Component

While we'll take a deeper dive into components in upcoming chapters by building our own, the following walk-through will give you a basic understanding of what LiveView components are, how they are used to organize live views, and what role they play in LiveView change management. You'll understand them as yet another LiveView feature you can use to write simple, organized code that handles complex single-page activities. Armed with that understanding, you'll be prepared to build your own components later on in this book.

We'll begin with a quick look at how the generated component code is organized into layers that compartmentalize presentation and state. This figure shows how the pieces fit together:



The Product Edit page will have three distinct layers. The first layer is the *background*. That's implemented with the base index template and Index live

view, and it's responsible for rendering the products table in the background. It's the full live view we've been examining.

The next layer is the modal dialog. Its job is to provide a window container, one that prevents interaction with the layers underneath and contains the form component. It's comprised of HTML markup with supporting CSS, and a small modal *component*. Components are like little live views that run in the process of their parent live view. This first component will render a container with arbitrary markup and handle events for closing the dialog.

The final layer is the form component. Its job is threefold. It holds data in its own socket, renders itself, and processes messages that potentially change its state.

It's time to dive into the modal dialog layer now.

Call the Modal Component

The code flow that renders the product form modal is kicked off in the very same Product Index template we've been examining. If the ProductLive.Index live view's state contains the `:edit` or `:new` live action, then the index template will invoke some code that renders the modal component.

Here's another look at the line of code that calls the modal component from the index template:

```
generate_web/pento/lib/pento_web/live/product_live/index.html.leex
<%= if @live_action in [:new, :edit] do %>
  <%= live_modal @socket, PentoWeb.ProductLive.FormComponent,
    id: @product.id || :new,
    title: @page_title,
    action: @live_action,
    product: @product,
    return_to: Routes.product_index_path(@socket, :index)%>
<% end %>
```

These few lines of code behave differently than the code we've traced so far, so we're going to take our time to walk through what's happening. They get the snowball rolling toward our product form component. There are three concepts crammed together tightly here, and we're going to take them apart one piece at a time.

The first is the conditional statement predicated on the value of the `@live_action` assignment. You'll use this technique to selectively display data on a page depending on what route a user has navigated to—recall that the route definition determines if and how the `live_action` assignment is populated.

Revisiting the change management workflow we’ve seen so far—when a user clicks the “edit” link next to a given product, the `ProductLive.Index` live view will invoke the `handle_params/3` callback with a `live_action` of `:edit` populated in the socket assigns. The template will then re-render with this `@live_action` assignment set to `:edit`. Therefore, the template *will* call the `live_modal/3` function. Let’s dig into this function now.

This function wraps up two concepts. The first is a CSS concept called a *modal dialog*. The generated CSS applied to the modal component will disallow interaction with the window underneath. The second concept is the component itself, and we’ve promised to give you details soon. This component handles details for a modal window, including an event to close the window.

In order to take a look at the modal dialog that will be rendered onto the index template via the call to `live_modal/3`, we need to look under the hood of this generated function.

The Phoenix Live generator builds the `live_modal/3` function and places it in the `lib/pento_web/live/live_helpers.ex` file. Its sole responsibility is to build a modal window in a div that holds the component defined in `PentoWeb.ModalComponent`. The only job of the `PentoWeb.ModalComponent` is to apply some markup and styling that presents a window in the foreground, and handles the events to close that window, without letting the user access anything in the background:

```
generate_web/pento/lib/pento_web/live/live_helpers.ex
def live_modal(socket, component, opts) do
  path = Keyword.fetch!(opts, :return_to)
  modal_opts = [
    id: :modal,
    return_to: path,
    component: component,
    opts: opts
  ]
  live_component(socket, PentoWeb.ModalComponent, modal_opts)
end
```

This function is just a couple of assignments and a function call. The first assignment defines the link that the component will use when a user closes the window. The second is a list of options we’ll send to the component. Then we call `live_component/3` to inject the component, `PentoWeb.ModalComponent`. Let’s take a closer look at how that component is rendered now.

Render The Modal Component

It’s finally time to discuss components in a bit more depth. A *component* is part of a live view that handles its own markup. Some components process

events as well. Just as you can break one function into smaller ones, you can break one live view into smaller pieces of code with components.

Crack open the `lib/pento_web/live/modal_component.ex` file, and let's read it from the top down. This overview will give us a basic sense of the responsibilities of the modal component. Then, in the following sections we'll dive further into how it all works.

First, you can see that the `Modal.Component` module uses the `Phoenix.LiveComponent` behaviour. More on this in a bit.

```
generate_web/pento/lib/pento_web/live/modal_component.ex
use PentoWeb, :live_component
```

You'll also notice that, rather than using a template, the generated component uses an explicitly defined `render/1` function to return the markup that will be sent down to the client:

```
generate_web/pento/lib/pento_web/live/modal_component.ex
@impl true
def render(assigns) do
  ~L"""
  <div id="<%= @id %>" class="phx-modal"
    phx-capture-click="close"
    phx-window-keydown="close"
    phx-key="escape"
    phx-target="#<%= @id %>"
    phx-page-loading>

    <div class="phx-modal-content">
      <%= live_patch raw("&times;"),
        to: @return_to, class: "phx-modal-close" %>
      <%= live_component @socket, @component, @opts %>
    </div>
  </div>
  """
end
```

Nice! We drop in the entire dialog modal in one short function. The markup in our modal component's `render/1` function is easy to understand, and easy to access. Since the component has just a few pieces of markup, the generator included this bit of HTML markup directly in the `render/1` function, rather than separating it out into a template file.

Let's take a moment to talk about the `assigns` argument with which the `render/1` function is called. The value of the `assigns` argument that is passed to `render/1` is the keyword list that was given as a third argument to the `PentoWeb.LiveHelpers.live_modal/3` function's call to `live_component/3`. Here's another look:

```

generate_web/pento/lib/pento_web/live/live_helpers.ex
def live_modal(socket, component, opts) do
  path = Keyword.fetch!(opts, :return_to)
  modal_opts = [
    id: :modal,
    return_to: path,
    component: component,
    opts: opts
  ]
  live_component(socket, PentoWeb.ModalComponent, modal_opts)
end

```

Taking a closer look at the markup implemented in the `render/1` function, you can see that the dialog is a mere `div` that contains a link and a call to render yet *another* component. We rely on a bit of CSS magic under the hood to show a modal form, and dim the elements in the background. Notice the `div` has a few `phx-` hooks (more on these in a bit) to pick up important events that are all ways to close our form. In this way, the component will receive events when the user clicks on certain buttons or presses certain keys. We'll look at that `close` event in more detail as we go.

Inside the `div`, you'll find a `live_patch/2` call to build a “close” link with the `:return_to` path. We passed in this `:return_to` option all the way back in the index template via the call to `live_modal/3`. You'll also see a call to `live_component/3` used to render the product form component. We'll take a closer look at this in an upcoming section.

That covers almost everything in the `ModalComponent` module. You might have expected to see a `mount/1` function. Let's find out why it's not there.

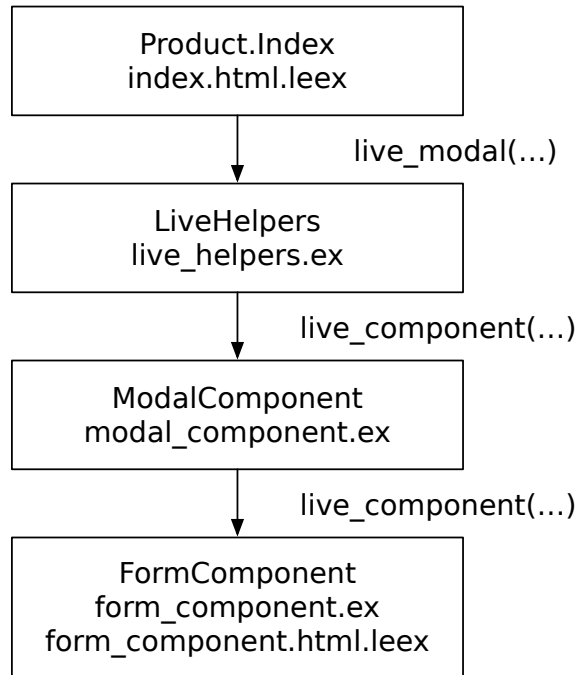
Mount the Modal Component

As you might imagine, a component has a lifecycle of its own. We'll get into the precise details in the next chapter. Here's an early hint. Components still support `mount/1` and `render/1`, but there are a few extra callbacks. Look out later on for a function called `update/2` that LiveView uses to update a component when needed. The default `mount/1` function just passes the socket through, unchanged. The default `update/2` function merely takes the assigns we call `live_component/3` with and passes them to the socket. Via the call to `use PentoWeb, :live_component`, the component implements and executes the default versions of these callbacks. We don't need to implement them ourselves unless we want to customize this behavior.

Our generated modal component doesn't need to keep any extra data in the socket, aside from the assigns we pass in via the call to `live_component/3`. That

means we can allow it to pick up the default mount/1 and update/2 functions from the behaviour. Our component therefore will implement only two functions itself—render/1 and handle_event/3.

Putting it all together in this figure, you can follow how the Product Index template ultimately renders the modal component:



The `ProductLive.Index` template calls a helper function, `live_component/3`, that in turn calls on the modal component. The modal component renders a template that presents a pop-up to the user. This template renders yet *another* component, the form component, via a call to `live_component/3`.

Now that you understand how the modal component is mounted and rendered, let's examine how it enacts a key part of LiveView's change management workflow—handling events from the user.

Handle Modal Component Events

There are two kinds of components, stateful and stateless. It won't surprise you to learn that only components that can implement a `handle_event/3` function can actually handle events. What you might not know is that only *stateful* components can implement `handle_event/3`. Adding an id to a component makes it stateful. We'll go into more detail on this topic in Part II and Part III. For now, let's keep going.

Here's a second look at the code that renders our component using the `LiveHelpers.live_modal/3` function in the `live_helpers.ex` file. Notice the `:id` key:

```
generate_web/pento/lib/pento_web/live/live_helpers.ex
def live_modal(socket, component, opts) do
  path = Keyword.fetch!(opts, :return_to)
  modal_opts = [
    id: :modal,
    return_to: path,
    component: component,
    opts: opts
  ]
  live_component(socket, PentoWeb.ModalComponent, modal_opts)
end
```

Now that you see that our modal component is in fact stateful, let's see how it is taught to handle events.

There are three ingredients to event management in LiveView.

First, we add a LiveView DOM Element binding, or LiveView binding, to a given HTML element. LiveView supports a number of such bindings that send events over the WebSocket to the live view when a specified client interaction, like a button click or form submit, occurs.

Then, we specify a target for that LiveView event by adding a `phx-target` attribute to the DOM element we've bound the event to. This instructs LiveView where to send the event—to the parent LiveView, the current component, or to another component entirely.

Lastly, we implement a `handle_event/3` callback that matches the name of the event in the targeted live view or component.

The modal component markup adds a few LiveView bindings to listen for close events: `phx-capture-click="close"`, `phx-window-keydown="close"`, and `phx-key="escape"`. This means that any of these client interactions, like clicking the “close” icon, will send an event with the name of “close” to the targeted live view. In this case, because the `phx-target` is set to the `@id` assignment, which is the id of our component, the modal component itself will receive the event.

That means the component must implement a `handle_event/3` function for the “close” event, which it does here:

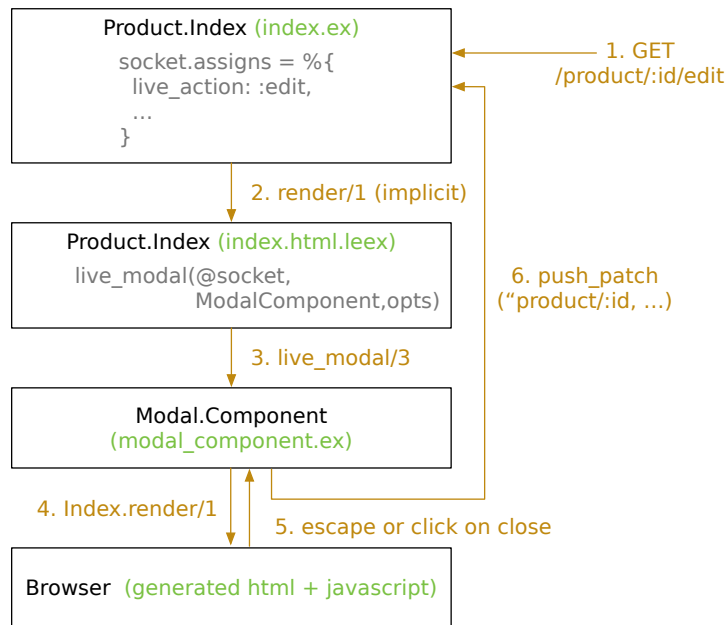
```
generate_web/pento/lib/pento_web/live/modal_component.ex
@impl true
def handle_event("close", _, socket) do
  {:noreply, push_patch(socket, to: socket.assigns.return_to)}
end
```

This generated event handler takes in arguments of the event name, ignored metadata, and the socket. Then, it *transforms* the socket by navigating back to the path we specified in `live_modal/3` with a call to `push_patch/2`. Let’s find out how that works now.

Live Navigation with `push_patch/2`

The `push_patch/2` function works just like the `live_patch/2` function we saw earlier being used in the Index template to link to the Product Edit view, with one exception. You’ll use `live_patch/2` in HTML markup running in the browser client. You’ll use `push_patch/2` in event handlers running on the server. It adds private data to the socket that LiveView’s server-side code and JavaScript will use to navigate between pages and manage the URL *all within the current LiveView*.

On the server side, the same change management lifecycle that we saw earlier will kick off. LiveView will call `handle_params/3`, but not `mount/3`. Let’s put it all together in this figure:



As the figure shows, when you click the “close” button, the browser navigates back to `/products`. That route will point us at `ProductLive.Index` with a `live_action` of `:index`. That change in state will cause another render of the index template. This time around, the template code’s if condition that checks for the `:edit` live action will evaluate to false, so LiveView will no longer render the modal.

Now that you're warmed up, let's take a look at the form component. It works mostly the same, but has a few more moving parts.

LiveView Layers: The Form Component

The job of the modal component was a simple one. It showed a modal pop up with whatever component we chose to put inside, and it knew how to close it. The content we put *inside* our modal window may have its own state, markup, and events, so we'll use another component, the form component. In this way, we are able to compose components into layers, each of which has a single responsibility. This is just one more way that the Phoenix Live generator provides us with an organized foundation for writing code that is capable of handling complex state, while being easy to maintain.

The form component is a bit more complex than the modal component. It allows us to collect the fields for a product a user wants to create or update. The form component will also have events related to submitting and validating the form.

Let's look at the form component in three steps: rendering the template, setting up the socket, and processing events.

Render the Form Component

We'll begin by tracing how the form component is rendered from within the modal component. This next bit of code flow is a little complex. Take your time to read through this section and let the provided code snippets guide you.

Remember, there are two kinds of components, stateful and stateless. Components with id keys are stateful; those without are stateless. It's been a while since we saw the code, but we actually specified the attributes for our form component within the `index.html.leex` template, like this:

```
generate_web/pento/lib/pento_web/live/product_live/index.html.leex
<%= if @live_action in [:new, :edit] do %>
  <%= live_modal @socket, PentoWeb.ProductLive.FormComponent,
    id: @product.id || :new,
    title: @page_title,
    action: @live_action,
    product: @product,
    return_to: Routes.product_index_path(@socket, :index)%>
  <% end %>
```

Notice there's an `:id` key, along with a `:component` key that specifies the `FormComponent` that will be rendered inside the modal. These attributes are passed into the modal component via `PentoWeb.LiveHelpers.live_modal/3`'s call to `live_component/3`.

```
generate_web/pento/lib/pento_web/live/live_helpers.ex
def live_modal(socket, component, opts) do
  path = Keyword.fetch!(opts, :return_to)
  modal_opts = [
    id: :modal,
    return_to: path,
    component: component,
    opts: opts
  ]
  live_component(socket, PentoWeb.ModalComponent, modal_opts)
end
```

The keyword list of options is made available to the modal component's `render/1` function as part of the the assigns. This means that the modal component's template has access to a `@component` assignment set equal to the name of the form component module.

Look at the call to `live_component/3` in the modal component's markup. This will mount and render the `FormComponent` and provide the additional options present in the `@opts` assignment.

```
<%= live_component @socket, @component, @opts %>
```

Since the `@opts` assignment includes a key of `:id`, the form component is stateful. It needs to be because it must process events to save and validate the form. Check the earlier call to the `live_modal/3` function from the Product Index template and you'll note that we also passed keys with a product, a title, the live action, and a path. All of those options, along with our `:id`, are in `@opts` and we can refer to them in the form component as part of the component's assigns.

Establish Form Component State

Now that you understand *how* the form component is being rendered, let's look at what happens *when* it is rendered.

The first time Phoenix renders the form component, it will call `mount/1` once. This is where we can perform any initial set-up for our form component's state. Then, the `update/2` callback will be used to keep the component up-to-date whenever the parent live view or the component itself changes. Because our generated component does not need a one-time setup, we won't see an explicit `mount/1` function at all. The default `mount/1` function from the call to use `PentoWeb`, `:live_component` will suffice.

The `update/2` function takes in two arguments, the map of assigns and the socket—both of which we provided when we called `live_component/3`. Here’s a refresher of that function call in the in-line LEEX template returned by the `ModalComponent`’s `render/1` function:

```
<%= live_component @socket, @component, @opts %>
```

The socket shared by the parent live view, in this case `ProductLive.Index`, is the first argument to `live_component/3`, followed by the name of the component to be rendered and the keyword list of options. These options are passed into the specified component’s `update/2` callback as the assigns argument, and the socket is passed in as the second argument.

Here’s what our `update/2` function looks like:

```
generate_web/pento/lib/pento_web/live/product_live/form_component.ex
@impl true
def update(%{product: product} = assigns, socket) do
  changeset = Catalog.change_product(product)

  {:ok,
   socket
   |> assign(assigns)
   |> assign(:changeset, changeset)}}
end
```

Let’s take a look at how this function uses the data in assigns to support the “product edit form” functionality now.

When you see a *form* anywhere in Phoenix, think *changing data*. As you saw in the previous chapter, change is represented with a *changeset*. The generated code uses the `Catalog.change_product/1` function to build a changeset for the product that is stored in assigns. Once again, you can see how the generated backend code is leveraged in the LiveView presentation layer.

All that remains is to take the socket, drop in all of the assigns that we passed through, and add in the new assignment for our changeset. With this, we’ve established the data for the form, and the component will go on to function just as other live views do. We will use handlers to wait for events, and then change the assigns in the socket in response to those events.

Let’s take a look at form component event handling now, starting with an exploration of the form component template.

Handle Form Component Events

In order to understand how the form component receives and handles events, we'll begin at the form component template and see how it sends events to the LiveView component.

Send Form Component Events

The form template is really just a standard Phoenix form. The main function in the template is the `form_for` function. We use the `changeset` that was put in assigns via the `update/2` callback:

```
generate_web/pento/lib/pento_web/live/product_live/form_component.html.leex
```

```
<%= f = form_for @changeset, "#",
  id: "product-form",
  phx_target: @myself,
  phx_change: "validate",
  phx_submit: "save" %>

  <%= label f, :name %>
  <%= text_input f, :name %>
  <%= error_tag f, :name %>

  <%= label f, :description %>
  <%= text_input f, :description %>
  <%= error_tag f, :description %>

  <%= label f, :unit_price %>
  <%= number_input f, :unit_price, step: "any" %>
  <%= error_tag f, :unit_price %>

  <%= label f, :sku %>
  <%= number_input f, :sku %>
  <%= error_tag f, :sku %>

  <%= submit "Save", phx_disable_with: "Saving..." %>
</form>
```

You can see the surrounding `form_for` function, with no target URL, an id, and three `phx-` attributes. This is what each of them do:

phx-change

Send the "validate" event to the live component each time the form changes

phx-submit

send the "save" event to the live component when the user submits the form

phx-target

Specify a component to receive these events. We specify `@myself` to send events to the current component

After the `form_for` function, you see some markup, a series of form fields, and a submit button. These tie back to the `@changeset` through the form variable, `f`. These form tags work exactly as they would for any live view. They will do two things. Upon rendering, they establish the value for each field. Upon submit, they send their values to the live view.

Notice also the error tags. These will come into play when a field is not valid based on the errors in the `changeset`.

You'll see more forms as this book unfolds. For now, let's move on to what happens when you change or submit a form.

Receive Form Component Events

Our form has two events. The `phx-change` event fires whenever the form changes—even a single character in a single field. The `phx-submit` event fires when the user submits the form, with a keystroke or with a button click. Here's what the form component's event handler for the "save" event looks like:

```
generate_web/pento/lib/pento_web/live/product_live/form_component.ex
def handle_event("save", %{"product" => product_params}, socket) do
  save_product(socket, socket.assigns.action, product_params)
end
```

The first argument is the event name. For the first time, we use the metadata sent along with the event, and we use it to pick off the form contents. The last argument to the event handler is the socket. When the user presses submit, the form component calls `save_product/3` which attempts either a product update or product create with the help of the Catalog context. If the attempt is successful, the component updates the flash messages and redirects to the Product Index view.

Let's take a look at this redirect in more detail.

Live Navigation with `push_redirect/2`

The `push_redirect/2` function, and its client-slide counterpart `live_redirect/2`, transform the socket. When the client receives this socket data, it will redirect to a live view, and will always trigger the `mount/3` function. It's also the only mechanism you can use to redirect to a *different* LiveView than the current one. Here's how it's called from within the form component:

```
socket
|> push_redirect(to: socket.assigns.return_to)}
```

Remember way back when we called `live_modal/3` from the `Index` template? That function was invoked with a set of options including a `:return_to` key set to a value of `/products`. That option was passed through the modal component, into the form component as part of the form component's socket assigns. So, we are redirecting to the *same* `Index` route we were already on. Because it's a `push_redirect/2` and not a `push_patch/2` however, *LiveView will trigger the `mount/3` function*. We want to ensure that `mount/3` re-runs now so that it can reload the product list from the database, grabbing and rendering any newly created products.

Putting it all together, you see how the form component is rendered within the `ProductLive.Index` live view, with state constructed from options passed in via the `ProductLive.Index` template, as well as additional form state set during the form component's own lifecycle. Then, when the form is submitted, the redirect causes the `Index` live view to re-render with fresh state for the `Index` view.

Your Turn

By tracing through the `ProductLive.Index` live view, you've seen the major pieces of the LiveView framework—the route, the live view module, the optional view template, and the helpers, component modules and component templates that support the parent view.

The entry point of the LiveView lifecycle is the route. The route matches a URL onto a LiveView module and sets a live action. The live view puts data in the socket using `mount/3` and `handle_params/3`, and then renders that data in a template with the same name as the live view. The `mount/render` and change management workflows make it easy to reason about state management and help you find a home for *all* of your CRUD code across just *two* live views.

When live views become too complex or repetitive, you can break off components. A `LiveComponent` compartmentalizes state, HTML markup, and event processing for a small part of a live view. The generators built two different components, one to handle a modal window and one to process a form.

All of this code demonstrates that LiveView provides an elegant system you can use to handle the complex interactions of a single-page app. LiveView empowers you to build highly interactive, real-time features in a way that is organized and easy to maintain. You could easily imagine adding custom features on top of the generated CRUD functionality, or applying the lessons of the generated code to your own hand-rolled live views.

Now that you're starting to see the beauty of LiveView as a single-page app system, it's time to get your hands dirty.

Give It a Try

These three problems are different in nature. You'll accomplish three tasks. The first, most straightforward one, is to trace through the `ProductLive.Show` live view.

Trace Through a Live View

Start from the `Index` page's implementation of the link to the product show page and work your way through the route, `mount/3`, `handle_params/3`, and `render/1` lifecycle. Answer these questions:

- Which route gets invoked when you click the link on the `Index` page to view a given product?
- What data does `Show.mount/3` add to the socket?
- How does the `ProductLive.Show` live view use the `handle_params/3` callback?
- How does the `ProductLive.Show` template render the Product Edit form and what events does that form support?

When you're done, display your own message on the page by adding some content to the `Show` live view's `socket.assigns` and then rendering it in the template.

Change the Index Live View

Let's tackle a more challenging problem now. You'll make a slight tweak to the `index.html.leex` live view.

- Instead of showing `<td><%= product.name %></td>` as a column containing the plain product name, make that name a link that brings the user to the product show page. Think about whether you need to reach for the `live_redirect/2` or `live_patch/2` function here.
- Next, remove the show link from the list of actions accompanying a given product table row.

Verify that you can click on a product, and that it navigates to the `Show` live view *without* reloading the page.

Generate Your Own LiveView

This final, more complex, task will ask you to combine everything you've learned in this and the previous chapter. You'll run the Phoenix Live generator again to create a new set of CRUD features for a resource, `FAQ`, or “frequently asked question”. This feature will allow users of our gaming site to submit questions, answer them, and up-vote them. Each `FAQ` should have fields for a question, an answer, and a vote count.

Devise your generator command and run it. Then, fire up the Phoenix server and interact with your generated FAQ CRUD features! Can you create a new question? Can you answer it? Trace some of the generated code pathways that support this functionality.

Next Time

In the next part of this book, we're ready to move away from generated code and roll our own LiveView from scratch. The following chapter will take a deep dive into working with LiveView forms and explore how changesets model changes to data in our live views, with and without database persistence. We'll finish with a look at an exciting and powerful LiveView feature—reactive file uploads. When we're done, you'll have built a new, custom live view, gained a solid understanding of how changesets and forms work together in LiveView, and be prepared to build interactive forms that meet a variety of user requirements. Let's go!

Part II

LiveView Composition

In Part II, you'll move away from generated code and start building your own live views from scratch. We'll dive into how LiveView lets you compose layers that manage change. We'll begin with a look at forms, and show you how to organize code for the changesets that model change in your live views. Then, we'll build a custom LiveView feature with components. Components allow programmers to partition and share views, including both presentation and event processing. First, we'll use a stateless component to share common presentation code while building a widget to collect demographic data from customers. Then, we'll extend that work with a stateful component, so the widget component can manage events.

Forms and Changesets

On the web, many user interactions are basic, such as typing a URL or clicking a link. But sometimes, users need more sophisticated interactions. HTML provides forms to represent complex data. In single-page apps, form interactions go beyond one-time submissions with a single response. Any user interaction could lead to immediate changes on a page. Today's users expect immediate feedback with clear error messages. Changing a country form field, for example, might impact available options in a state or province field. LiveView meets this need perfectly, giving us the opportunity to make adjustments to a page in real time, as the user fills out a form piece by piece.

Let's look at how these forms relate to the generated code you've seen so far.

The past few chapters focused on generated code, specifically database-backed live views. These generated web pages let users type data into Phoenix forms to change data in your database through Ecto-backed context and schema layers. Ecto changesets provide the connective tissue to weave these two disparate worlds together. In fact, Phoenix forms *are* representations of changesets in the user interface. This chapter will take a deeper dive into forms and changesets. You'll see how to compose LiveView code specifically, and Phoenix code generally, to manage change.

Model Change with Changesets

Before we get too deep into this topic, let's think about the role that forms and changesets play in our application.

First, consider Ecto changesets. Changesets are policies for changing data and they play these roles:

- Changesets *cast unstructured user data* into a known, structured form—most commonly, an Ecto database schema, ensuring data *safety*.

- Changesets *capture differences* between safe, consistent data and a proposed change, allowing *efficiency*.
- Changesets *validate data* using known consistent rules, ensuring data *consistency*.
- Changesets provide a *contract* for communicating error states and valid states, ensuring a *common interface for change*

You saw changesets in action in the `Product.changeset/2` function:

```
def changeset(product, attrs) do
  product
  |> cast(attrs, [:name, :description, :unit_price, :sku])
  |> validate_required([:name, :description, :unit_price, :sku])
  |> unique_constraint(:sku)
end
```

The `changeset/2` function captures differences between the structured product and the unstructured `attrs`.

Then, with `cast/4`, the changeset trims the attributes to a known field list and converts to the correct types, ensuring safety by guaranteeing that you don't let any unknown or invalid attributes into your database.

Finally, the `validate/2` and `unique_constraint/2` functions validate the inbound data, ensuring consistency.

The result is a data structure with known states and error message formats, ensuring interface compatibility.

Consequently, the forms in the `ProductLive` views knew exactly how to behave—validating form input and presenting errors in accordance with the changeset's rules in real-time, as the users typed. We didn't have to change the generated code much at all.

In this chapter, we're going to shift off of the well-known path of generated, database-backed changesets. You'll learn just how versatile changesets can be when it comes to modeling changes to data, with or without a database. You'll build a custom, schemaless changeset for data that *isn't* backed by a database table, and you'll use that changeset in a form within a live view. Along the way we'll explore some of the niceties LiveView provides for working with forms. Finally, we'll work with an exciting and powerful LiveView feature—live uploads. You'll use this feature to build an image uploader in LiveView. When we're done, you'll have built a custom live view, worked extensively with Ecto changesets, and seen the full power of LiveView forms.

Let's get started.

Model Change with Schemaless Changesets

We’ve used changesets to model changes to data that is persisted in our database, but we can easily imagine scenarios in which we want to present the user with the ability to input data that *isn’t* persisted. Consider the following examples:

- A guest checkout experience in which a user inputs their billing and shipping info without saving it.
- A gaming UI in which a user provides a temporary username for the lifespan of the game.
- A search form in which input is validated but not saved.

All of these scenarios require presenting some interface to the user for collecting input, validating that input, and managing the results of that validation. This is exactly what changesets and forms did for us in our `ProductLive` views. Luckily for us, we can continue to use changesets in this way, even without schema-backed modules and data persistence.

In this section, we’ll show you how to use schemaless changesets to model data that you won’t save in your database. You’ll build a new live view that uses schemaless changesets to allow users to send promo codes for game purchases to their friends. Then, we’ll take a look at some of the tools that `LiveView` provides for working with forms. Let’s dive in.

Build Schemaless Changesets from Structs

Simply put, you can use changesets with basic Elixir structs or maps—you don’t need to use Ecto schemas to generate those structs. But, when you do use changesets with plain structs, your code needs to provide the type information Ecto would normally handle.

That might sound confusing at first, but after a quick example, you’ll get the hang of it. All you need to do is call `Ecto.Changeset.cast/4`. For the first argument, you’ll pass a tuple containing your struct and a map of your struct’s attribute types, and you’re off to the races.

Let’s take a look at a brief example. Then, we’ll outline a use-case for schemaless changesets in our `Pento` app and build it out together.

Open up `IEx` and key in this simple module definition for a game player:

```
[pento] → iex -S mix
iex> defmodule Player do
  defstruct [:username, :age]
end
```

It's a plain old struct. Now, create a new, empty struct:

```
iex> player = %Player{}
```

We have a struct, and we've done nothing Ecto-specific. We want to use a changeset to cast and validate the data in our struct. Let's think a bit about how we can give the changeset enough information to do its job.

A typical changeset pipeline combines data, a cast, and validations. To successfully judge whether a change is consistent, the provided data *must* include information about both the changes and the type. So far, we've passed schema-backed structs as a first argument to `Ecto.Changeset.cast/4`. Such structs are produced by modules, like `Catalog.Product`, that implement a `schema/1` function containing information about the struct's allowed types. We don't have a schema struct though, so let's dig a bit deeper. Get help for the `Changeset.cast/4` function:

```
iex> h Ecto.Changeset.cast/4
```

```
def cast(data, params, permitted, opts \\ [])
```

```
Applies the given params...
```

```
The given data may be either a changeset, a schema struct or a {data, types} tuple. ...
```

This sentence is the key: “The given data may be either a changeset, a schema struct or a {data, types}”. We can start with a changeset or a schema struct, both of which embed data and type information. *Or* we can start with a two tuple that explicitly contains the data as the first element and provides type information as the second. Now, let's follow that advice and build a tuple with both a player struct and a map of types, like this:

```
iex> types = %{username: :string, age: :integer}
%{username: :string, age: :integer}
iex> attrs = %{name: "player1", age: 20}
%{name: "player1", age: 20}
iex> changeset = {player, types} \
  |> Ecto.Changeset.cast(attrs, Map.keys(types))
#Ecto.Changeset<changes: %{age: 20, ...}, ...,valid?: true>
```

Brilliant! This bit of code can create a changeset, but it's not too interesting unless we can also write validations. Let's say we have a game that can only be played by users who are over 16. We can add a validation like this:

```
iex> changeset = {player, types} \
  |> Ecto.Changeset.cast(attrs, Map.keys(types)) \
  |> Ecto.Changeset.validate_number(:age, greater_than: 16)
#Ecto.Changeset<...data: #Player<>,valid?: true>
```

We cast some data into a changeset, then pipe that changeset into a validation, and everything works. This code returns a valid changeset because we provided valid data according to our policy.

Let's see what happens if the `attrs` input is not valid:

```
iex> attrs = %{username: "player2", age: 15}
%{age: 15, username: "player2"}
iex> changeset = {player, types} \
  |> Ecto.Changeset.cast(attrs, Map.keys(types)) \
  |> Ecto.Changeset.validate_number(:age, greater_than: 16)
#Ecto.Changeset<
...
  errors: [age: {"must be greater than %{number}"},
  [validation: :number, kind: :greater_than, number: 16]]],...valid?: false>
```

Perfect. This changeset behaves just like the generated `Product` one. Piping a changeset with invalid data through the call to the `Ecto.Changeset` validation function returns an invalid changeset that contains errors. Next up, let's see how we can use schemaless changesets in a live view.

Use Schemaless Changesets in LiveView

To celebrate the one week anniversary of our wildly successful game company, we're offering a special promotion. Any registered user can log in and visit the `/promo` page. There, they can submit the email address of a friend and our app will email a promo code to that person providing 10% off of their first game purchase.

We'll need to provide a form for the promo recipient's email, but we *won't* be storing this email in our database. We don't have that person's permission to persist their personal data, so we'll use a schemaless changeset to cast and validate the form input. That way, the email layer will only send promotional emails to valid email addresses. Let's plan a bit.

We'll need a new `/promo` live view with a form backed by a schemaless changeset. The form will collect a name and email for a lucky 10% off promo recipient. Changeset functions are purely functional, so we'll build a model and some changeset functions in a tiny core. You'll notice that once we've coded up the schemaless changeset, the live view will work exactly the same way it always has, displaying any errors for invalid changesets and enabling the submit button for valid ones.

We'll start in the core. The `Promo.Recipient` core module will—you guessed it—model the data for a promo recipient. It will have a converter to produce the changeset that works with the live view's form. Then, we'll build a context

module, called `Promo`, that will provide an interface for interacting with `Promo.Recipient` changesets. The context is the boundary layer between our predictable core and the outside world. It is the home of code that deals with uncertainty. It will be responsible for receiving the uncertain form input from the user and translating it into predictable changesets. The context will also interact with potentially unreliable external services—in this case the code that sends the promotional emails. We won't worry about the email sending code. We'll keep our focus on changesets in `LiveView` and create a tiny stub instead.

Once we have the backend wired up, we'll define a live view, `PromoLive`, that will manage the user interface for our feature. We'll provide users with a form through which they can input the promo recipient's name and email. That form will apply and display any recipient validations we define in our changeset, and the live view will manage the state of the page in response to invalid inputs or valid form submissions.

Let's get started!

The Promo Boundary and Core

First up, we'll build the core of the promo feature. Create a file `lib/pento/promo/recipient.ex` and key in the following:

```
defmodule Pento.Promo.Recipient do
  defstruct [:first_name, :email]
end
```

Our module is simple so far. It implements a struct with two keys, `:first_name` and `:email`.

Next up, we need to give our module awareness of the types that will be considered valid by any changeset we create. Let's use a module attribute to store this map of types so that we can access it later:

```
defmodule Pento.Promo.Recipient do
  defstruct [:first_name, :email]
  @types %{first_name: :string, email: :string}
end
```

Now, we'll alias the module and import `Ecto.Changeset` so we can use the changeset functions:

```
defmodule Pento.Promo.Recipient do
  defstruct [:first_name, :email]
  @types %{first_name: :string, email: :string}

  alias Pento.Promo.Recipient
```

```
import Ecto.Changeset
end
```

Finally, we're ready to define the `changeset/2` function that will be responsible for casting recipient data into a changeset and validating it:

```
forms/pento/lib/pento/promo/recipient.ex
def changeset(%Recipient{} = user, attrs) do
  {user, @types}
  |> cast(attrs, Map.keys(@types))
  |> validate_required([:first_name, :email])
  |> validate_format(:email, ~r/@/)
end
```

We validate the presence of the `:first_name` and `:email` attributes, and then validate the format of `:email`.

Now we can create recipient changesets like this:

```
iex> alias Pento.Promo.Recipient
iex> r = %Recipient{}
iex> Recipient.changeset(r, %{email: "joe@email.com", first_name: "Joe"})
#Ecto.Changeset<...valid?: true>
```

Let's see what happens if we try to create a changeset with an attributes of an invalid type:

```
iex> Recipient.changeset(r, %{email: "joe@email.com", first_name: 1234})
#Ecto.Changeset<errors: [first_name: {"is invalid", ...}],valid?: false>
```

`Ecto.Changeset.cast/4` relies on `@types` to identify the invalid type and provide a descriptive error.

Next, try a changeset that breaks one of the custom validation rules:

```
iex> Recipient.changeset(r, %{email: "joe's email", first_name: "Joe"})
#Ecto.Changeset<changes: %{email: "joe's email", ...},
  errors: [email: {"has invalid format", ...}],valid?: false>
```

This function successfully captures our change policy in code, and the returned changeset tells the user exactly what is wrong.

Now that our changeset is up and running, let's quickly build out the `Promo` context that will present the interface for interacting with the changeset. Create a file, `lib/pento/promo.ex` and add in the following:

```
defmodule Pento.Promo do
  alias Pento.Promo.Recipient

  def change_recipient(%Recipient{} = recipient, attrs \\ %{}) do
    Recipient.changeset(recipient, attrs)
  end
end
```

```
def send_promo(recipient, attrs) do
  # send email to promo recipient
end
end
```

This context is a beautifully concise boundary for our service. The `change_recipient/2` function returns a recipient changeset and `send_promo/2` is a placeholder for sending a promotional email. Other than the internal tweaks we made inside `Recipient.changeset/2`, building the context layer with a schemaless changeset looks identical to building an Ecto-backed one. When all is said and done, in the view layer, schemaless changesets and schema-backed ones will look identical.

The Promo Live View

This live view will have the feel of a typical live view with a form. By this time, the development flow will look familiar to you. First, we'll create a simple route and wire it to the live view. Next, we'll use our Promo context to produce a schemaless changeset, and add it to the socket within a `mount/3` function. We'll render a form with this changeset and apply changes to the changeset by handling events from the form.

This section will move quickly, since you already know the underlying concepts. Create a file, `lib/pento_web/live/promo_live.ex` and fill in the following:

```
defmodule PentoWeb.PromoLive do
  use PentoWeb, :live_view
  alias Pento.Promo
  alias Pento.Promo.Recipient

  def mount(_params, _session, socket) do
    {:ok, socket}
  end
end
```

We pull in the `LiveView` behavior, alias our modules for later use and implement a simple `mount/3` function.

Let's use an implicit `render/1`. Create a template file in `lib/pento_web/live/promo_live.html.leex`, starting with some promotional markup:

```
forms/pento/lib/pento_web/live/promo_live.html.leex
<h2>Send Your Promo Code to a Friend</h2>
<h4>
  Enter your friend's email below and we'll send them a
  promo code for 10% off their first game purchase!
</h4>
```

Now, let's define a live route and fire up the server. In the router, add the following route behind authentication:

```
forms/pento/lib/pento_web/router.ex
scope "/", PentoWeb do
  pipe_through [ :browser, :require_authenticated_user ]
  live "/promo", PromoLive
```

Start up the server, log in, and point your browser at /promo. You should see the following:

Send Your Promo Code to a Friend

Enter your friend's email below and we'll send them a promo code for 10% off their first game purchase!

Everything is going according to plan. With the live view up and running, we're ready to build out the form for a promo recipient. We'll use mount/3 to store a recipient struct and a changeset in the socket:

```
forms/pento/lib/pento_web/live/promo_live.ex
def mount(_params, _session, socket) do
  { :ok,
    socket
    |> assign_recipient()
    |> assign_changeset() }
end

def assign_recipient(socket) do
  socket
  |> assign(:recipient, %Recipient{})
end

def assign_changeset(%{ assigns: %{ recipient: recipient } } = socket) do
  socket
  |> assign(:changeset, Promo.change_recipient(recipient))
end
```

The mount/3 function uses two helper functions, assign_recipient/1 and assign_changeset/1 to add a recipient struct and a changeset for that recipient to socket assigns. These pure, single-purpose, reducer functions are reusable building blocks for managing the live view's state.

Remarkably, the schemaless changeset can be used in our form exactly like database-backed ones. We'll use `socket.assigns.changeset` in the template's form, like this:

`forms/pento/lib/pento_web/live/promo_live.html.leex`

```
<%= f = form_for @changeset, "#",
  id: "promo-form",
  phx_change: "validate",
  phx_submit: "save" %>

<%= label f, :first_name %>
<%= text_input f, :first_name %>
<%= error_tag f, :first_name %>

<%= label f, :email %>
<%= text_input f, :email %>
<%= error_tag f, :email %>

<%= submit "Send Promo" %>
</form>
```

Our form implements two LiveView bindings, `phx-change` and `phx-submit`. Let's focus on the `phx-change` event first. LiveView will send a "validate" event each time the form changes, and include the form params in the event metadata. So, we'll implement a `handle_event/3` function for this event that builds a new changeset from the params and adds it to the socket:

`forms/pento/lib/pento_web/live/promo_live.ex`

```
def handle_event(
  "validate",
  %{"recipient" => recipient_params},
  %{assigns: %{recipient: recipient}} = socket) do
  changeset =
    recipient
    |> Promo.change_recipient(recipient_params)
    |> Map.put(:action, :validate)

  {:noreply,
   socket
   |> assign(:changeset, changeset)}
end
```

This code should look familiar to you; it's almost exactly what the generated `ProductLive.FormComponent` did. The `Promo.change_recipient/2` context function creates a new changeset using the recipient from state and the params from the form change event.

Then, we use `Map.put(action, :validate)` to add the validate action to the changeset, a signal that instructs Phoenix to display errors. Phoenix otherwise will *not* display the changeset's errors. When you think about it, this approach makes

sense. Not all invalid changesets should show errors on the page. For example, the empty form for the new changeset *shouldn't* show any errors, because the user hasn't provided any input yet. So, the Phoenix `form_for` function needs to be told when to display a changeset's errors. If the changeset's action is empty, then no errors are set on the form object—even if the changeset is invalid and has a non-empty `:errors` value.

Finally, `assigns/2` adds the new changeset to the socket, triggering `render/1` and displaying any errors. Let's take a look at the form tag that displays those errors on the page. Typically, each field has a label, an input control, and an error tag, like this:

```
<%= label f, :email %>
<%= text_input f, :email%>
<%= error_tag f, :email %>
```

The `error_tag/2` Phoenix view helper function displays the form's errors for a given field on a changeset, when the changeset's action is `:validate`.

Let's try it out. Point your browser at `/promo` and fill out the form with a name and an invalid email. As you can see in this image, the UI updates to display the validation errors:

Send Your Promo Code to a Friend

Enter your friend's email below and we'll send them a promo code for 10% off their first game purchase!

First name

Email

has invalid format

SEND PROMO

That was surprisingly easy! We built a simple and powerful live view with a reactive form that displays any errors in real-time. The live view calls on the context to create a changeset, renders it in a form, validates it on form change, and then re-renders the template after each form event. We get reactive form validations for free, without writing any JavaScript or HTML. We let Ecto changesets handle the data validation rules and we let the LiveView framework

handle the client/server communication for triggering validation events and displaying the results.

As you might imagine, the `phx-submit` event works pretty much the same way. The "save" event fires when the user submits the form. We can implement a `handle_event/3` function that uses the (currently empty) context function, `Promo.send_promo/2`, to respond to this event. The context function should create and validate a changeset. If the changeset is in fact valid, we can pipe it to some helper function or service that handles the details of sending promotional emails. If the changeset is not valid, we can return an error tuple. Then, we can update the UI with a success or failure message accordingly. We'll leave building out this flow as an exercise for the reader.

Now you've seen that while Ecto changesets are delivered with Ecto, they are not tightly coupled to the database. Schemaless changesets let you tie backend services to Phoenix forms any time you require validation and security, whether or not your application needs to access a full relational database.

Before we move on to our last LiveView form feature, the live uploader, let's take a quick look at some additional LiveView form bindings.

LiveView Form Bindings

You already know that LiveView uses annotations called bindings to tie live views to events using platform JavaScript. This chapter demonstrated the use of two form bindings: `phx-submit` for submitting a form and `phx-change` for form validations.

LiveView also offers bindings to control how often, and under what circumstances, LiveView JavaScript emits form events. These bindings can disable form submission and *debounce*, or slow down, form change events. These bindings help you provide sane user experiences on the frontend and ensure less unnecessary load on the backend.

Let's take a brief look at these bindings and how they work.

Submit and Disable a Form

By default, binding `phx-submit` events causes three things to occur on the client:

- The form's inputs are set to "readonly"
- The submit button is disabled
- The "phx-submit-loading" CSS class is applied to the form

While the form is being submitted, no further form submissions can occur, since LiveView JavaScript disables the submit button. You can further cus-

tomize this behavior. The `phx-disable-with` binding lets you configure the text of a disabled submit button. Let's try it out now.

Open up `lib/pento_web/live/promo_live.html.leex` and add a `phx-disable-with` attribute:

```
<%= submit "Send Promo", phx_disable_with: "Sending promo..." %>
```

Normally, our form submission happens so quickly that you won't really notice this disabled form state and updated submit button text. Slow it down by adding a 1 second sleep to the save event in `promo_live.ex`, like this:

```
def handle_event("save", %{"recipient" => recipient_params}, socket) do
  :timer.sleep(1000)
  # ...
end
```

Now, point your browser at `/promo` and submit the form. You should see the disabled form with our new button text:



Send Your Promo Code to a Friend

Enter your friend's email below and we'll send them a promo code for 10% off their first game purchase!

First name

Email

Nice! Once again, the LiveView framework handles the details for us—doing the work of disabling the form submit button and applying the new button text.

Next up, we'll take a look at a couple of bindings to control rapidly repeating form events.

Rate Limit Form Events

LiveView makes it easy to rate limit events on the client with the `phx-debounce` and `phx-throttle` LiveView bindings. You'll use `phx-debounce` when you want to rate limit form events, like `phx-change`, for a single field.

By default, our promo form will send a `phx-change` event *every time* the form changes. As soon as a user starts typing into the email input field, LiveView JavaScript will start sending events to the server. These events trigger the event handler for the "validate" event, which validates the changeset and renders any errors.

Let's think through what this means for the user.

If a user visits `/promo` and types even just one letter into the email field, then the error message describing an invalid email will immediately appear, as in this image:

Send Your Promo Code to a Friend

Enter your friend's email below and we'll send them a promo code for 10% off their first game purchase!

First name

Email

has invalid format

SEND PROMO

This provides a somewhat aggressive user experience and creates a situation in which both the client and the server will have to process a lot of information quickly. Let's fix this by giving our users a chance to type the entire email into the field before validating it. We can do so with the help of `phx-debounce`. The `phx-debounce` binding lets you specify an integer timeout value or a value of `blur`. Use an integer to delay the event by the specified number of milliseconds. Use `blur` to have LiveView JavaScript emit the event when the user finishes and tabs away from the field.

Let's use `debounce` to delay the firing of the `phx-change` event until a user has blurred the email input field:

```
<%= text_input f, :email, phx_debounce: "blur" %>
```

Now, if you visit `/promo` and type just one letter into the email field, the error message will not appear prematurely.

Send Your Promo Code to a Friend

Enter your friend's email below and we'll send them a promo code for 10% off their first game purchase!

First name

Email

SEND PROMO

If you blur away from the email input field, however, you will see the error message.

Now you know almost everything that you can do with forms in LiveView. Before we go, there's one more LiveView form feature you'll need to master—live uploads.

Live Uploads

The LiveView framework supports the most common features single-page apps must offer their users, including multipart uploads. LiveView can give us highly interactive file uploads, right out of the box.

In this section, you'll add a file upload feature to your application. You'll use LiveView to display upload progress and feedback while editing and saving uploaded files. When we're done, you'll have all the tools you need to handle complex forms, even those that require file uploads.

We'll add file uploads to the ProductLive form so users can choose an image to upload and associate with the product in a database. Let's plan this new feature first. We'll start on the backend by adding an `image_upload` field to the table and schema for products. Then, we'll update the `ProductLive.FormComponent` to support file uploads. Finally, the live view should report on upload progress and other bits of upload feedback.

Let's get started!

Persist Product Images

We'll start in the backend by updating the products table and Product schema to store an attribute `image_upload`, pointing to the location of the uploaded file. Once we have our backend wired up, we'll be able to update our live view's form to accommodate file uploads.

We'll start at the database layer by generating a migration to add a field, `:image_upload` to the products table.

First, generate your migration file:

```
[pento] → mix ecto.gen.migration add_image_to_products
* creating priv/repo/migrations/20201231152152_add_image_to_products.exs
```

This creates a migration file for us, `priv/repo/migrations/20201231152152_add_image_to_products.exs`. Open up that file now and key in the contents to the change function:

```
forms/pento/priv/repo/migrations/20201231152152_add_image_to_products.exs
defmodule Pento.Repo.Migrations.AddImageToProducts do
  use Ecto.Migration

  def change do
    alter table(:products) do
      add :image_upload, :string
    end
  end
end
```

This code will add the new database field when we run the migration. Let's do that now:

```
[pento] → mix ecto.migrate
[info] == Running 20201231152152 \
Pento.Repo.Migrations.AddImageToProducts.change/0 forward
10:22:24.034 [info] alter table products
10:22:24.041 [info] == Migrated 20201231152152 in 0.0s
```

This migration added a new column `:image_upload`, of type `:string`, to the products table, but our schema still needs attention.

Update the corresponding Product schema by adding the new `:image_upload` field to the schema function, like this:

```
forms/pento/lib/pento/catalog/product.ex
schema "products" do
  field :description, :string
  field :name, :string
  field :sku, :integer
```

```

field :unit_price, :float
field :image_upload, :string
timestamps()

```

Remember, the `changeset` `cast/4` function must explicitly whitelist new fields, so make sure you add the `:image_upload` attribute:

```
forms/pento/lib/pento/catalog/product.ex
```

```

def changeset(product, attrs) do
  product
  |> cast(attrs, [:name, :description, :unit_price, :sku, :image_upload])
  |> validate_required([:name, :description, :unit_price, :sku])
  |> validate_number(:unit_price, greater_than: 0)
  |> unique_constraint(:sku)
end

```

We don't need to add any validations for a product's image upload. We simply add `:image_upload` to `cast/4` and that's it.

Now that the `changeset` has an `:image_upload` attribute, we can save product records that know their image upload location. With that in place, we can make an image upload field available in the `ProductLive.FormComponent`'s form. We're one step closer to giving users the ability to save products with images.

Let's turn our attention to the component now.

Allow Live Uploads

We'll see our updated product `changeset` in action in a bit. First, we need to update the product form to support file uploads. Recall that both the "new product" and "edit product" pages are backed by the `ProductLive.FormComponent`. This provides one centralized place to maintain our product form. Changes to this component will therefore mean that we are enabling users to upload an image for a new product as well as for a product they are editing.

In order to enable uploads for our component, or any live view, we need to call the `allow_upload/3` function with an argument of the socket. This will put the data into socket assigns that the `LiveView` framework will then use to perform file uploads. So, for a component, we'll call `allow_upload/3` when the component first starts up and establishes its initial state in the `update/2` function. For a live view, we'd call `allow_upload/3` in the `mount/3` function.

The `allow_upload/3` function is a reducer that takes in an argument of the socket, the upload name, and the upload options and returns an annotated socket. Supported options include file types, file size, number of files per upload name, and more. Let's see it in action:


```
forms/pento/lib/pento_web/live/product_live/form_component.ex
def update(%{product: product} = assigns, socket) do
  changeset = Catalog.change_product(product)
  {:ok, socket
    |> assign(assigns)
    |> assign(:changeset, changeset)
    |> allow_upload(:image,
      accept: ~w(.jpg .jpeg .png),
      max_entries: 1,
      auto_upload: true,
      progress: &handle_progress/3
    )}
end
```

In `allow_upload/3`, we pipe in a socket and specify a name for our upload, `:image`. We provide some options, including the maximum number of permitted files, a progress function (more on that later), and an `auto_upload` setting of `true`. Setting this option tells LiveView to begin uploading the file as soon as a user attaches it to the form, rather than waiting until the form is submitted.

Let's take a look at what our socket assigns looks like after `allow_upload/3` is invoked:

```
%{
  # ...
  uploads: %{
    __phoenix_refs_to_names__: %{"phx-FlZ_j-hPIdCQuQGG" => :image},
    image: #Phoenix.LiveView.UploadConfig<
      accept: ".jpg,.jpeg,.png",
      auto_upload?: true,
      entries: [],
      errors: [],
      max_entries: 1,
      max_file_size: 8000000,
      name: :image,
      progress_event: #Function<1.71870957/3 ...>,
      ref: "phx-FlZ_j-hPIdCQuQGG",
      ...
    >
  }
}
```

The socket now contains an `:uploads` map that specifies configuration for each upload field your live view allows. We allowed uploads for an upload called `:image`. So, our map contains a key of `:image` pointing to a value of the configuration constructed using the options we gave `allow_upload/3`. This means that we can add a file upload field called `:image` to our form, and LiveView will track the progress of files uploaded via the field within `socket.assigns.uploads.image`.

You can call `allow_upload/3` multiple times with different upload names, thus allowing any number of file uploads in a given live view or component. For example, you could have a form that allows a user to upload a main image, a thumbnail image, a hero image, and more.

Now that we've set up our uploads state, let's take a closer look at the `:image` upload configuration.

Upload Configurations

The `:image` upload config looks something like this:

```
#Phoenix.LiveView.UploadConfig<
  accept: ".jpg,.jpeg,.png",
  auto_upload?: true,
  entries: [],
  errors: [],
  max_entries: 1,
  max_file_size: 8000000,
  name: :image,
  progress_event: #Function<1.71870957/3 ...>,
  ref: "phx-FLZ_j-hPIIdCQuQGG",
  ...
>
```

Notice that it contains the configuration options we passed to `allow_upload/3`: the accepted file types list, the auto upload setting, and the progress function, among other things.

It also has an attribute called `:entries`, which points to an empty list. When a user uploads a file for the `:image` form field, LiveView will automatically update this list with the file upload entry as it completes.

Similarly, the `:errors` list starts out empty and will be automatically populated by LiveView with any errors that result from an invalid file upload entry.

In this way, the LiveView framework does the work of performing the file upload and tracking its state for you. We'll see both of these attributes in action later on in this chapter.

Now that we've allowed uploads in our component, we're ready to update the template with the file upload form field.

Render The File Upload Field

You'll use the function `live_file_input/2` to generate the HTML for a file upload form field. Open up the form component's template and add the following:

```
forms/pento/lib/pento_web/live/product_live/form_component.html.leex
<%= live_file_input @uploads.image %>
```

Remember, `socket.assigns` has a map of uploads. Here, we provide `@uploads.image` to `live_file_input/2` to create a form field with the right configuration, and tie that form field to the correct part of socket state. This means that LiveView will update `socket.assigns.uploads.image` with any new entries or errors that occur when a user uploads a file via this form input.

The live view can present upload progress by displaying data from the `@uploads.image.entries` and `@uploads.image.errors`. LiveView will handle all of the details of uploading the file and updating socket assigns `@uploads.image` entries and errors for us. All we have to do is render the data that is stored in the socket. We'll take that on bit later.

Now, if you point your browser at `/products/new`, you should see the file upload field displayed like this:



New Product ✕

Name

Description

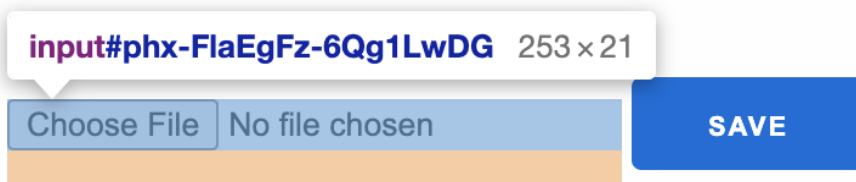
Unit price

Sku

Image

No file chosen

And if you inspect the element, you'll see that the `live_file_input/2` function generated the appropriate HTML:



Elements	Console	Sources	Network	Performance	Memory	Application	Security	Lighthouse	Adblock Plus	EditTt
<pre> <input id="product_name" name="product[name]" type="text"> <label for="product_description">Description</label> <input id="product_description" name="product[description]" type="text"> <label for="product_unit_price">Unit price</label> <input id="product_unit_price" name="product[unit_price]" step="any" type="number"> <label for="product_sku">Sku</label> <input id="product_sku" name="product[sku]" type="number"> <input id="product_image_upload" name="product[image_upload]" type="hidden"> <label for="product_image">Image</label> <input accept=".jpg,.jpeg,.png" data-phx-active-refs data-phx-done-refs data-phx-preflighted-refs data-phx-update="ignore" data-phx-upload-ref="phx-FlaEgFz-6Qg1LwDG" id="phx-FlaEgFz-6Qg1LwDG" name="image" phx-hook="Phoenix.LiveFileUpload" type="file"> == \$0 <button phx-disable-with="Saving..." type="submit">Save</button> </form> </pre>										

You can see that the generated HTML has the `accept=".jpg,.jpeg,.png"` attribute set, thanks to the options we passed to `allow_upload/3`.

LiveView also supports drag-and-drop file uploads. All we have to do is add an element to the page with the `phx-drop-target` attribute. Let's do that now:

```

forms/pento/lib/pento_web/live/product_live/form_component.html.leex
<div phx-drop-target="<%= @uploads.image.ref %>">
  <%= live_file_input @uploads.image %>
</div>

```

We give the attribute a value of `@uploads.image.ref`. This socket assignment is the ID that LiveView JavaScript uses to identify the file upload form field and tie it to the correct key in `socket.assigns.uploads`. So now, if a user clicks the “choose file” button *or* drags-and-drops a file into the area of this div, LiveView will upload the file and track its progress as part of `socket.assigns.uploads.image`.

As with other form interactions, LiveView automatically handles the client/server communication. Depending on the `auto_upload` setting, LiveView will upload the file when the user attaches the file or when the user submits the form. Since we specified an `auto_upload` setting of `true`, LiveView will start uploading the file as soon as it is attached. All we have to do is implement an event handler to respond to the file upload progress event. Let's do that now.

Handle Upload Progress

LiveView sends a progress event for each bit of data transfer until all the images are done uploading. Earlier, when we called `allow_upload/3`, we specified the callback function that should be invoked to handle this event: `handle_progress/3`. Let's write that function now.

```

defp handle_progress(:image, entry, socket) do
  if entry.done? do
    path =
      consume_uploaded_entry(
        socket,

```

```

    entry,
    &upload_static_file(&1, socket)
  )

  {:noreply, socket}
else
  {:noreply, socket}
end
end

```

The `handle_progress/3` callback takes in three arguments—the name of the upload field, the file being uploaded, and the socket. Let's dig into this code a bit further. Our function checks to see if the entry is done uploading with the help of a call to `entry.done?`. If it is done, we call the `consume_uploaded_entry/3` function with the socket, the entry, and a callback function that does something with the file. This callback function would be the perfect place to add other processing, like uploading the file to a Amazon S3 bucket.¹ We won't be integrating with any such third-party services here though. For now, we'll simply write the uploaded file to our app's static assets in `priv/static/images` so that we can display it on the product show template later on.

So, our `upload_static_file/2` callback function is pretty straightforward:

```

forms/pento/lib/pento_web/live/product_live/form_component.ex
defp upload_static_file(%{path: path}, socket) do
  # Plug in your production image file persistence implementation here!
  dest = Path.join("priv/static/images", Path.basename(path))
  File.cp!(path, dest)
  Routes.static_path(socket, "/images/#{Path.basename(dest)}")
end

```

It is invoked by `consume_uploaded_entry/3` with the processed file upload. It writes the file to `priv/static/images` and returns the resulting file path.

The last step is to ensure that this file path gets saved to the product as its `:image_upload` attribute when the form is submitted later. We updated our `Product.changeset` function earlier to cast the `:image_upload` attribute. We'll take advantage of that now.

The `handle_progress/3` function will take the newly returned static file path and use it update the changeset stored in socket assigns:

```

forms/pento/lib/pento_web/live/product_live/form_component.ex
defp handle_progress(:image, entry, socket) do
  # :timer.sleep(1000)
  if entry.done? do
    path =

```

1. <https://www.poeticoding.com/aws-s3-in-elixir-with-exaws/>

```

    consume_uploaded_entry(
      socket,
      entry,
      &upload_static_file(&1, socket)
    )

    {:noreply,
     socket
    }
    |> put_flash(:info, "file #{entry.client_name} uploaded")
    |> update_changeset(:image_upload, path)}
  else
    {:noreply, socket}
  end
end
end

```

Taking a closer look at the `update_changeset/3` function, you can see here that it uses the `Ecto.Changeset.put_change/3` function to produce a new changeset with the `:image_upload` attribute set and returns a socket containing this new changeset:

```

forms/pento/lib/pento_web/live/product_live/form_component.ex
def update_changeset(%{assigns: %{changeset: changeset}} = socket, key, value) do
  socket
  |> assign(:changeset, Ecto.Changeset.put_change(changeset, key, value))
end

```

Finally, we'll need to make sure that the `:image_upload` attribute from this changeset becomes part of the form params when the form is submitted so that this attribute will be applied to any new or edited product. We'll add a hidden field to the form that contains this data point:

```

forms/pento/lib/pento_web/live/product_live/form_component.html.leex
<%= hidden_input f, :image_upload %>

```

Now, when the `phx-change` or `phx-submit` form event fires, the form params sent to the component will include the `:image_upload` key/value pair and that data will be applied to the product record along with the rest of the data in params.

In order to see our code in action, let's add some markup to the product show to display image uploads. Then, we'll try out our feature.

Display Image Uploads

Open up `lib/pento_web/product/show.html.leex` and add the following markup to display the uploaded image or a fallback:

```

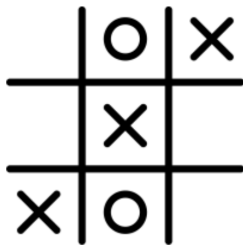
<article class="column">
  ">
</article>
<!-- product details... -->

```

Perfect. Now, we can test drive this fine new machine. Visit `/products/1/edit`, and upload a file:

Once you submit the form, you'll see the show page render the newly uploaded image, like this:

Show Product



- **Name:** Tic-Tac-Toe
- **Description:** The game of Xs and Os
- **Unit price:** 3.0
- **Sku:** 11121314

[EDIT](#) [Back](#)

We did it! Yet again, the LiveView framework handled all of the details of the client/server communication that makes the page interactive. LiveView performed the file upload for you, and made responding to upload events easy and customizable. All you needed to do was tell the live view which uploads

to track and what to do with uploaded files. Then, you added the file upload form field to the page with the view helper and LiveView handled the rest!

There's one last thing to do. Earlier, we promised *reactive* file uploads that share feedback with the user. Let's take a look now.

Display Upload Feedback

We know that LiveView automatically updates the `:entries` and `:errors` lists in the uploads config portion of `socket.assigns` once the upload begins. Let's display this information in the template to give the user real-time progress tracking. The code is amazingly simple. We'll iterate over `@uploads.image.entries` to display the progress for each entry:

```
<%= for entry <- @uploads.image.entries do %>
  <p>
    <%= entry.client_name %> - <%= entry.progress %>%
  </p>
<% end %>
```

Uploads happen pretty quickly, so you might not notice this progress info appear on the page. Add a `:timer.sleep(1000)` to the top of your `handle_progress/3` function, and then upload a file. You should see the progress tracking tick up from 0% to 100%, displaying progress at any given moment in time like this:



The screenshot shows a web form titled "New Product" with a close button (X) in the top right corner. The form contains four input fields: "Name", "Description", "Unit price", and "Sku". Below the "Sku" field, there is a file upload section with a "Choose File" button and a file name "Sample-ima...ownload.png". A blue "SAVE" button is located below the file upload section. At the bottom of the form, there is a progress bar for the file upload, labeled "Sample-Image-file-Download.png - 16%".

LiveView handled the work of tracking the changes to the image entry's progress. All we had to do was display it.

You can use a similar approach to iterate over and display any errors stored in `@uploads.image.errors`, and you'll get a chance to do exactly that at the end of this chapter. You'll find that you don't have to do any work to validate files

and populate errors. LiveView handles those details. All you need to do is display any errors based on the needs of your user interface.

There's more that LiveView file uploads can do. LiveView makes it easy to cancel an upload, upload multiple files for a given upload config, upload files directly from the client to a cloud provider, and more. Check out the LiveView file upload documentation² for details.

This chapter has been brief but dense, so it's time to wrap up.

Your Turn

LiveView supports custom integration of forms to backend code with schemaless changesets. To do so, you need only replace the first argument to `Changeset.cast/4` with a two tuple holding both data and type information. This type of code is ideal for implementing form scenarios requiring validation but without the typical database backend.

Whether you're working with schema-backed or schemaless changesets, LiveView provides features to throttle events for a smoother user experience and better performance on the backend.

In addition to these powerful, flexible form features, LiveView enables reactive file uploads, right out of the box. Without writing any JavaScript, or even any custom HTML, you can build interactive file upload forms directly into your live view. LiveView handles the details of client/server communication and upload state management, leaving you on the hook for writing a very small amount of custom code to specify how you uploads should behave and how uploaded files should be saved. This is a pattern we'll see again and again in LiveView—the framework handles the communication and state management details of our SPA and we can focus on writing application-specific code to support our features.

Now, take the time to put these ideas into practice.

Give It a Try

These three exercises will help you master a few different principles. First, you'll work with changesets in a traditional database-backed form. Then, we'll provide an exercise to use schemaless changesets on your own. Finally, you'll get to customize file uploads.

2. https://hexdocs.pm/phoenix_live_view/uploads.html#content

Add a Custom Validation

This simple task will give you a chance to practice working with changesets in LiveView.

First, add a custom validation to the Product schema's changeset that validates that `:unit_price` is greater than 0.00.

Then, visit `/products/new` and try to create a new product with an invalid unit price.

What happens when you start typing into the unit price field? What happens if you submit the form with an invalid unit price? Can you trace through the code flow for each of these scenarios and identify when and how the template is updated to display the validation error?

Use Schemaless Changesets

This second, more complex, exercise requires you to build out a new live view, backed by a schemaless changeset, from scratch.

Define a new live view, `Pento.SearchLive`, that lives at a route, `/search`. This live view should present a user with a search form allowing them to search products by SKU, and only by SKU. Assuming that all product SKUs have at least 7 digits, ensure that the form validates the SKU input and displays errors when provided with an invalid SKU. Use a schemaless changeset to build this form and enact these validations.

Customize Your File Uploader

This last task provides a deeper dive into the LiveView file upload feature. You'll make our existing product image uploader even more reactive and communicative to the user.

First, update the `ProductLive.FormComponent`'s template to display any upload errors. How will you ensure that the error messages are user-friendly and tied to the correct upload entry?

Once you're successfully displaying errors, try to drag-and-drop a file of an invalid type (a PDF for example). If you did your job correctly, you should see the appropriate error message on the page. Then, try to upload another file of the correct type. Since our call to `allow_upload/3` specified a `:max_entries` option of 1, you'll run into the "too many files" error. So, you'll need to provide your

user with a way to cancel any stuck, errored uploads *before* uploading again. Implement an upload cancel feature using the docs here.³

Finally, if you have an Amazon S3 account, upload your image to S3.

Next Time

In the next chapter, we'll build on what we've learned about forms to construct a layered live view that manages the state of a multi-stage form. We'll create a user survey feature that asks users to rate our games. Along the way, we'll take a deep dive into LiveView components. You'll learn how to compose LiveView pipelines for elegant state management and design your own set of LiveView layers to handle complex user interactions. Let's get going!

3. https://hexdocs.pm/phoenix_live_view/Phoenix.LiveView.html#cancel_upload/3

Stateless Components

At every level of difficulty, writing good code depends on breaking complex problems into several simpler ones. As yet, we haven't built any very complex live views. That changes in this chapter. We'll exercise the tools we've explored so far to build a complex live view with a multi-stage form, and you'll build your own components from scratch to help you manage this complexity. We'll begin building a simple survey tool, one with both demographic and rating information.

Along the way, we'll focus specifically on use-cases that require components, both stateless and stateful. In this chapter, you'll create your own stateless component that you'll layer into a parent live view. Stateless components allow the extraction of common *rendering* code. You'll use them to wrap up re-usable markup. We'll start by building a multi-stage form in which the state of the survey changes to progressively reveal more and more questions depending on the user's input. In the following chapter, we'll take our survey to the next level. We'll show you how user interfaces interact with state and events and introduce a the concept of stateful components that encapsulate not just markup, but also behavior.

While the survey itself is simple, it represents the most complex functionality you'll have seen so far. When you're done building it, you'll be able to orchestrate a set of LiveView components to cleanly handle even the most complex interactive, real-time features in your Phoenix app.

Back in [Chapter 3, Generators: Contexts and Schemas, on page 57](#), we promised that you'd be programming LiveView like a professional. That means that we won't just take you on a whirlwind tour of stateless components in this chapter. Instead, we'll build our survey feature from the ground up, starting with the schema and context our survey live view will need. This will

give you another opportunity to practice good code organization and it's in-line with how you'll build live views on your own, in the future.

It's going to be an exciting two chapters, so let's get started.

The Survey

Great companies know what their customers think, and Pento should be no different. We'd like to build a survey tool. We want to be able to track what our customers think about us over time, and our data scientists want to be able to slice and dice those results by several important demographics.

A sure way to irritate our customers is to ask the same demographic questions each time, so we'll ask demographic questions *once*. Then, we can ask a few short questions multiple times, and track those responses over time.

To satisfy these requirements, we'll build a survey feature that asks a user to fill out a survey to review our products. The survey will consist of a *demographics* section in which we ask a user to fill out a few basic questions about themselves. Then we will ask the user to rate each product on a scale of one to five stars. Logged-in users will be able to visit `/survey` and fill out the survey.

Our survey will be dynamic. First, it will prompt the user to fill out the demographics section. Only when that section has been successfully completed will we reveal the product rating sections. Here's how it will work.

- When no demographic exists for the user, we will show just the demographic portion of the survey, like this:

Survey

Demographics

Gender

female ▼

Year of birth

2020 ▼

SAVE

- When the demographic portion of the survey is complete, we will show demographic details and the product ratings portion of the survey, like this:

Survey

Demographics ✓

Gender: female

Year of birth: 2003

Ratings

Chess

Stars

5 ▼

SAVE

Tic-Tac-Toe

Stars

5 ▼

SAVE

Table Tennis

Stars

5 ▼

SAVE

- For any product ratings that are complete, we will display rating details, like this:

Survey

Demographics ✓

Gender: female

Year of birth: 2003

Ratings

Chess: ★ ★ ★ ★ ★

Tic-Tac-Toe

Stars

Table Tennis

Stars

When all ratings are complete, we will show the completed survey, like this image shows:

Survey

Demographics 

Gender: female

Year of birth: 2003

Ratings 

Chess:     

Tic-Tac-Toe:     

Table Tennis:     

The dynamic nature of the survey gently guides the user through a multi-page form and shows them exactly what they need to see, exactly when they need to see it. This approach adds a bit of complexity to our application, but you'll see that LiveView gives us the tools we need to manage this complexity with ease.

We'll begin by building the backend context and schemas that support the survey. Then, we'll move onto the frontend. We'll set up the live view and use a component to compartmentalize the demographic portion of the survey's markup and behavior. When we're done, you'll have a firm understanding of when to reach for stateless components and when to reach for stateful components.

With a plan placed firmly in our pocket, let's take another major look at the main feature we need to use—components.

Organize Your LiveView with Components

Let's think through the design considerations for our survey. We may eventually want to display the survey in several different places on the site. You

could imagine, for example, wanting to place *just* the product rating portion of the survey on the show page for a given product, or *just* the demographic details portion on some sort of user profile page. And, as we've seen, the dynamic nature of the survey represents a decent amount of complexity.

Both of these considerations push us toward components. Having a dedicated place to put the code related to each portion of our survey will allow us to share these concepts across the site. Also, we've said before that great software is built in layers, and components are ideal layers for live view because they help us compartmentalize the markup of our survey sections and the state of each of those sections. LiveView components are the perfect fit to meet the requirements of reusability and complex state management.

Let's take a closer look at what a component is under the hood and how it fits into a live view.

Components Breakdown LiveView Markup and State

When you're building applications with pure HTML, it's relatively easy to share code. HTML is a string, so composing with them is straightforward. Live views are different. A single live view combines the ideas of state management, HTML rendering, and event handling. We need a more sophisticated strategy to compose code with them beyond the typical ideas of helpers and templates, which can't do much more than wrap up sections of HTML. That leaves a void.

Components step neatly into that void. You've already seen that a component is a way to build live views in layers. Each layer maintains its own markup, state, and events. In a component, we establish state, express what happens upon a render, and how to process events. LiveView takes care of the rest. Components therefore allow us to break down *all* of the functionality of LiveView into smaller sections that are composable and reusable.

Components Share the Parent LiveView Process

Components run in the same LiveView process as the parent LiveView in which they are rendered. That means the parent live view manages the overall state of the survey and each LiveView component manages its markup and handles the state for the individual part of the view it represents.

OTP, LiveView, and Components

Components run in the same OTP server as their parent. There's one shared state, and one supervisor. That means error and failure handling all happen at the level of

the parent live view. If you don't know what these details mean, don't worry about them for now. Make a note to yourself to study these concepts later if they interest you.

For our survey feature, a parent live view will manage the state changes related to the overall survey. Individual components will handle the markup details and manage the state of the individual survey sections—the demographics section and the product ratings sections.

Now that you have a little more background on what components are and how they function, we can get to work. We're going to generate a context to build the base model, one that will let us manage the surveys.

Then, we'll build a frontend that leverages components to let our users do what we want. Let's get rolling.

Build The Survey Context

Before we can create the live view and components that represent the survey feature, we need to build out the backend services that will support them. We'll design a Survey context, with schemas for Demographic and Rating. Then, we'll be able to use the Survey context in our live view.

We'll take a slightly different approach to building the context and schemas than the one you saw in the previous chapters. We'll still rely on code generation, this time reaching for the `phx.gen.context` generator to build *just* a context and schemas, rather than the Phoenix Live generator that *also* creates live views and routes. This is because we'll be creating our own custom live view and components to handle the survey functionality later on. We're building a LiveView frontend with specific behaviors and features that the Phoenix Live generator just won't accommodate.

We'll begin by running the generator, but we'll need to do a little bit of customization on top of the generated code in order to get our data into the correct shape. When we're done with this section, you'll know how to strategically deploy the Phoenix Context generator to build the foundation of a custom feature set, you'll be comfortable adding your own code on top of the generated code, and you'll be prepared to use your new context in LiveView to build out the dynamic, interactive survey.

Generate and Customize the Context

Type this command to generate the context:

```
[pento] → mix phx.gen.context Survey Demographic demographics gender:string \
          year_of_birth:integer user_id:references:users:unique

* creating lib/pento/survey/demographic.ex
* creating priv/repo/migrations/20200919160125_create_demographics.exs
...
```

This command tells the Phoenix Context generator to make a demographics database table with the provided columns, a Demographic schema module for interacting with that database table, and a Survey context to present an API through which to interact with the Demographic core. Look at the `:user_id` column. It has a unique constraint that will allow only one demographic record per user. This kind of constraint enforces uniqueness at the database level, and that will prevent our database from persisting bad data.

Next, generate the Rating schema, like this:

```
[pento] → mix phx.gen.context Survey Rating ratings stars:integer \
          user_id:references:users product_id:references:products
```

You are generating into an existing context.

```
...
Y
...
* creating lib/pento/survey/rating.ex
* creating priv/repo/migrations/20200917002620_create_ratings
```

Phoenix warns us that we're putting our Rating schema in the same Survey context as the Demographic schema. Since we believe these concepts are closely related, that's exactly what we want to do. So we specify Y to continue.

We'll want to ensure that a user rates a given product just once, so open up the generated ratings migration and add a unique index on the user and product_id fields, like this:

```
stateless_components/pento/priv/repo/migrations/20200917002620_create_ratings.exs
create index(:ratings, [:user_id])
create index(:ratings, [:product_id])
# Add the following unique index
create unique_index(
  :ratings,
  [:user_id, :product_id],
  name: :index_ratings_on_user_product
)
```

It's just an Ecto index, but instead of specifying index we specify unique_index. That way, our database will only allow one rating per [:user_id, :product_id] combination.

We also need to add the corresponding unique constraint to the Rating schema's changeset, like this:

```
stateless_components/pento/lib/pento/survey/rating.ex
```

```
|> unique_constraint(:product_id, name: :index_ratings_on_user_product)
```

While we're here in the Rating schema, let's make a few other changes. First, we'll update the schema to reflect that ratings belong to both users and products. That way, we'll have access to user and product fields, as well as the existing user_id and product_id fields on our Rating struct. Add a call to the belongs_to macro for both User and Product, like this:

```
stateless_components/pento/lib/pento/survey/rating.ex
```

```
alias Pento.Catalog.Product
```

```
alias Pento.Accounts.User
```

```
schema "ratings" do
```

```
  field :stars, :integer
```

```
  belongs_to :user, User
```

```
  belongs_to :product, Product
```

```
  timestamps()
```

```
end
```

Next up, let's update the changeset to cast and require the :user_id and :product_id attributes. Finally, validate :stars as an integer between 1 and 5, like this:

```
stateless_components/pento/lib/pento/survey/rating.ex
```

```
def changeset(rating, attrs) do
```

```
  rating
```

```
  |> cast(attrs, [:stars, :user_id, :product_id])
```

```
  |> validate_required([:stars, :user_id, :product_id])
```

```
  |> validate_inclusion(:stars, 1..5)
```

```
  |> unique_constraint(:product_id, name: :index_ratings_on_user_product)
```

```
end
```

Excellent. We take advantage of the built-in validate_inclusion/3 Ecto changeset validation, passing the field and the range of possible values.

We've told the Rating schema that ratings belong to a product. Now, we need to add the inverse of this relationship to the Product schemas. Open up the Product schema and add these changes to specify that a product has many ratings:

```
stateless_components/pento/lib/pento/catalog/product.ex
```

```
alias Pento.Survey.Rating # Add this line
```

```

schema "products" do
  field :description, :string
  field :name, :string
  field :sku, :integer
  field :unit_price, :float
  timestamps()
  has_many :ratings, Rating # Add this line
end

@doc false
def changeset(product, attrs) do
  product
  |> cast(attrs, [:name, :description, :unit_price, :sku])
  |> validate_required([:name, :description, :unit_price, :sku])
  |> unique_constraint(:sku)
end
end

```

This will give us the ability to ask a given product for its ratings by calling `product.ratings`. We'll take advantage of this capability later on. Let's move on for now to the Demographic schema.

First, update the Demographic schema to use the `belongs_to` macro for the User association:

```

stateless_components/pento/lib/pento/survey/demographic.ex
alias Pento.Accounts.User

```

```

schema "demographics" do
  field :gender, :string
  field :year_of_birth, :integer
  belongs_to :user, User

  timestamps()
end

```

Perfect. It works the same way that it did in the Rating schema. Now, update the Demographic schema's `changeset/2` function to cast and require the `user_id` field, add a constraint for the unique `user_id` index, and add some custom validations for demographic gender and year of birth.

```

stateless_components/pento/lib/pento/survey/demographic.ex
def changeset(demographic, attrs) do
  demographic
  |> cast(attrs, [:gender, :year_of_birth, :user_id])
  |> validate_required([:gender, :year_of_birth, :user_id])
  |> validate_inclusion(
    :gender, ["male", "female", "other", "prefer not to say"]
  )
  |> validate_inclusion(:year_of_birth, 1900..Date.utc_today.year)
  |> unique_constraint(:user_id)
end

```

end

Done and done.

Now, run the migration:

```
[pento] → mix ecto.migrate
```

Excellent. We have an up-to-date database, and a working Survey context. Now, we can take it for a test drive.

Explore the Generated Context and Schema

Let's fire up IEx and play around with creating some demographics and ratings using the generated Survey context, which provides the API for the CRUD interactions of these schemas. This will familiarize us with the usage of our generated and customized context so that we'll be prepared to leverage it in our live views.

We'll create a user with the help of the Accounts context:

```
iex> alias Pento.Accounts
Pento.Accounts
iex> user_attrs = %{email: "cassandra@griox.io", password: "Tr0yW1llF8ll"}
%{email: "cassandra@griox.io", password: "Tr0yW1llF8ll"}
iex> {:ok, user} = Accounts.register_user(user_attrs)
...
{:ok,
 #Pento.Accounts.User<email: "cassandra@griox.io",id: 1,...>}
```

We added a user, and now we can create a demographic for them:

```
iex> alias Pento.Survey
Pento.Survey
iex> demo_attrs = %{
  user_id: user.id,
  gender: "prefer not to say",
  year_of_birth: 1989
}
%{gender: "prefer not to say", user_id: 1, year_of_birth: 1989}
iex> Survey.create_demographic(demo_attrs)
...
{:ok,
 %Pento.Survey.Demographic{gender: "prefer not to say",id: 1,user_id: 1,...}
}
```

Nice. Now, assuming you have a product in your database from the seeding exercise we did in [Chapter 3, Generators: Contexts and Schemas, on page 57](#), you can create a rating for the new user and the product with an ID of 1. Go back to your IEx session and add in this:

```
iex> rating_attrs = %{user_id: user.id, product_id: 1, stars: 5}
%{user_id: user.id, product_id: 1, stars: 5}
iex> Survey.create_rating(rating_attrs)
{:ok,%Pento.Survey.Rating{id: 1,product_id: 1,stars: 5,user_id: 1}}
```

Easy enough. Now, let's exercise the rating constraints, like this:

```
iex> Survey.create_rating(%{user_id: user.id, product_id: 1, stars: 1})
[debug] QUERY ERROR db=4.5ms queue=0.5ms idle=1952.2ms...
{:error, #Ecto.Changeset<...
  errors: [
    product_id: {"has already been taken",
      [constraint: :unique, constraint_name: "index_ratings_on_user_product"]}
  ],
  ...
  valid?: false
>}
```

It's not valid, and the message tells us exactly why.

We've seen the basic functionality of the context in action. Let's shift our attention to working with the core.

Organize The Application Core and Boundary

In previous chapters, we didn't need to execute queries that were more complex than the CRUD-supporting ones provided by generated code. Our survey feature is a bit different, however. In order to support the survey functionality, we'll need to execute some custom queries. In this section, you'll learn how to compose and execute complex database queries with Ecto, and you'll see how this work fits into the organized core and boundary layers of an application. Then, you'll be ready to use your custom queries in the survey live view.

Ecto query composition, as you already know, is certain and predictable. It belongs in your application's core. But where exactly in the core should you put code that dynamically constructs complex queries?

Queries are a little bit like functions. It's fine to express short ones in-line, much like anonymous functions, within the scope of a module like a context. Sometimes, however, it is important to provide a first class function to express and name more complex queries. These functions belong in their very own dedicated query builder modules in the application core. Before we build any such modules however, let's discuss the queries that our survey feature will need to use.

We will need the following individual queries to support the survey feature:

- The demographic section of our survey will need a query to return the demographic for a given user.
- The ratings section of the survey will rely on a query to return all products, with preloaded ratings for a given user.

Let's begin with the first query.

Query for User Demographics

We need to define a module that will implement the function for querying a user's demographic record. This module will live in the application core and, since it is responsible for demographic queries, we'll name it `Survey.Demographic.Query`:

```
stateless_components/pento/lib/pento/survey/demographic/query.ex
```

```
defmodule Pento.Survey.Demographic.Query do
  import Ecto.Query
  alias Pento.Survey.Demographic

  def base, do: Demographic

  def for_user(query \\ base(), user) do
    query
    |> where([d], d.user_id == ^user.id)
  end
end
```

With the `base/0` function, we name the concept of a base query and we provide one common way to build the foundation for all Demographic queries. This type of function is called a *constructor*. We'll rely on it to create an initial query for demographics.

Next, we have another kind of function called a *reducer*. These are not specifically functions that we can use in `Enum.reduce/2`. Instead, they are functions that take some type along with additional arguments, and apply those additional arguments to return the same type. In our case, our classic reducer takes a `user_id` and transforms the initial query with an additional `where` clause. By building code in this way, we create elements that pipe together cleanly. This reducer pattern should look familiar to you from our examination of Phoenix request handling in [Chapter 2, Phoenix and Authentication, on page 31](#). It's no different from the manner in which a pipeline of plugs operates on a connection.

Now, we can make the query available in the context, like this:

```
stateless_components/pento/lib/pento/survey.ex
```

```
def get_demographic_by_user(user) do
  user
```

```
|> Demographic.Query.for_user()
|> Repo.one()
end
```

We always wrap calls to the query builder in the relevant context. The Survey context pipes the constructed query into a call to `Repo.one/1`. Now, we can test drive it in IEx:

```
iex> Survey.get_demographic_by_user(user)
...
%Pento.Survey.Demographic{gender: "prefer not to say", id: 1,user_id: 1...}
```

Now let's apply the same approach to our product ratings query.

Query for Product Ratings

We'll begin by implementing another dedicated querying module in the application's core—`Catalog.Product.Query`.

```
stateless_components/pento/lib/pento/catalog/product/query.ex
defmodule Pento.Catalog.Product.Query do
  import Ecto.Query
  alias Pento.Catalog.Product
  alias Pento.Survey.Rating

  def base, do: Product
```

In the `base/0` function, we establish the base query for returning all products. Once again, it makes sense to put this base query in a reusable function. Beyond naming the concept explicitly, which is a good practice in its own right, this approach saves us a lot of potential future work—if we ever need to change the base query for our whole application, we can do so in one place.

Next up, we'll create a reducer function that takes in a query and returns an annotated query to preload user ratings for the desired products.

```
stateless_components/pento/lib/pento/catalog/product/query.ex
def with_user_ratings(query \ base(), user) do
  ratings_query = Rating.Query.preload_user(user)

  query
  |> preload(ratings: ^ratings_query)
end
end
```

In the `with_user_ratings/2` reducer, we execute the `Ecto preload/2` function to fetch user ratings. Remember, `Ecto` is explicit. If you want it to load relationships, you need to ask for them. We execute the `preload/2` function with a query for ratings belonging to the given user. That logic is in turn wrapped up in

another query builder module responsible for rating query logic, `Survey.Rating.Query`:

```
stateless_components/pento/lib/pento/survey/rating/query.ex
```

```
defmodule Pento.Survey.Rating.Query do
  import Ecto.Query
  alias Pento.Survey.Rating

  def base, do: Rating

  def preload_user(user) do
    base()
    |> for_user(user)
  end

  defp for_user(query, user) do
    query
    |> where([r], r.user_id == ^user.id)
  end
end
```

Next, we'll consume our reducer function in the Catalog context. Remember that the context module functions as the boundary layer of the Phoenix application. It handles the uncertainty of executing database interactions. So, we'll call on our new query function on the context, piping it into a call to `Repo.all/2` to execute the query like this:

```
stateless_components/pento/lib/pento/catalog.ex
```

```
def list_products_with_user_ratings(user) do
  Product.Query.with_user_ratings(user)
  |> Repo.all()
end
```

Our function accepts a user, calls `with_user_ratings/2`, and pipes it straight to `Repo.all/1`. The function will return a list of all products, each with any user ratings. Let's see it in action:

```
iex> alias Pento.Survey
iex> alias Pento.Accounts
iex> alias Pento.Catalog
iex> user = Accounts.get_user!(1)
iex> Survey.create_rating(%{user_id: user.id, product_id: 1, stars: 5})
...
iex> Catalog.list_products_with_user_ratings(user)
[
  %Pento.Catalog.Product{
    description: "The classic strategy game", name: "Chess", ...
    ratings: [%Pento.Survey.Rating{id: 1, product_id: 1, stars: 1, user_id: 1}]
  },
  %Pento.Catalog.Product{
    description: "The game of Xs and Os", name: "Tic-Tac-Toe", ratings: []
  }
]
```

```
  }, ...
]
```

And it works! We alias what we need, create a rating, get a user, and then fetch our products. Notice that the products include the preloaded ratings belonging to the given user.

Now that we have a handle on the core functionality of our survey, let's build some LiveView.

Build The Survey Live View

It's time to focus on the survey live view. We know in broad strokes what it will look like. Users will be asked to fill out their demographic information, followed by a rating for each of our products. We're going to approach the survey feature from the outside in. We'll build a route first, then we'll mount and render the initial live view.

Establishing the initial state of the survey live view in the mount/render workflow will give you yet another opportunity to see the reducer pattern in action. You've seen plug pipelines iteratively transform a connection struct, and you've written query builders that do the same for Ecto queries. In this section, you'll see that live view applies this same exact pattern to create and update the state of a live view for our users by reducing over the common data structure of the socket struct. You'll build your own live view reducer pipeline and use it in the mount/3 function. Along the way, you'll get a look at one of the tools that LiveView provides to improve performance during the mount/render workflow, the `assign_new/3` function.

Let's get started with the route.

Define The Survey Route

Our first job is to establish a route. The survey will live at `/survey`, and it should work only for authenticated users so we can deliver a survey to single, identifiable users. We'll tie the route to the yet-to-be-written `SurveyLive` live view, with the `:index` live action, like this:

```
stateless_components/pento/lib/pento_web/router.ex
scope "/", PentoWeb do
  pipe_through [:browser, :require_authenticated_user]
  live "/survey", SurveyLive, :index
end
```

Notice we use the pipeline `[:browser, :require_authenticated_user]`, which means our new connections will flow through the full browser pipeline and then the `require_authenticated_user` plug before matching our route.

Recall that the `require_authenticated_user/2` is one of the function plugs we generated earlier on in [Chapter 2, Phoenix and Authentication, on page 31](#). As a result, anyone who tries to visit `/survey` without first logging in will be redirected to the log-in page. Once again, we're seeing our generated authentication layer used to protect a live view route.

With our route established, it's time to define the `SurveyLive` live view.

Mount the Survey Live View

The `mount/3` function builds the initial state for `SurveyLive`. Let's think a bit about that initial state. We know we'll need to use the current user to build the demographic and rating portions of our survey, so we want to store that user in the live view's state. This way, we can make it available to the demographic and ratings components later on. We'll implement `mount/3` to find the current user and add them to state, and we'll do it in a way that leverages a pattern you're getting pretty familiar with—the reducer pattern.

```
# lib/pento_web/live/survey_live.ex
defmodule PentoWeb.SurveyLive do
  use PentoWeb, :live_view
  alias Pento.{Catalog, Accounts, Survey}

  @impl true
  def mount(_params, %{ "user_token" => token } = _session, socket) do
    { :ok,
      socket
      |> assign_user(token) }
  end
end
```

Let's break this down. Notice the function head. It's ignoring the inbound params, but we do grab a `user_token` out of the session argument. The `phx.gen.auth` generator built the code that adds this secure token to the session, and the session is made available to any live view as the second argument to the `mount/3` function. We pick the user token off here so we can use it to fetch a user. Fetching the user will happen in the `assign_user/1` reducer.

Remember, a reducer is a function that takes an accumulator of some type and returns a transformed accumulator of the same type. When you think about it, even the plain old `assign/2` function is a reducer.

As we build the `mount/3` function, we will break down the responsibility of reducing the socket into smaller layers. Each layer will be comprised of an

individual reducer function responsible for adding some piece of state to the socket. The `mount/3` function will orchestrate these reducer layers into a pipeline. That's exactly what we begin to do with `assign_user/2`.

We want `assign_user/2` to take a first argument of our live view's type, the socket, and return an updated socket that has the `:current_user` key. It will do so by using the token from the session to fetch the user from the database using the Accounts context.

But wait! You might be thinking that the `Plug.Conn` *already* stores the current user, once again courtesy of our generated authentication code's `fetch_current_user` plug:

```
stateless_components/pento/lib/pento_web/controllers/user_auth.ex
@doc """
Authenticates the user by looking into the session
and remember me token.
"""
def fetch_current_user(conn, _opts) do
  {user_token, conn} = ensure_user_token(conn)
  user = user_token && Accounts.get_user_by_session_token(user_token)
  assign(conn, :current_user, user)
end
```

Why should we have to execute *another* database query for something that is *already* stored in the `Plug.Conn` connection object? On top of that, you'll remember that the `mount/3` function is actually called *twice* for any given live view: once to do the initial page load and again to establish the live socket. This means we're in danger of executing the same database query *twice*, once each time the live view's `mount/3` is invoked, to fetch a current user that we *already* fetched and stored in the `Plug.Conn` before the request even reached the live view.

If only there was some way to access the current user from the `Plug.Conn` when the live view first mounts...

As it turns out, we can use the `assign_new/3` function to do exactly that. When a live view first mounts in the disconnected state, the `Plug.Conn` assigns is available inside the live view's socket under `socket.private.assign_new`. This allows the connection assigns to be shared for the initial HTTP request. The `Plug.Conn` assigns will *not* be available during the connected mount.

This small feature is actually a pretty important one. It means that on the initial mount, we can set the live view's socket assigns to contain the current user stored in the `Plug.Conn` assigns. Then, on the *second*, connected mount, when we no longer have access to the `Plug.Conn` assigns, we'll fetch the current

user from the database using the token from the session. In this way, we avoid making unnecessary database calls. We only have to execute our “get user” query once, on the second, connected, mount.

We’ll wrap up the `assign_new/3` in a reducer function, `assign_user/1`, shown here:

```
stateless_components/pento/lib/pento_web/live/survey_live.ex
def assign_user(socket, token) do
  assign_new(socket, :current_user, fn ->
    Accounts.get_user_by_session_token(token)
  end)
end
```

`assign_new/3` takes in three arguments: the socket, the key to add to socket assigns, and a function. Let’s find out exactly what happens under the hood.

Keep in mind that `Plug.Conn` also has an `assigns` field where data describing the connection is stored. When the router invokes `mount/3`, the live view’s socket will have the `Plug.Conn` assigns in a private holding area called `socket.private.assign_new`. So, the `assign_new/3` function can look in `socket.private.assign_new` for the `:current_user` key we request. If it finds that key, it will use its value to populate that same key in the live view’s socket assigns. If it does not find that key in `socket.private.assign_new`, it will use the function we provide to populate a key by that name in the live view’s socket assigns.

Let’s see it in action. First, save an empty template in `lib/pento_web/live/survey_live.html.leex`. Next, add these debugging statements to the `assign_user/2` reducer:

```
defp assign_user(socket, token) do
  IO.puts "Assign User with socket.private:"
  IO.inspect socket.private
  assign_new(socket, :current_user, fn ->
    Accounts.get_user_by_session_token(token)
  end)
end
```

Now, point your browser at `localhost:4000/survey` and the following process will occur:

- The `assign_user/2` function is invoked twice, once on the first disconnected mount and again when the connected mount is invoked
- The first time around, `socket.private` contains an `assign_new` map that holds the `:current_user` from `Plug.Conn` assigns
- The second time around, `socket.private` contains an empty `assign_new` map

You should see the following output in your server logs, illustrating this process exactly:

```

...
Assign User with socket.private:
%{
  assign_new: {%{
    current_user: #Pento.Accounts.User<
      ...
      email: "sophie6@email.com",
      id: 2,
      ...
    >
  }, []},
  ...
}
...
Assign User with socket.private:
%{
  assign_new: {%{}, [:current_user]},
  ...
}
...

```

That's exactly what we expected to see, and it underscores why we need `assign_new`. On the first page render of our live view, we use the current user from the `Plug.Conn`. On the second, WebSocket-backed render of our live view, we no longer have access to that user from the `Plug.Conn` so we use the session token to fetch the current user from the database. This way, we avoid making unnecessary database queries.

Now that our data is set up, it's time to render.

Render the Template

After `mount/3` finishes, the live view will render. We don't provide a `render/1` function, instead we're using a template—`lib/pento_web/live/survey_live.html.leex`. Let's keep it simple for now:

```

<% # lib/pento_web/live/survey_live.html.leex %>
<section class="row">
  <h2>Survey</h2>
</section>

```

Reload your browser and you'll see the bare bones template shown here:



Phoenix Framework

[Get Started](#)
[LiveDashboard](#)[cassandra@grox.io](#)[Settings](#)[Log out](#)

Survey

When all our ratings are complete, we'll show the completed survey, like this:

Survey

Demographics ✓

Gender: female

Year of birth: 2003

Ratings ✓

Chess: ★ ★ ★ ★ ★

Tic-Tac-Toe: ★ ★ ★ ★ ★

Table Tennis: ★ ★ ★ ★ ★

Let's begin building the Demographic form component so we can use it on this page.

Build The Demographic Form

The first part of the survey is the demographic section, and we'll wrap up that section in a component. First, we'll review how to implement and render a stateless component. You'll build a stateless component and render it within the SurveyLive live view. Then, we'll dive into the component's lifecycle. You'll set the necessary component state and use it to render the form for creating a demographic.

Let's do a quick refresher. Remember, you'll render a component with `live_component/3` with a socket, the component module, and some assigns. A component is stateless if it doesn't have an `:id` key in assigns. That means the component can be pretty simple, often with nothing more than an `update/2` function and a template or a render.

Let's call a stateless component in our SurveyLive template. Open up `live/survey_live.html.leex`, and add this:

```
# pento/lib/pento_web/live/survey_live.html.leex
<section class="row">
  <%= live_component @socket,
    PentoWeb.DemographicLive.FormComponent,
    content: "Hello to the Demographic Form Component" %>
</section>
```

This won't work because we haven't built the `DemographicLive.FormComponent` yet. Still, we know where to put the new code. We're starting dead simple, with a stateless component. We can tweak it to do what we want as we go.

Our next step is to define the `DemographicLive.FormComponent`, like this:

```
# pento/lib/pento_web/live/demographic_live/form_component.ex
defmodule PentoWeb.DemographicLive.FormComponent do
  use PentoWeb, :live_component
end
```

...and a corresponding template, in `pento_web/live/demographic_live/form_component.html.leex`, like this:

```
<div class="hero"><%= @content %></div>
```

Remember, the `use PentoWeb, :live_component` announces our intention to consume the `LiveComponent` behaviour, and also imports some useful functions like the `~L` sigil to construct LEX markup. The rest of the component is surprisingly sparse.

Believe it or not, since this is a stateless component, the module has everything it needs to be a valid LiveView component.

We don't implement our own `mount/1`, so our component picks up the default `mount/1` from the `LiveComponent` behaviour. The default `mount/1` function returns an unmodified `{:ok, socket}` tuple.

It also picks up a default `update/2` function that passes any options we specify directly into the component's `socket assigns`, including our `:content` key.

In the template, we drop our content into a `div` with a little CSS to make it pretty.

Now, if you reload your browser, you should see the sentence we added here:

Survey

Hello to the Demographic Form Component

Now we're ready to render the demographic form. But in order to do so, we need to make our component a little smarter.

Stateless Components as Tiny Helpers

If you need a component to handle markup issues like lists or the like, without the need to process events or states, a stateless component is a good way to go. CSS frameworks also have specific markup requirements for onscreen elements like menus that need only input parameters. These kinds of problems are perfect for stateless components. They participate well in `LiveView`'s change tracking because they will update and re-render as-needed, whenever the parent live view changes.

Implement the Demographic Lifecycle

We'll use Phoenix's `form_for` function to construct the demographic form. This function requires a `changeset`, so we'll need a slightly more sophisticated component. `live_component/3` starts the component in the parent view's process, and calls these callbacks, in order:

mount/1

The single argument is the `socket`, and we use this callback to set initial state. This callback is invoked only once, when the component is first rendered from the parent live view.

update/2

The two arguments are the assigns argument given to `live_component/3` and the socket. By default, it merges the assigns argument into the `socket.assigns` established in `mount/1`. We'll use this callback to add additional content to the socket *each time* `live_component/3` is called.

render/1

The one argument is `socket.assigns`. It works like a render in any other live view.

Stateless components are always mounted, updated, and rendered whenever the parent LiveView changes state. That's why they are stateless. They don't maintain their own state after a render.

We'll use the `update/2` callback to add a Demographic changeset to `socket.assigns` so we can render it in a form on the template.

Our demographic *belongs to* a user and we'll need access to that user to construct a demographic changeset. So, our `SurveyLive` live view will need to pass in the current user from its own socket assigns, like this:

```
<!-- lib/pento_web/live/survey_live.html.leex -->
<%= live_component @socket,
  PentoWeb.DemographicLive.FormComponent,
  user: @current_user %>
```

Now, we can implement an `update/2` function to build a Demographic struct and changeset, like this:

```
stateless_components/pento/lib/pento_web/live/demographic_live/form_component.ex
```

```
alias Pento.Survey
alias Pento.Survey.Demographic

def update(assigns, socket) do
  {:ok,
   socket
   |> assign(assigns)
   |> assign_demographic()
   |> assign_changeset() }
end
```

This code uses the same technique we used in our `SurveyLive.mount/3` function. We build a couple of reducers to add the demographic and changeset to our `socket.assigns` and string them into a nice pipeline. By this point, the reducer functions should look familiar. Here's the first one, `assign_demographic/1`:

```
stateless_components/pento/lib/pento_web/live/demographic_live/form_component.ex
```

```
def assign_demographic(%{assigns: %{user: user}} = socket) do
  assign(socket, :demographic, %Demographic{user_id: user.id})
end
```

end

It simply adds an almost empty demographic struct containing the the user_id for the current user.

And here's the one that adds the changeset:

```
stateless_components/pento/lib/pento_web/live/demographic_live/form_component.ex
def assign_changeset(%{assigns: %{demographic: demographic}} = socket) do
  assign(socket, :changeset, Survey.change_demographic(demographic))
end
```

We use the Survey context to build a changeset, and we're off to the races. Once the update/2 function finishes, the component renders the template. Let's update that template now to give it a real demographic form for our shiny new changeset.

Render The Demographic Form

You've seen what a Phoenix form looks like. We won't bore you with the details. For now, add this to lib/pento_web/live/demographic_live/form_component.html.leex:

```
<%= f = form_for @changeset, "#",
  phx_submit: "save",
  id: "demographic-form"%>

  <%= label f, :gender %>
  <%= select f, :gender, ["female", "male", "other", "prefer not to say"] %>
  <%= error_tag f, :gender %>

  <%= label f, :year_of_birth %>
  <%= select f, :year_of_birth, Enum.reverse(1940..2020)%>
  <%= error_tag f, :year_of_birth %>

  <%= hidden_input f, :user_id %>

  <%= submit "Save", phx_disable_with: "Saving..." %>
</form>
```

Our update/2 function added the changeset to our socket assigns, and we access it with @changeset in our form_for function. form_for takes in the changeset, has an id, and applies the phx-save LiveView binding for saving the form. Our form has labels, fields, and error tags for each field we want the user to populate, and an additional user_id hidden field to ensure the user ID is included in the form params. Finally, there's a submit tag with a phx-disable_with function—a little nicety that LiveView provides to handle multiple submits.

Now if we visit /survey we should see our survey page, including the demographic form, as shown here.

Survey

Demographics

Gender

female ▼

Year of birth

2020 ▼

SAVE

But, if you try to submit the form, you should see this error page:

Phoenix.Router.NoRouteError at POST /survey

no route found for POST /survey (PentoWeb.Router)



Available routes

page_path	GET	/	Phoenix.LiveView.Plug :inc
product_index_path	GET	/product	Phoenix.LiveView.Plug :inc
product_index_path	GET	/product/new	Phoenix.LiveView.Plug :new
product_index_path	GET	/product/:id/edit	Phoenix.LiveView.Plug :edi
product_show_path	GET	/product/:id	Phoenix.LiveView.Plug :shc
product_show_path	GET	/product/:id/show/edit	Phoenix.LiveView.Plug :edi
rating_index_path	GET	/rating	Phoenix.LiveView.Plug :inc
rating_index_path	GET	/rating/new	Phoenix.LiveView.Plug :new
rating_index_path	GET	/rating/:id/edit	Phoenix.LiveView.Plug :edi
rating_show_path	GET	/rating/:id	Phoenix.LiveView.Plug :shc
rating_show_path	GET	/rating/:id/show/edit	Phoenix.LiveView.Plug :edi
live_dashboard_path	GET	/dashboard	Phoenix.LiveView.Plug :hon
live dashboard path	GET	/dashboard/:node/home	Phoenix.LiveView.Plug :hon

It predictably crashed, but not for the reason we thought! We haven't implemented our handler yet, but the problem is something different. Look at the logs:

```
[error] GenServer #PID<0.1478.0> terminating
...
```

```
** (UndefinedFunctionError) function PentoWeb.SurveyLive.handle_event/3 is
undefined or private
```

Did you catch the problem? We did get an undefined `handle_event/3`, but we got it for the `SurveyLive` view, *not* our component! While we *could* send the event to `SurveyLive`, that's not really in the spirit of using components. Components are responsible for wrapping up markup, state, and events. Let's keep our code clean, and respect the *single responsibility principle*.

The `DemographicLive.FormComponent` should handle both the state for the survey's demographic section and the events to manage that state. As a stateless component, however, it can't process any events and it doesn't maintain its own state. We need to fix that. In fact, we need a stateful component, but we'll save that for the next chapter. For now, it's time to wrap up.

Your Turn

The art of building software is the art of breaking down one complex problem into several simple ones, and that means layering. `LiveView` provides two kinds of components for this purpose. Stateless components encapsulate common *rendering* code and allow you to compose such code into layers.

In this chapter, you built a stateless component, and the same technique you used here can work for a wide variety of tasks. Start by generating common code in a context, and tailoring that code with the backend features your component will need. Then, use that context to establish the data your component will need via a `mount/1` or `update/2` function.

After you've nailed down the backend and data for your component, it will be time to shift to the presentation, and the `render/1` component callback works just as it does within regular live views. Once you implement a `render/1` function or a template, you use your component with a call to `live_component/3`. You don't have to stop there. Each component can embed components of its own, and a live view can render multiple components.

In this way, you can compose a set of components, each of which is responsible for rendering some distinct portion of the page, into one single live view. By building your live view in layers, you keep your code clean, organized and easy to maintain.

After building this component together, it's your turn to put what you've learned into practice.

Give It a Try

These problems let you build your own components.

- Stateless components provide a great way of sharing common user interface blocks. Build a component to render an HTML title, with a heading and a configurable message. Render this component multiple times with different messages on the same page in your SurveyLive live view. What are the strengths and limitations of this approach?
- Build a component that renders an HTML list item. Then, build a component that uses this list item component to render a whole HTML list in the SurveyLive live view. Can you configure your components to render any given list of items? Although this composable list exercise is somewhat simplified, can you think of some scenarios in which this component-based approach will help you build live views in an organized, re-usable way?

Next Time

Stateful components allow shared rendering just as stateless ones do, and also support *events that manage state*. In the next chapter, we'll transform our stateless component into a stateful one so that our demographic form can be submitted. Then, we'll move on to the product ratings functionality of our survey. When we're done with the survey feature, you'll have learned how a set of components can be composed to manage the state and behavior of a single-page flow.

Stateful Components

In the previous chapter, we began building an interactive survey with components. First, we reached for a stateless component to render the demographic form, only to find that it's not sufficient for our purposes. In this chapter, we'll convert that stateless component into a stateful one so that it can have event handlers that change state. Then, we'll build out the ratings survey components and compose them into our fully interactive survey.

Along the way, you'll learn how components can communicate with their parent live view, you'll see how components allow you to build clean and organized code that adheres to the single responsibility principle, and you'll implement component composition logic that allows you to manage even complex state for your single page applications.

Make the Demographic Component Stateful

We already know that we need to convert our stateless demographic form component into a stateful one. Luckily, it's easy to make our component stateful. All we need to do is add an `:id`. Then our component will be able to respond to events and manage its own state.

You probably expected a long, involved discussion about stateful components here, but that's really all there is to it. Let's add that `:id` now, like this:

```
stateful_components/pento/lib/pento_web/live/survey_live.html.leex
<%= live_component @socket,
  PentoWeb.DemographicLive.FormComponent,
  user: @current_user,
  id: "demographic-form-#{@current_user.id}" %>
```

Nice. We simply add the `:id` to the `assigns` argument given to `live_component/3` and we're off to the races. LiveView identifies stateful components by their component module and the provided `:id`. Ecto IDs and other application IDs

are fair game as long as that ID is unique to the call to `live_component/3` on the given page. It's also worth noting that the given `:id` is *not* used as the DOM ID. If you want to set a DOM ID, it is your responsibility to set it in your template.

Now, we can send events to our demo form, so it's time to add some handlers. First, we'll briefly discuss the stateful component lifecycle that we'll take advantage of in order to manage component state.

Manage Component State

The lifecycle of a stateful component is a little different from the stateless component lifecycle. Whenever `live_component` is called, a stateless component will invoke `mount/1`, `update/2` and `render/1`. A stateful component, on the other hand, has an additional callback. On first render, it will call `preload/1` *before* calling `mount/1`, `update/2` and then `render/1`. Subsequent renders of stateful components will *not* trigger `mount/1`. Instead, `preload/1` is called, followed by `update/2` and then `render/1`.

We won't take advantage of `preload/1` in our component, but it's worth discussing what it can do for us. The `preload/1` function lets LiveView load all components of the same type at once. In order to understand how this works, we'll look at an example.

Let's say you were rendering a list of product detail components. You might accomplish this by iterating over a list of product IDs in the parent live view and calling `live_component/3` to render each product detail component with a given product ID. Each component in our scenario is responsible for taking the product ID, using it to query for a product from the database, and rendering some markup that displays the product info. Now, imagine that `preload/1` does not exist. This means you are rendering a product detail component once for each product ID in the list. 20 product IDs would mean 20 components and 20 queries—each product detail component would need to issue its own query for the product with the given ID.

With `preload/1`, you can specify a way to load *all* components of the same type *at once*, while issuing a single query for all of the products in the list of product IDs. You should reach for this approach whenever you find yourself in such a situation.

We don't have that problem, so let's move on. We're ready to teach our stateful component how to handle events.

Target an Event

So far, most of the events we've seen get sent to the parent live view. We even accidentally sent the demographic form save event to the parent live view in the previous chapter. Let's fix that now. In order to send an event to some component, we need to specify a `phx-target` HTML attribute with the id of the component we want to receive the event.

Most often, we want events to be received by the component whose template is sending them. To make that easy to do, `LiveComponent` automatically sets a key in the component's assigns called `:myself` with a value of the id we pass in to the component. Let's add the `phx-target` to our form component now:

```
<%= f = form_for @changeset, "#",
  phx_target: @myself,
  phx_submit: "save",
  id: "demographic-form"%>

  <%= label f, :gender %>
  <%= select f, :gender, ["female", "male", "other", "prefer not to say"] %>
  <%= error_tag f, :gender %>

  <%= label f, :year_of_birth %>
  <%= select f, :year_of_birth, Enum.reverse(1940..2020)%>
  <%= error_tag f, :year_of_birth %>

  <%= hidden_input f, :user_id %>

  <%= submit "Save", phx_disable_with: "Saving..." %>
</form>
```

Here, we've added the new `phx-target` attribute, giving it a value of the `@myself` assignment.

Now that we're sending events to the stateful demographic form component, let's teach it how to handle them.

Handle The Save Event

We need to implement a `handle_event/3` function for the save event. Here's how it will work.

First, we'll build our `handle_event/3` function head that matches the save event. The event will receive a socket and the parameters of the form.

Next, we'll make a reducer to save the form, and return the saved socket.

Finally, we'll call our reducer in `handle_event/3`. In this way, our handler will stay skinny, and we'll have another single-purpose function to add to our module.

Let's start with the handler. We'll define a function head that pattern matches the save event, and simply logs the result, like this:

```
# pento/lib/pento_web/live/demographic_live/form_component.ex
def handle_event("save", %{"demographic" => demographic_params}, socket) do
  IO.puts("Handling 'save' event and saving demographic record...")
  IO.inspect(demographic_params)
  {:noreply, socket}
end
```

Now, if we visit /survey, fill out the demographics form and hit “save”, we should see the following log statements:

```
Handling 'save' event and saving  and saving demographic record...
%{"gender" => "female", "year_of_birth" => "1989"}
```

Perfect! Thanks to the `phx_target: @myself` attribute, our component is getting the event. Now, we can build our reducer to save the event:

```
defp save_demographic(socket, demographic_params) do
  case Survey.create_demographic(demographic_params) do
    {:ok, demographic} ->
      # coming soon!
      socket
    {:error, %Ecto.Changeset{} = changeset} ->
      assign(socket, changeset: changeset)
  end
end
```

Our component is responsible for managing the state of the demographic form and saving the demographic record. We lean on the context function, `Survey.create_demographic/1`, to do the heavy lifting. We need to handle both the success and error cases, and we do so. We save the implementation of the `:ok` case for later, and simply put the changeset back in the socket in the event of an `:error`. That way, the error tags in our form can tell our user exactly what to do to fix the form data.

Now, we need to call the reducer in the handler. Key in the following `handle_event/3` function to your `DemographicLive.FormComponent`:

```
stateful_components/pento/lib/pento_web/live/demographic_live/form_component.ex
def handle_event("save", %{"demographic" => demographic_params}, socket) do
  {:noreply, save_demographic(socket, demographic_params)}
end
```

We plug in the reducer, and we're off to the races. Our implementation is almost complete. We're left with one final question, what should our reducer do if the save succeeds? We'll look at that problem next.

Send a Message to the Parent

At a high level, when the form saves successfully, we should *stop* rendering the form and instead render the demographic's details. This sounds like a job for the SurveyLive view! After all, SurveyLive is responsible for managing the overall survey state.

If the SurveyLive is going to stop showing the demographic form and instead show the completed demographic details, we'll need some way for the form component to tell SurveyLive that it's time to do so. We need to send a message from the child component to the parent live view.

It turns out that it's easy to do so with plain old Elixir message passing via the `send` function.

Remember, our component is running in the parent's process and they share a pid. So, we can use the component's own pid to send a message to the parent. Then, we can implement a handler in the parent live view that receives that. It turns out that `handle_info/2` is the tool for the task.

Update `save_demographic/2` to send a message to the parent on success:

```
stateful_components/pento/lib/pento_web/live/demographic_live/form_component.ex
def save_demographic(socket, demographic_params) do
  case Survey.create_demographic(demographic_params) do
    {:ok, demographic} ->
      send(self(), {:created_demographic, demographic})
      socket

    {:error, %Ecto.Changeset{} = changeset} ->
      assign(socket, changeset: changeset)
  end
end
```

Now, we'll implement `handle_info/2` to teach the SurveyLive view how to respond to our message.

```
stateful_components/pento/lib/pento_web/live/survey_live.ex
def handle_info({:created_demographic, demographic}, socket) do
  {:noreply, handle_demographic_created(socket, demographic)}
end
```

The function head of `handle_info/2` matches our message—a tuple with the message name and a payload containing the saved demographic—and receives the socket. As usual, we want skinny handlers, so we call the `handle_demographic_created/2` reducer to do the work. Now, we need to decide exactly what work to do in the `handle_demographic_created/2` function.

Let's add a flash message to the page to indicate to the user that their demographic info is saved, and let's store the newly created demographic in the survey state by adding it to `socket.assigns`. Define your `handle_demographic_create/2` to do exactly that:

```
stateful_components/pento/lib/pento_web/live/survey_live.ex
def handle_demographic_created(socket, demographic) do
  socket
  |> put_flash(:info, "Demographic created successfully")
  |> assign(:demographic, demographic)
end
```

We pipe our socket through functions to store a flash message and add the `:demographic` assign key to our socket. Now, we are ready to act on that data.

At this point, we want the `SurveyLive` to *stop* rendering the `DemographicLive.FormComponent`, so the corresponding template needs to get just a little bit smarter. If the demographic is present, the template should display the demographic details. Otherwise, it should display the form component. Go ahead and add this logic to your template:

```
<%= if @demographic do %>
  <h3>Demographics</h3>
  <ul>
    <li>Year of birth: <%= @demographic.year_of_birth %></li>
    <li>Gender: <%= @demographic.gender %></li>
  </ul>
<% else %>
  <%= live_component @socket,
    PentoWeb.DemographicLive.FormComponent,
    user: @current_user,
    id: "demographic-form-#{@current_user.id}" %>
<% end %>
```

Perfect.

We have one problem though. As written, this template will fail when we initially mount the live view—our live view only adds the `:demographic` key to `assigns` *after* the user submits the form and saves the record successfully. That means we need to tweak `mount/3` function to query for the demographic and add it to `socket.assigns`. Update your `mount/3` function with this new reducer pipeline:

```
def mount(_params, %{"user_token" => token}, socket) do
  {:ok,
   socket
   |> assign_user(token)
   |> assign_demographic()}
end
```

As usual, we delegate this job to a single-purpose reducer, increasing the library of functions we can reuse. Recall we already have a context function to do the job, so let's write that reducer now:

```
stateful_components/pento/lib/pento_web/live/survey_live.ex
def assign_demographic(%{assigns: %{current_user: current_user}} = socket) do
  assign(socket, :demographic, Survey.get_demographic_by_user(current_user))
end
```

It is a short, single-purpose function that does exactly what you'd expect, and that's the mark of good Elixir.

It's finally time to put all of this code together. Now, when we point our browser to /survey, and submit the demographic form, we should see the flash message, and we'll also see the form replaced with the demographic details, as in this image:

Demographic created successfully

Survey

Demographics ✓

Gender: other

Year of birth: 1961

If you refresh the page, everything works as expected because our mount/3 function correctly sets the demographic data.

The user interface in our template looks OK, but the code is starting to get a little messy. Luckily, we can clean this up by wrapping up the demographic details markup in a stateless component. In this way, we can assemble multiple components into one coherent view.

Let's build a simple counterpart to our form component, the one that we'll show when demographic data exists.

Show a Demographic

Our new component won't do too much. It will just make the right details available for a user who has filled out a demographic. This component doesn't

need to respond to any events. All it needs to do is render the demographic details in some markup. A stateless component should do the trick here.

First, we'll implement the component and its template. Then, we'll render the component from within the SurveyLive view. We'll allow SurveyLive to handle the logic relating to the state of the overall survey—like whether to show the demographic form or the demographic details—while breaking out individual pieces of the survey page into their own smaller, more manageable parts. With this layering of components, LiveView allows us to build complex single-page flows with ease.

To begin, define the component:

```
stateful_components/pento/lib/pento_web/live/demographic_live/show_component.ex
defmodule PentoWeb.DemographicLive.ShowComponent do
  use PentoWeb, :live_component
end
```

The component is an empty shell. The `use PentoWeb, :live_component` line does all of the heavy lifting for us. We'll pick up all of the default callbacks, and we'll relegate rendering to our template.

```
stateful_components/pento/lib/pento_web/live/demographic_live/show_component.html.leex
<div class="survey-component-container">
  <h2>Demographics <i class="fa fa-check survey"></i></h2>
  <ul>
    <li>Gender: <%= @demographic.gender %></li>
    <li>Year of birth: <%= @demographic.year_of_birth %></li>
  </ul>
</div>
```

The `DemographicLive.ShowComponent` nicely compartmentalizes the markup for the completed demographic, including a snazzy green check to indicate a completed demographic.

Now, we can render it from our template, below the header, with a call to `live_component/3`:

```
stateful_components/pento/lib/pento_web/live/survey_live.html.leex
<%= if @demographic do %>
  <%= live_component @socket,
    PentoWeb.DemographicLive.ShowComponent,
    demographic: @demographic %>
```

There's no `:id`, so it's stateless. When you need to send an event to a live view, you'd need to specify which component gets the event. The `:id` key serves that purpose, so if there's no `:id`, you can't send it events, and it's stateless.

Let's see it in action. If a user who has not filled out the demographic form visits the page, they will see the form rendered:

Survey

Demographics

Gender

female ▼

Year of birth

2020 ▼ **SAVE**

And if a user who *has* filled out the demographic form visits the page, they will see their demographic details rendered:

Survey

Demographics ✓

Gender: female

Year of birth: 1998

That's exactly what we want, so the demographics are done. Now, we can move on to the ratings portion.

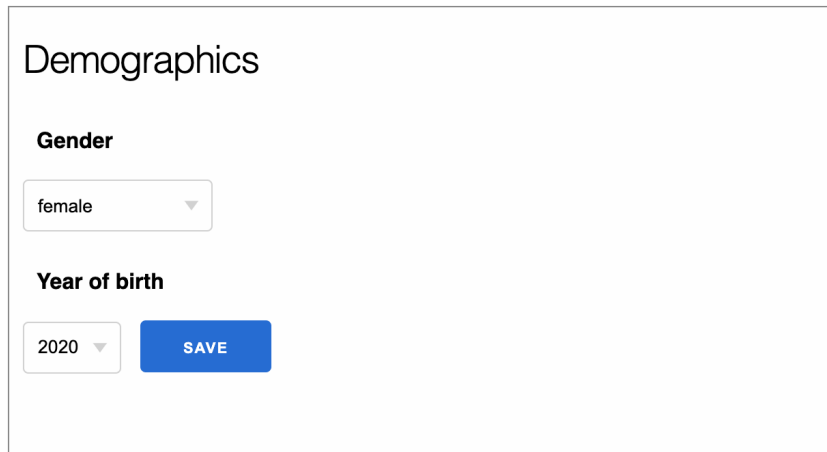
Build The Ratings Components

We're going to do very much the same thing we did with demographics—let the SurveyLive view orchestrate the state and appearance of the overall survey and devise a set of components to handle the state of the product ratings.

We'll have the SurveyLive template implement some logic to display product rating components only if the demographic form is complete and the demographic exists. If there's an existing demographic, we'll render a ratings index component that will iterate over the products and render the rating details or rating form accordingly.

Again, here's roughly what a user will see if they've not yet entered demographic data:

Survey



Demographics

Gender

female ▼

Year of birth

2020 ▼ **SAVE**

Notice we present a form for demographic data, but no product ratings.

And this is what a user will see after completing the demographic form:

Survey

Demographics ✓

Gender: female

Year of birth: 2003

Ratings

Chess

Stars

5 ▼

SAVE

Tic-Tac-Toe

Stars

5 ▼

SAVE

Table Tennis

Stars

5 ▼

SAVE

Our code doesn't give the user a chance to enter any product rating data until they've given us demographics. After that, they can rate a product.

That means our live view will have a lot to manage. But, by organizing our code with components, we'll avoid the needless complexity.

We'll have an index component to hold the whole list of ratings, a show component to show a completed rating, and form component to manage the form for a single rating. In this way, we'll maintain a nice separation of concerns. The SurveyLive will manage the state of the survey form, implementing logic that dictates whether to show the ratings index component. And the ratings index component will manage the state of product ratings, implementing logic that dictates whether to show rating details or rating forms.

Let's begin with a ratings index component that the SurveyLive template can render.

List Ratings

We'll build a ratings index component that will be responsible for orchestrating the state of *all* of the product ratings in our survey. This component will iterate over the products and determine whether to render the rating details if a rating by the user exists, or the rating form if it doesn't. The responsibility for rendering rating details will be handled by a stateless “rating show” component and the responsibility for rendering and managing a rating form will be handled by a stateful “rating form” component.

Meanwhile, SurveyLive will continue to be responsible for managing the overall state and appearance of the survey page. Only if the demographic record exists for the user will the SurveyLive view render the ratings index component.

In this way, we keep our code organized and easy to maintain because it is adherent to the single responsibility principle—each component has one job to do. By layering these component within the parent SurveyLive view, we are able to compose a series of small, manageable pieces into one interactive feature—the user survey page.

We'll begin by implementing the RatingsLive.IndexComponent. Then, we'll move on to the rating show component, followed by the rating form component. Let's get started.

Build the Ratings Index Component

Our RatingsLive.IndexComponent will be stateless—it doesn't need to respond to any events. All it needs to do is iterate over products and show a rating or a form accordingly. Additionally, this component won't need to take advantage of any lifecycle callbacks. We'll use the default implementation of each callback. The default mount/1 simply returns the socket, and the default update/2

function takes any assigns you pass in and adds them to the socket. Keep in mind that whether or not you provide an implementation, the lifecycle remains the same: mount/1, then update/2, then render/1.

Create a file, `lib/pento_web/live/rating_live/index_component.ex`, and key in the following component definition, closing it off with an `end`:

```
stateful_components/pento/lib/pento_web/live/rating_live/index_component.ex
defmodule PentoWeb.RatingLive.IndexComponent do
  use PentoWeb, :live_component
```

The only function our `RatingsLive.IndexComponent` needs to implement is a tiny helper function to determine whether the current user has completed ratings for *all* of the products. We'll define a function, `ratings_complete?`, that does exactly that:

```
stateful_components/pento/lib/pento_web/live/rating_live/index_component.ex
def ratings_complete?(products) do
  Enum.all?(products, fn product ->
    length(product.ratings) == 1
  end)
end
```

We'll use this function in the template to show a nice green check mark next to the "Ratings" heading if a user has completed all of the product ratings.

Let's implement the component's template now. First, it will render a heading and contain some logic to display that check mark if all of the products have been rated by the user:

```
stateful_components/pento/lib/pento_web/live/rating_live/index_component.html.leex
<h2>Ratings <%= if ratings_complete?(@products) do %>
  <i class='fa fa-check survey'></i>
  <%= end %>
</h2>
```

If the `ratings_complete?/1` function returns true, meaning the user has submitted a rating for all of the available products, then our template will display a helpful green check mark favicon.

Finally, our component's template will move on to it's main job—showing a product rating if it exists, and showing the rating form if it doesn't:

```
stateful_components/pento/lib/pento_web/live/rating_live/index_component.html.leex
<%= for {product, index} <- Enum.with_index(@products) do %>
  <%= if rating = List.first(product.ratings) do %>
    <%= live_component @socket,
      PentoWeb.RatingLive.ShowComponent,
      rating: rating,
      product: product%>
```

```

<% else %>
<%= live_component @socket,
  PentoWeb.RatingLive.FormComponent,
  user: @current_user,
  product: product,
  product_index: index,
  id: "product-#{product.id}-form" %>
<% end %>
<% end %>

```

Here we have a for comprehension that maps over all of the products in the system, where each product's ratings list contains the single preloaded rating by the given user, if one exists.

Inside that comprehension, the template renders rating details with `RatingLive.ShowComponent` or renders the `RatingLive.FormComponent`, depending on whether the rating exists. Nesting components in this manner lets the reader of the code deal with a tiny bit of complexity at a time.

We'll dig into this logic a bit more when we're ready to implement our `RatingLive.ShowComponent` and `RatingLive.FormComponent`.

With the index component out of the way, we are finally ready to weave it into our `SurveyLive` template.

Render the Component

The next bit of code we'll write shows how the presentation of our view can change based on the contents of the socket. The `SurveyLive` view will use the state of the overall survey to control what is shown to the user on the page. Specifically, the template will determine what to show based on whether a demographic exists.

In `SurveyLive`, we query for a demographic and store the results of that query in the socket. If no demographic exists, and the socket assigns key of `:demographic` points to nil, the template renders the form. Otherwise, we render the demographic show component and call on the `RatingLive.IndexComponent` to add the product ratings to our view.

Let's build out this logic now. Open up the `SurveyLive` template, and look for the `live_component/3` for the `DemographicLive.ShowComponent`. Beneath it, add the code to render `RatingsLive.IndexComponent`, shown here:

```
stateful_components/pento/lib/pento_web/live/survey_live.html.leex
```

```

<%= if @demographic do %>
  <%= live_component @socket,
    PentoWeb.DemographicLive.ShowComponent,
    demographic: @demographic %>

```

```
<%= live_component @socket,
  PentoWeb.RatingLive.IndexComponent,
  products: @products,
  current_user: @current_user%>
```

Perfect. Now our view renders the component that will present ratings. To make that work, we need to pass the list of products to the `RatingLive.IndexComponent` so that the component can iterate over them to render ratings for each one. In the `SurveyLive` template, we pass the list, `@products`, to our component, but we haven't added it to the live view socket yet. Let's fix that now.

Update the `mount/3` function of `SurveyLive` to query for products and their associated rating by the given user and put them in assigns.

```
stateful_components/pento/lib/pento_web/live/survey_live.ex
def mount(_params, %{"user_token" => token}, socket) do
  {:ok,
   socket
   |> assign_user(token)
   |> assign_demographic()
   |> assign_products()}
end
```

We're up to our old tricks, building another reducer called `assign_products/1` to do the work:

```
stateful_components/pento/lib/pento_web/live/survey_live.ex
def assign_products(%{assigns: %{current_user: current_user}} = socket) do
  assign(socket, :products, list_products(current_user))
end

defp list_products(user) do
  Catalog.list_products_with_user_ratings(user)
end
```

We use our `Catalog` context and the `assign/2` function to drop the requisite key/value pair into our socket. Now that we're rendering our `RatingLive.IndexComponent` with the product list, let's build the stateless `RatingLive.ShowComponent` that will conditionally show existing ratings for a product.

Show a Rating

We're getting closer to the goal of showing ratings, step by step. Remember, we'll show the ratings that exist, and forms for ratings otherwise. Let's cover the case for ratings that exist first. We'll define a stateless component to show a rating. Then, we'll render that component from within the `RatingLive.IndexComponent`. Let's get started.

Build the Rating Show Component

Implement a simple `RatingLive.ShowComponent` module to implicitly render the template. Create a file, `lib/pento_web/live/rating_live/show_component.ex`, and key this in:

```
stateful_components/pento/lib/pento_web/live/rating_live/show_component.ex
defmodule PentoWeb.RatingLive.ShowComponent do
  use PentoWeb, :live_component
```

As you can see here, the template is also pretty slim since it only needs the product name and star rating.

```
stateful_components/pento/lib/pento_web/live/rating_live/show_component.html.leex
<h4>
  <%= @product.name%>:
  <%= raw render_rating_stars(@rating.stars) %>
</h4>
```

It's content is fairly trivial, rendering only the product name and some raw star rating HTML. The last thing we need is to implement the `render_rating_stars/1` helper function. We can add it to the component, like this:

```
stateful_components/pento/lib/pento_web/live/rating_live/show_component.ex
def render_rating_stars(stars) do
  filled_stars(stars)
  |> Enum.concat(unfilled_stars(stars))
  |> Enum.join(" ")
end

def filled_stars(stars) do
  List.duplicate("<span class='fa fa-star checked'></span>", stars)
end

def unfilled_stars(stars) do
  List.duplicate("<span class='fa fa-star'></span>", 5 - stars)
end
```

The `render_rating_stars/1` function is interesting. It uses the `List.duplicate/2` function to produce a list of filled-in, or “checked”, star favicons corresponding to the number of stars that the product rating has. Then, it uses `List.duplicate/2` again to produce a list of empty, or not checked, star favicons for the remaining number of stars. For example, if the number of stars in the rating is 3, this function will create a list of three checked stars and two un-checked stars. Lastly, the function concatenates the two lists together and joins them into a string of HTML that we can render in the template.

We have everything we need to display a completed rating, so it's time to roll several components up together. Let's see how it works.

Render the Component

Now we're ready to implement the next phase of our plan. The `RatingLive.IndexComponent` should iterate over the list of products in the `@products` assigns. If a rating is present, we'll show it. Here's the code to implement this behavior:

```
stateful_components/pento/lib/pento_web/live/rating_live/index_component.html.leex
<%= for {product, index} <- Enum.with_index(@products) do %>
  <%= if rating = List.first(product.ratings) do %>
    <%= live_component @socket,
      PentoWeb.RatingLive.ShowComponent,
      rating: rating,
      product: product%>
  <%= else %>
```

It's a straight for comprehension with an if statement. If a rating exists, we render the component with the familiar `live_component/3`, passing the rating and product. If not, we need to render the form. The else logic is omitted for now, since we still need to build the form. Let's do that now.

Show the Rating Form

Our rating form will display the form and manage its state, validating and saving the rating. We'll need to pass a product and user for our relationships, and also the product's index in the parent `LiveView`'s `socket.assigns.products`. We'll use this index later on to update `SurveyLive` state efficiently.

Build the Rating Form Component

The component will be stateful, and we'll need to use `update/2` to stash our rating and changeset in the socket. Create a new file, `lib/pento_web/rating_live/form_component.ex` and define a component, `PentoWeb.RatingLive.FormComponent`. Then, key in this update function:

```
stateful_components/pento/lib/pento_web/live/rating_live/form_component.ex
defmodule PentoWeb.RatingLive.FormComponent do
  use PentoWeb, :live_component
  alias Pento.Survey
  alias Pento.Survey.Rating

  def update(assigns, socket) do
    {:ok,
     socket
     |> assign(assigns)
     |> assign_rating()
     |> assign_changeset{}}
  end
end
```

These reducer functions will add the necessary keys to our `socket.assigns`. They'll drop in any assigns our parent sends, add a new `Rating` struct, and finally establish a changeset for the new rating. Here's a closer look at the our “add rating” and “add changeset” reducers:

```
stateful_components/pento/lib/pento_web/live/rating_live/form_component.ex
def assign_rating(%{assigns: %{user: user, product: product}} = socket) do
  assign(socket, :rating, %Rating{user_id: user.id, product_id: product.id})
end

def assign_changeset(%{assigns: %{rating: rating}} = socket) do
  assign(socket, :changeset, Survey.change_rating(rating))
end
```

There are no surprises here. One reducer builds a new rating, and the other uses the `Survey` context to build a changeset for that rating. Now, on to render.

With our socket established, we're ready to render. As usual, we'll choose a template to keep our markup code neatly compartmentalized. Create a file, `lib/pento_web/live/rating_live/form_component.html.leex`. Add the product title markup followed by the product rating form shown here:

```
stateful_components/pento/lib/pento_web/live/rating_live/form_component.html.leex
<div class="survey-component-container">
  <section class="row">
    <h4><%= @product.name %></h4>
  </section>
  <section class="row">
    <%= f = form_for @changeset, "#",
      id: @id,
      phx_target: @myself,
      phx_change: "validate",
      phx_submit: "save" %>
    <section class="row">
      <%= label f, :stars%>
      <%= select f, :stars, Enum.reverse(1..5) %>
      <%= error_tag f, :stars %>

      <%= hidden_input f, :user_id%>
      <%= hidden_input f, :product_id%>

      <%= submit "Save", phx_disable_with: "Saving..." %>
    </section>
  </form>
</section>
</div>
```

We bind two events to the form, a `phx_change` to send a validate event and a `phx_submit` to send a save event. We target our form component to receive events by setting `phx-target` to `@myself`, and we tack on an `id`. Note that we've set a

dynamic HTML id of the stateful component id, stored in socket assigns as @id. This is because the product rating form will appear multiple times on the page, once for each product, and we need to ensure that each form gets a unique id.

Our form has a stars field with a label and error tag, and also a hidden field for each of the user and product relationships. We tie things up with a submit button.

We'll come back to the events a bit later. For now, let's fold our work into the RatingLive.IndexComponent.

Render the Component

The RatingLive.IndexComponent template should render the rating form component if no rating for the given product and user exists. Let's do that now.

[stateful_components/pento/lib/pento_web/live/rating_live/index_component.html.leex](#)

```
<% else %>
<%= live_component @socket,
  PentoWeb.RatingLive.FormComponent,
  user: @current_user,
  product: product,
  product_index: index,
  id: "product-#{product.id}-form" %>
<% end %>
```

Here, we call on the component with live_component/3, passing the user and product into the component as assigns, along with the product's index in the @products assignment. We add an :id, so our rating form component is stateful. Since we'll only have one rating per component, our id with an embedded product.id should be unique.

It's been a while since we've been able to look at things in the browser, but now, if you point your browser at /survey, you should see something like this:

Survey

Demographics ✓

Gender: female

Year of birth: 2003

Ratings

Chess

Stars

5 ▼

SAVE

Tic-Tac-Toe

Stars

5 ▼

SAVE

Table Tennis

Stars

5 ▼

SAVE

Handle Component Events

You know the drill by now. We've created events to save and validate our form, so we should teach our component how to do both. We need one `handle_event/2` function head for each of the save and validate events. Let's start with `validate`:

```
stateful_components/pento/lib/pento_web/live/rating_live/form_component.ex
def handle_event("validate", %{"rating" => rating_params}, socket) do
  {:noreply, validate_rating(socket, rating_params)}
end
```

You've seen these handlers before, so you know we're matching events, and that we need to build the reducer next:

```
stateful_components/pento/lib/pento_web/live/rating_live/form_component.ex
def validate_rating(socket, rating_params) do
  changeset =
    socket.assigns.rating
    |> Survey.change_rating(rating_params)
    |> Map.put(:action, :validate)

  assign(socket, :changeset, changeset)
end
```

Our `validate_rating/2` reducer function validates the `changeset` and returns a new `socket` with the validated `changeset` (containing any errors) in `socket.assigns`. This will cause the component to re-render the template with the updated `changeset`, allowing the `error_tag` helpers in our `form_for` form to render any errors.

Next up, we'll implement a `handle_event/2` function that matches the `save` event:

```
stateful_components/pento/lib/pento_web/live/rating_live/form_component.ex
def handle_event("save", %{"rating" => rating_params}, socket) do
  {:noreply, save_rating(socket, rating_params)}
end
```

And here's the reducer:

```
stateful_components/pento/lib/pento_web/live/rating_live/form_component.ex
def save_rating(
  %{assigns: %{product_index: product_index, product: product}}
  = socket,
  rating_params
) do
  case Survey.create_rating(rating_params) do
    {:ok, rating} ->
      product = %{product | ratings: [rating]}
      send(self(), {:created_rating, product, product_index})
      socket
```

```

{:error, %Ecto.Changeset{} = changeset} ->
  assign(socket, changeset: changeset)
end
end

```

Just as we did in the demographic form component, we attempted to save the form. On failure, we assign a new changeset. On success, we send a message to the parent live view to do the heavy lifting for us. Then, as all handlers must do, we return the socket.

Update the Rating Index

What should happen when the rating is successfully saved? The `RatingLive.Index` Component should no longer render the component form. Instead, the survey should display the saved rating for the given product. This kind of state change is squarely the responsibility of the `SurveyLive`. Our message will serve to notify the parent live view to change.

Here's the interesting bit. All the parent really needs to do is update the socket. The `RatingLive.Index` Component already renders the right thing based on the contents of the socket. All we need to do is implement a handler to deal with the “created rating” message.

```

stateful_components/pento/lib/pento_web/live/survey_live.ex
def handle_info({:created_rating, updated_product, product_index}, socket) do
  {:noreply, handle_rating_created(socket, updated_product, product_index)}
end

```

We use a `handle_info`, just as we did before with the demographic. Now, our reducer can take the appropriate action. Notice that the message we match has a message name, an updated product *and* its index in the `:products` list. We can use that information to update the product list, without going back to the database. We'll implement the reducer below to do this work:

```

stateful_components/pento/lib/pento_web/live/survey_live.ex
def handle_rating_created(
  %{assigns: %{products: products}} = socket,
  updated_product,
  product_index
) do
  socket
  |> put_flash(:info, "Rating submitted successfully")
  |> assign(
    :products,
    List.replace_at(products, product_index, updated_product)
  )
end

```

The `handle_rating_created/3` reducer adds a flash message and updates the product list with its rating. This causes the template to re-render, passing this updated product list to the `RatingLive.IndexComponent`. That component in turn knows just what to do with a product that does contain a rating by the given user—it will render that rating’s details instead of a rating form.

Notice the lovely layering. In the parent live view layer, all we need to do is manage the list of products and ratings. All of the form handling and rating or demographic details go elsewhere.

The end result of a submitted rating leads is an updated product list and a flash message. Submit a rating, and see what happens:

Rating submitted successfully

Survey

Demographics ✓

Gender: female

Year of birth: 2003

Ratings

Chess: ★ ★ ★ ★ ★

Tic-Tac-Toe

Stars

5 ▼

SAVE

Table Tennis

Stars

5 ▼

SAVE

You just witnessed the power of components, and LiveView.

Your Turn

Stateful components are useful for extracting common code that will be *rendered*. Though every component has a socket with state, only stateful components can *modify* that state. In this chapter, you turned your *stateless* component into a *stateful* one. The techniques we applied here can work to make any stateless component stateful.

To transform a stateless component, just add an `id` assign key. Then, you can add event handlers to process events, and specify a `phx-target` with another component's `id`, or the `@myself` attribute.

Stateful components are a bit more complex than stateless ones, but they're a bit more powerful too. A stateful component's lifecycle is similar to that of a stateless one, except for the addition of the `preload/1` callback at the beginning of the process. Then, after a stateful component is rendered, events will come in to `handle_event/3` functions which can change component state and trigger a re-render.

With our set of stateless and stateful components, we've built out a fully interactive survey feature in a way that is sane, organized, and easy to maintain. By breaking out the specific responsibilities of the survey page into discrete components, we keep our code adherent to the single responsibility principle. LiveView then allows us to layer those components, composing them into one single-page flow orchestrated by the parent live view, `SurveyLive`. In this way, LiveView lets us build complex interactive features quickly and easily.

Now that you have a fully functioning set of components, it's your chance to put what you've learned into practice.

Give It a Try

These problems will let you extend what we've already done.

- Stateful components are often tied to backend database services—our `DemographicLive.FormComponent` is backed by the `Survey` context, which wraps interactions with the `Demographic` schema. Add a field to the `Demographic` schema and corresponding database table to track the education level of a user, allowing them to choose from 'high school', 'bachelor's degree', 'graduate degree', 'other' or 'prefer not to say'. Then, update your LiveView code to support this field in the `demographic` form.
- Stateful components are handy for building widgets you might normally build with JavaScript. Build a component that toggles a button showing

either + expand or - contract, and then marks a corresponding div as hidden or visible. Under what circumstances would you use a CSS style with `display: none`, versus rendering/removing the whole div? Hint: think about how many bytes LiveView would need to move and when it would move them.

Next Time

Now we have a set of components for collecting survey data, but nowhere to aggregate that data. In the next chapter, we'll review many of the techniques you've seen in the first part of this book as we build an admin dashboard that allows us to view survey results and more. Since this dashboard is built with LiveView, it will be more interactive than typical dashboards.

Part III

Extend LiveView

In Part III, we'll build another custom LiveView feature that we'll extend with Phoenix PubSub-backed capabilities in order to support real-time interactions. We'll use LiveView communication mechanisms, along with PubSub, to build an admin dashboard that reflects not only the state of the page, but of the application at large. We'll wrap up with a look at LiveView testing to ensure that our admin dashboard is well tested.

Build an Interactive Dashboard

In the previous part, we completed a Survey tool our company will use to collect data from our customers. In the next two chapters, we're going to build an interactive dashboard that tracks real-time data, including survey data, as it flows into our system. This chapter will focus on building the dashboard, and the next will integrate real-time data feeds into that dashboard. Interactive views presenting data synchronized in real-time is a perfect LiveView use case and you'll see how you can extend a custom live view with the help of Phoenix PubSub in order to support such synchronization.

Many dashboards fall into one of two traps. Some are afterthoughts, seemingly slapped together at the last moment. These views are often casualties of a time crunch. Other live views have lots of interactive bells and whistles, but they lack the impact they might otherwise have because the dashboard shows content that lags behind the needs of the organization. LiveView can help solve both of these common problems by making it easy to quickly put together components that snap seamlessly into LiveView's overall architecture.

In this chapter, you'll discover how easy it can be to build a dashboard that does what your users need, but also fits into the quick development cycle-times most organizations require. When you're done, you'll have more experience writing core and boundary functions in Phoenix, and more experience composing live views with components. You'll also be able to use libraries that leverage SVG to render graphics, and wrap them into APIs that are easy to consume.

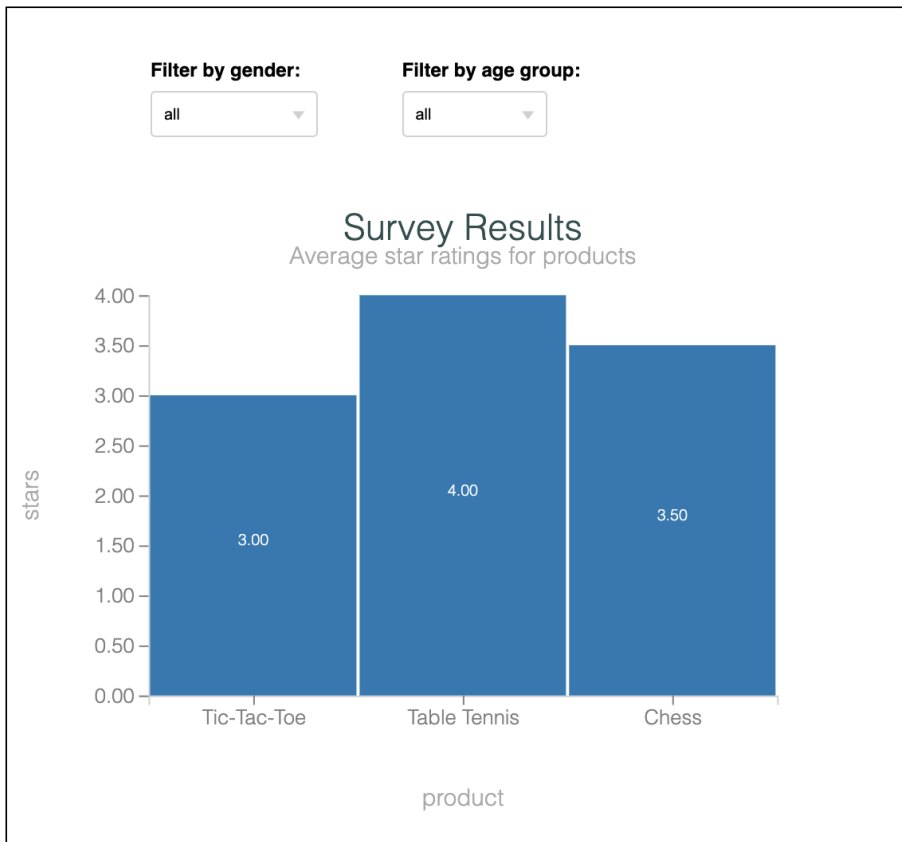
Let's make a plan.

The Plan

Our interactive dashboard will show the health of our products with a glance. It will have several different elements on the page. A survey component will display survey results for each product and its average star rating. In the next chapter, we'll add a real-time list of users and we'll supercharge our survey results chart by enabling it to update in real-time, as new results come in.

Here's a rough mock up of what our users say they want:

Survey Results



User Activity

Active users currently viewing games

Chess

- samantha@email.com

In this chapter, we'll focus on building the interactive survey results chart portion of our dashboard. Tracking customer satisfaction is critical for a game company's marketing, so the survey results chart will show the average survey

star rating for each product. To assist our marketing efforts, we'll let our users visualize star ratings across demographic groups.

The dashboard will be its own live view. We'll delegate the responsibilities of presenting survey data to a component.

We'll start by leveraging the CRC pattern to define a core module that composes the queries we need, and a context in which to execute them.

Then, we'll wrap that much in a live view with a survey results component, and use an SVG graphics charting library to display data on the page.

Finally, we'll make our chart interactive by providing a simple form letting the user filter survey data by demographics.

When we're done, we'll use the common `_using_` macro to make our chart helper functions easier to use.

We'll need three things to kick things off. We'll define the view in the `AdminDashboardLive` `LiveView`. Then, we'll wire that view to a live route. Finally, we'll delegate the survey data on the page to a stateful component called `SurveyResultsLive`.

Let's start things off with the live view.

Define The `AdminDashboardLive` `LiveView`

The socket for the live view will have a bit of state that models the page we're trying to build. For a dashboard, the state is the data we're presenting. The dashboard we're building will represent each major section of the page as a *component*. That means the socket of the live view itself will be pretty empty—instead, the socket of each component will hold the data that the component is responsible for rendering.

Define `AdminDashboardLive` in a file called `lib/pento_web/admin_dashboard_live.ex`, and key in the ceremony at the top of the file along with the initial `mount/3` function.

```
interactive_dashboard/pento/lib/pento_web/live/admin_dashboard_live.ex
defmodule PentoWeb.AdminDashboardLive do
  use PentoWeb, :live_view

  def mount(_params, _session, socket) do
    {:ok,
     socket
     |> assign(:survey_results_component_id, "survey-results")}
  end
end
```

We declare our module, and establish our file as a live view with the `use PentoWeb, :live_view` directive. Next, we move on to the `mount/3` statement. One of our requirements is for users to be able to filter survey results by demographic.

So, a component will need to implement a form and respond to events. This means it will need to be stateful. Stateful components are called with an id. We'll put this ID in the parent live view's state—you'll see why the parent needs awareness of this ID in the next chapter.

For now, let's add this code to connect our route in router.ex:

```
interactive_dashboard/pento/lib/pento_web/router.ex
scope "/", PentoWeb do
  pipe_through [:browser, :require_authenticated_user]
  live "/admin-dashboard", AdminDashboardLive, :index
end
```

This route is for browser users who are logged in, so the route uses pipe_through with both the browser and require_authenticated_user pipelines. Now, we'll get all of the benefits of the browser pipeline in router.ex and the require_authenticated_user plug we created in [Chapter 2, Phoenix and Authentication, on page 31](#).

Now, we can start with just enough of a template to test out our new view. Create the file live/admin_dashboard_live.html.leex and add just a simple header, like this:

```
interactive_dashboard/pento/lib/pento_web/live/admin_dashboard_live.html.leex
<section class="row">
  <h1>Admin Dashboard</h1>
</section>
```

There's not much in there for now, but we do have a header to show whether the code is working or not. Now, you can start your server and point your browser to /admin-dashboard to see the sparse, but working, view:



Phoenix Framework

[Get Started](#)
[LiveDashboard](#)

robert@email.com
[Settings](#)
[Log out](#)

Admin Dashboard

One of the nice things about LiveView is that you can often stand up a new page in a few minutes, and then build many quick iterations from there. Now we're ready to build the SurveyResultsLive component.

Represent Dashboard Concepts with Components

A dashboard is a metaphor for all of the gauges on a complex machine, where each gauge is a self-contained component. We're going to model this concept with code, putting each major concept in its own LiveView component. That strategy will let us isolate all of the code for each isolated concept.

Let's kick things off with the `SurveyResultsLive` component, which will be responsible for the survey results chart that displays interactive product ratings.

Create a Component Module

We'll start by implementing a basic component and template that don't do much. Since the component will eventually need to handle events so the user can ask for demographic data, we know the component will be stateful, so we'll give it an ID. We'll also render it from `AdminDashboardLive` to make sure everything's working.

Open up `lib/pento_web/live/survey_results_live.ex` for the initial component skeleton:

```
defmodule PentoWeb.SurveyResultsLive do
  use PentoWeb, :live_component
end
```

That's all for now. There's no `render/1`, so we need a template. Let's do that next.

Build the Component Template

We'll start by building a section with a heading. Key this into `live/survey_results_live.html.leex`:

```
interactive_dashboard/pento/lib/pento_web/live/survey_results_live.html.leex
<section class="row">
  <h1>Survey Results</h1>
</section>
```

It's just a section and a header, but that's enough. In the spirit of getting small wins and iterating quickly, let's stand that much up. Our component is stateful, so we'll need to call `live_component/3` with the `:id` we specified in `mount/3` earlier. Render the component statefully from the `admin_dashboard_live.leex` template, as shown here:

```
interactive_dashboard/pento/lib/pento_web/live/admin_dashboard_live.html.leex
<%= live_component @socket,
  PentoWeb.SurveyResultsLive,
  id: @survey_results_component_id %>
```

Perfect. We supply the socket, the component's module, and the id from socket.assigns. Point your browser at /admin-dashboard:



Phoenix Framework

[Get Started](#)
[LiveDashboard](#)

robert@email.com

[Settings](#)
[Log out](#)

Admin Dashboard

Survey Results

Excellent. Now that everything is wired up and running, we're ready to build the survey results bar chart.

Fetch Survey Results Data

To go much further, we're going to need data, so we'll switch gears from the view and focus on the backend service. In order to render products and their average star ratings in a chart, the live view must be able to query for this data in the form of a list of product names and their associated average star ratings.

This will be a good time to practice good Phoenix design. You'll add a new API function to the Catalog context to make requests to the database. Your context function will rely on new query functions in the core to extract exactly the data it needs. Separating these concerns will keep the codebase organized and beautiful.

Shape the Data With Ecto

The format of the data is somewhat dictated by the manner in which we will need to feed it into our chart. We'll provide the exact details later. For now, it's enough to understand that we need to fetch a list of products and average ratings as a list of tuples, as in this example:

```
[
  {"Tic-Tac-Toe", 3.4285714285714284},
  {"Table Tennis", 2.5714285714285716},
  {"Chess", 2.625}
]
```

With any luck, Ecto can return data in exactly the shape we need, but first we need to decide where the queries should go. If we make sure to validate any data before it ever reaches the query layer, the process of building a query should not ever fail unless there's a bug in our code—in other words, the process is certain and predictable, exactly the kind of job that belongs in the core. So, we'll create a query builder module, `Pento.Catalog.Product.Query` in our application's core.

We'll need a query to fetch products with average ratings, so we'll build a few reducers in the `Pento.Catalog.Product.Query` module to shape a query that does just that. We'll use Ecto where clauses to select the right demographic, a join clause to pluck out the ratings for relevant users, a `group_by` clause to provide the average statistic, and a select clause to pluck out the tuples that match the required shape. That's a bit much to add to one giant function, but we know how to break the code down into single-purpose reducers. Take a look at the following functions:

```
interactive_dashboard/pento/lib/pento/catalog/product/query.ex
def with_average_ratings(query \\ base()) do
  query
  |> join_ratings
  |> average_ratings
end

defp join_ratings(query) do
  query
  |> join(:inner, [p], r in Rating, on: r.product_id == p.id)
end

defp average_ratings(query) do
  query
  |> group_by([p], p.id)
  |> select([p, r], {p.name, fragment("?::float", avg(r.stars))})
end
```

As usual, our module starts with a constructor, `base/0`, and pipes that query through a set of two reducers—one that joins products on ratings, and another that selects the product name and the average of its ratings' stars.

Let's see our query in action now.

Test Drive the Query

Make sure you alias `Pento.Survey.Rating` at the top of the `Catalog.Product.Query` module. Then, open up IEx and execute the query as follows:

```
iex> alias Pento.Catalog.Product
Pento.Catalog.Product
iex> alias Pento.Repo
```

```
Pento.Repo
iex> Product.Query.with_average_ratings() |> Repo.all()
...
[
  {"Tic-Tac-Toe", 3.4285714285714284},
  {"Table Tennis", 2.5714285714285716},
  {"Chess", 2.625}
]
```

Excellent. That's the exact format that the graphics library needs, so we don't need to do any further processing. Now, it's time to leave the calm, predictable world of the core for the chaotic, failure-prone world of the boundary.

Extend the Catalog Context

Where the core is calm and predictable, the boundary, or the context, is more complex because it might fail. The code in the boundary isn't *always* more complex, but it does have responsibilities that the core does not. The context must validate any data from external sources, usually with changesets. If a function might return an `{:ok, result}` or an `{:error, reason}` tuple, it falls on the context to do something about that failure.

Luckily, our context function doesn't have validation or error conditions to worry about, so our context function will be blissfully short and simple. Still, the new API should go in the context as a reminder that any data must be validated, and errors must be handled appropriately. Define a context function in the Catalog module to execute the new query:

```
interactive_dashboard/pento/lib/pento/catalog.ex
def products_with_average_ratings do
  Product.Query.with_average_ratings()
  |> Repo.all()
end
```

We feed the query into `Repo.all/1` and we're off to the races.

Your Turn: Verify the API in IEx

It's important to verify results as you go. Try out the new context API in IEx. Come back when you're ready to integrate the results our live view.

Initialize the SurveyResultsLive Component State

Now, our application's core contains all of the functions we need to fetch bar chart data in our live view. In this section, we'll teach the `SurveyResultsLive` component to fetch this data, put it in state, and render it. Let's get going.

Add Survey Results to the Component Socket

We could add the survey result data to the parent live view's mount/1 callback, but there's a better way. The component responsible for the given portion of the dashboard should hold, render, and manage the state for that portion of the dashboard. This keeps our code clean and organized.

The component's update/2 callback will fire each time AdminDashboardLive renders our component, so this is where we will add survey results data to component state. Since we're going to have to add survey results each time someone interacts with our view, we'll build a reusable reducer that does the work for us. Add the following update/2 function to survey_results_live.ex:

```
defmodule PentoWeb.SurveyResultsLive do
  use PentoWeb, :live_component
  alias Pento.Catalog

  def update(assigns, socket) do
    {:ok,
     socket
     |> assign(assigns)
     |> assign_products_with_average_ratings()}
  end
end
```

Our little pipeline calls two reducers, assign/2 and assign_products_with_average_ratings/2. Remember, reducers transform accumulators, and in a live view, the accumulator is the socket. That means assign/2 is a socket, and we use it to add all of the assigns keys and values that came from the live_component/3 function.

Use Custom Reducers to Initialize State

Next, we need to write that second reducer to drop our survey results into the socket. Implement this assign_products_with_average_ratings/1 function:

```
defp assign_products_with_average_ratings(socket) do
  socket
  |> assign(
    :products_with_average_ratings,
    Catalog.products_with_average_ratings())
end
```

The assign_products_with_average_ratings/1 reducer function is implemented to call on our Catalog.products_with_average_ratings/0 function and add the query results to socket assigns under the :products_with_average_ratings key.

Notice how we could have dropped this code right into `update/2`, and it would have worked. Keep an eye out for the code that will eventually support user interactions. We can re-use this reducer function later when we build the code flow that fires when a user filters the survey data by demographic. Take this small piece of advice: use reducers over raw socket interactions in live views to maintain both your code organization and your sanity!

Your Turn: Render Intermediate Results

If you'd like, add a tiny bit of code to your template to render a list of products. Take the data from the `:products_with_average_ratings` assign. Once you've verified that your code works, come back and we'll render it as a bar chart.

Render SVG Charts with Context

Most of the time, web developers reach for JavaScript to build beautiful graphs and charts. Because our server always has an up-to-date view of the data and a convenient way to send down changes, we don't have to settle for a cumbersome workflow that splits our focus across the client-server boundary. We can render *graphics* the same way we render *html*, with server-side rendering. That means we need a dependency that can draw our charts on the server and send that chart HTML down to the client.

We'll use the Context charting library¹ to handle our server-side SVG chart rendering. Using Context, we'll build out charts in two steps. We'll initialize the chart's dataset first, and then render the SVG chart with that dataset. We'll continue building out the elegant reducer pipeline that our component uses to establish state—adding new functions in the pipeline for each step in our chart building and rendering process. You'll see how the reducer pattern can help us build out and maintain even complex state in an organized way.

First, let's initialize the data.

Initialize the Dataset

As with many Elixir libraries, Context works well with CRC. The accumulator is a struct called a dataset. Context provides us with the `DataSet`² module to produce structs describing the state of the chart, with reducer functions to manipulate that data, and with converter functions to convert the data to different kinds of charts. That structure should sound familiar.

1. <https://github.com/mindok/context>

2. <https://hexdocs.pm/context/Context.Dataset.html>

You can specify your chart data as a list of maps, list of lists, or a list of tuples. Recall that we ensured that our query for products with average ratings returns a list of tuples, and now you know why.

We'll begin by adding a new reducer function to the pipeline in `update/2` to add a Dataset to our `socket.assigns`. We'll build the Dataset with the survey results already in our `socket.assigns`.

Define a reducer, `assign_dataset/1`, that adds a new dataset to socket assigns in `survey_results_live.ex`:

```
defmodule PentoWeb.SurveyResultsLive do
  use PentoWeb, :live_component
  alias Pento.Catalog

  # ...

  def assign_dataset(
    %{assigns: %{
      products_with_average_ratings: products_with_average_ratings
    }} = socket) do
    socket
    |> assign(
      :dataset,
      make_bar_chart_dataset(products_with_average_ratings)
    )
  end

  defp make_bar_chart_dataset(data) do
    Context.Dataset.new(data)
  end
end
```

Then, invoke it in the reducer pipeline that we're building out in the `update/2` function:

```
defmodule PentoWeb.SurveyResultsLive do
  use PentoWeb, :live_component
  alias Pento.Catalog

  def update(assigns, socket) do
    {:ok,
     socket
     |> assign(assigns)
     |> assign_products_with_average_ratings()
     |> assign_dataset()}
  end

  # ...
end
```

Once again, we create simple reducers to assign data, and Elixir rewards us with the beautiful pipeline in `update/2`. We tack on another reducer,

assign_dataset/2 that picks off the ratings and uses them to make a new dataset that we add to the socket.

If you were to inspect the return of the call to Context.Dataset.new/1, you'd see the following struct:

```
%Context.Dataset{
  data: [
    {"Tic-Tac-Toe", 3.4285714285714284},
    {"Table Tennis", 2.5714285714285716},
    {"Chess", 2.625}
  ],
  headers: nil,
  title: nil
}
```

The first element in a Dataset is :data, pointing to the data we'd like to render in the chart.

Initialize the BarChart

The next step is to initialize the bar chart. This intermediate form will give Context the descriptive metadata it needs to render the chart. We'll wrap up the code to initialize a bar chart and add it to socket state with a nice reducer function. Define a function, assign_chart/1 as shown here:

Now we can make a reducer to initialize a BarChart with the DataSet in survey_results_live.ex:

```
defmodule PentoWeb.SurveyResultsLive do
  use PentoWeb, :live_component
  alias Pento.Catalog

  # ...

  defp assign_chart(%{assigns: %{dataset: dataset}} = socket) do
    socket
    |> assign(:chart, make_bar_chart(dataset))
  end

  defp make_bar_chart(dataset) do
    Context.BarChart.new(dataset)
  end
end
```

Then, call it from the reducer pipeline we're building our update/2 function:

```
defmodule PentoWeb.SurveyResultsLive do
  use PentoWeb, :live_component
  alias Pento.Catalog

  def update(assigns, socket) do
```



```

{:ok,
 socket
  |> assign(assigns)
  |> assign_products_with_average_ratings()
  |> assign_dataset()
  |> assign_chart()}
end

# ...

```

The call to `BarChart.new/1` creates a `BarChart` struct that describes how to plot the bar chart. The `BarChart` module provides a number of configurable options with defaults.³ You can use these options to set the orientation, the colors, the padding, and more.

The `BarChart.new/1` constructor will produce a map. The `column_map` key will have a mapping for each bar, as you can see here:

```
column_map: %{category_col: 0, value_cols: [1]}
```

The `column_map` tells the bar chart how to chart the data from the dataset. The first key, the `category_col`, has an index of 0 and serves as the *label* of our bar chart. This means it will use the element at the 0 index of each tuple in the dataset to inform the bar chart's column name. The chart has only one column in the list of `value_cols`, our product rating average at index 1 of the dataset tuples. A `value_col` specifies the *height* of a bar.

Believe it or not, now `Contex` has all it needs to render an SVG chart. Let's do it.

Transform the Chart to SVG

The final step of showing our survey data is to render SVG markup on the server. We'll do this step with the `Contex.Plot` module. You'll notice that the `Plot` module is a converter that takes the intermediate accumulator and converts it to an SVG chart, the same way our `render/1` function translates a live view to HTML.

We'll tack on a reducer added to our `update/2` pipeline to build the SVG that we'll later access as we render the chart in `survey_results_live.ex`, like this:

```

defmodule PentoWeb.SurveyResultsLive do
  use PentoWeb, :live_component
  alias Pento.Catalog
  alias Contex.Plot

  def update(assigns, socket) do

```

3. <https://hexdocs.pm/contex/Contex.BarChart.html#summary>

```

{:ok,
 socket
 |> assign(assigns)
 |> assign_products_with_average_ratings()
 |> assign_dataset()
 |> assign_chart()
 |> assign_chart_svg()}}
end

...

def assign_chart_svg(%{assigns: %{chart: chart}} = socket) do
  socket
  |> assign(:chart_svg, render_bar_chart(chart))
end

defp render_bar_chart(chart) do
  Plot.new(500, 400, chart)
end

```

There are no surprises here. We merely tack another reducer onto the chain. This one renders the bar chart, and assigns the result to the socket. We'll customize our plot with some titles and labels for the x- and y-axis. Add to the `render_bar_chart/1` function, like this:

```

# lib/pento_web/live/survey_results_live.ex

defp render_bar_chart(chart) do
  Plot.new(500, 400, chart)
  |> Plot.titles(title(), subtitle())
  |> Plot.axis_labels(x_axis(), y_axis())
end

defp title do
  "Product Ratings"
end

defp subtitle do
  "average star ratings per product"
end

defp x_axis do
  "products"
end

defp y_axis do
  "stars"
end

```

We create tiny single-purpose functions to do the work of building out the rest of the graph. This code will (you guessed it), apply the title, subtitles, and axis labels to our chart. Now we're ready to transform our plot into an

SVG with the help of the Plot module's `to_svg/1` function. Then, we'll add that SVG markup to socket assigns:

```
# lib/pento_web/live/survey_results_live.ex
def render_bar_chart(chart) do
  Plot.new(500, 400, chart)
  |> Plot.titles(title(), subtitle())
  |> Plot.axis_labels(x_axis(), y_axis())
  |> Plot.to_svg()
end
```

The code in `render_bar_chart/1` is a converter, and the implementation is yet another beautiful microcosm of the CRC pattern. We take a new plot, and call a couple of intermediate reducers to tack on the title and subtitles. Then, we pipe the result to the `Plot.to_svg/1` converter.

We're finally ready to render this chart SVG in our template.

Render the Chart in the Template

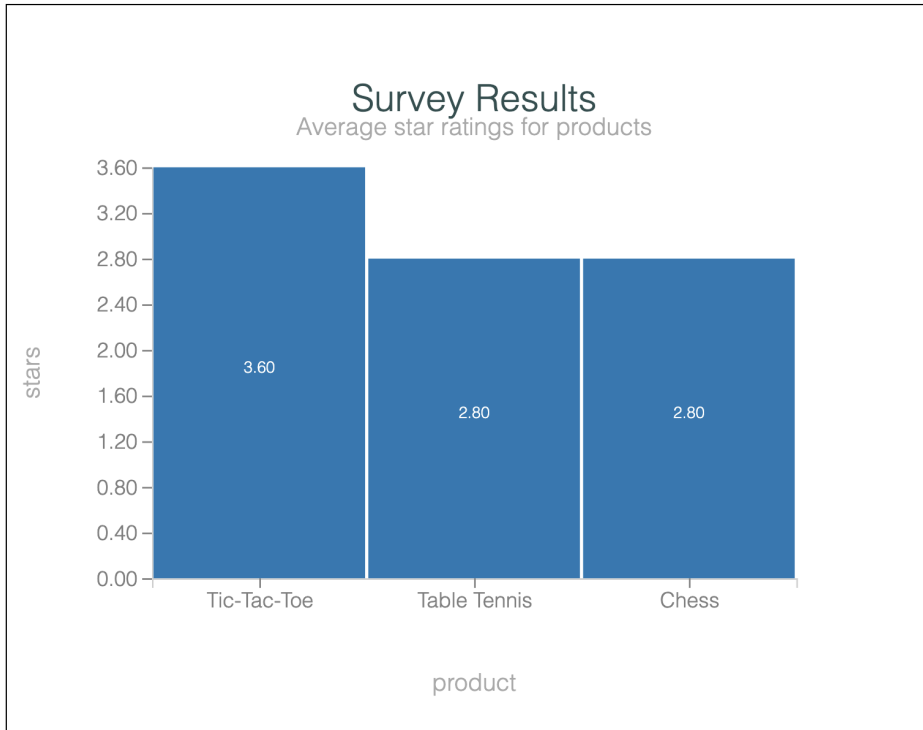
Now, we've implemented the `update/2` constructor to establish the data in the socket. The next step is to add a bit of code to our template. Conceptually, the converter is a function that takes the data in the socket and prepares it for use in the browser. At least, that's the theory. Let's see what will happen in practice.

Our `SurveyRatingsLive` template is still pretty simple. It merely needs to call the functions we've already built:

```
interactive_dashboard/pento/lib/pento_web/live/survey_results_live.html.leex
<div id="survey-results-chart">
  <%= @chart_svg %>
</div>
```

That's pretty thin, exactly as we like it. The template delegates the heavy Elixir to the helpers we've written. Our template renders the SVG stored in the `@chart_svg` assignment, and wraps that much in a div.

Now is the moment we've waited for. Navigate to `/admin-dashbaord` to see the results of all of our hard work:



It works! Thanks to the beauty of CRC and reducer pipelines, we were able to manage the non-trivial work of building and rendering our SVG chart in an easy-to-read and easy-to-maintain way.

Our chart is beautiful, and it's rendered on the server. The next step is to make it responsive. Let's get to work on the demographic filters.

Add Filters to Make Charts Interactive

So far, we have a beautiful server-side rendered dashboard, but we haven't done anything yet that is LiveView specific. In this section, we change that. We'll give our users the ability to filter the survey results chart by demographic, and you'll see how we can re-use the reducers we wrote earlier to support this functionality.

In this section, we'll walk-through building out a "filter by age group" feature, and leave it up to you to review the code for the "filter by gender" feature.

Filter By Age Group

It's time to make the component smarter. When it's done, it will let users filter the survey results chart by demographic data. Along the way, you'll get another chance to implement event handlers on a stateful component. All we need to do is build a form for various age groups, and then capture a LiveView event to refresh the survey data with a query.

We'll support age filters for “all”, “under 18”, “18 to 25”, “25 to 35”, and “over 35”. Here's what it will look like when we're done:



It's a pretty simple form with a single control. We'll capture the form change event to update a query, and the survey will default to the unfiltered “all” when the page loads. Let's get started.

Build the Age Group Query Filters

We'll begin by building a set of query functions that will allow us to trim our survey results to match the associated age demographic. We'll need to surface an API in the boundary code and add a query to satisfy the age requirement in the core. The result will be consistent, testable, and maintainable code.

Let's add a few functions to the core in `product/query.ex`. First, make sure you alias `Pento.Accounts.User` and `Pento.Survey.Demographic` at the top of the `Catalog.Product.Query` module. Then, add these functions:

```
interactive_dashboard/pento/lib/pento/catalog/product/query.ex
def join_users(query \ base()) do
  query
  |> join(:left, [p, r], u in User, on: r.user_id == u.id)
end

def join_demographics(query \ base()) do
  query
  |> join(:left, [p, r, u, d], d in Demographic, on: d.user_id == u.id)
end

def filter_by_age_group(query \ base(), filter) do
  query
  |> apply_age_group_filter(filter)
end
```

First off, two of the reducers implement join statements. The syntax is a little confusing, but don't worry. The lists of variables represent the tables in the resulting join. In Ecto, it's customary to use a single letter to refer to associated tables. Our tables are `p` for product, `r` for results of surveys, `u` for users, and `d` for demographics. So the statement `join(:left, [p, r, u, d], d in Demographic, on: d.user_id == u.id)` means we're doing:

- a `:left` join
- that returns [products, results, users, and demographics]
- where the `id` on the user is the same as the `user_id` on the demographic

We also have a reducer to filter by age group. That function relies on the `apply_age_group_filter/2` helper function that matches on the age group. Let's take a look at that function now.

```
interactive_dashboard/pento/lib/pento/catalog/product/query.ex
defp apply_age_group_filter(query, "18 and under") do
  birth_year = DateTime.utc_now().year - 18

  query
  |> where([p, r, u, d], d.year_of_birth >= ^birth_year)
end
```

```

defp apply_age_group_filter(query, "18 to 25") do
  birth_year_max = DateTime.utc_now().year - 18
  birth_year_min = DateTime.utc_now().year - 25

  query
  |> where(
    [p, r, u, d],
    d.year_of_birth >= ^birth_year_min and d.year_of_birth <= ^birth_year_max
  )
end

defp apply_age_group_filter(query, "25 to 35") do
  birth_year_max = DateTime.utc_now().year - 25
  birth_year_min = DateTime.utc_now().year - 35

  query
  |> where(
    [p, r, u, d],
    d.year_of_birth >= ^birth_year_min and d.year_of_birth <= ^birth_year_max
  )
end

defp apply_age_group_filter(query, "35 and up") do
  birth_year = DateTime.utc_now().year - 35

  query
  |> where([p, r, u, d], d.year_of_birth <= ^birth_year)
end

defp apply_age_group_filter(query, _filter) do
  query
end

```

Each of the demographic filters specifies an age grouping and does a quick bit of date math to date-box the demographic to the right time period. Then, it's only one more short step to interpolate those dates in an Ecto clause. Notice that the default query will handle "all" and also any other input the user might add.

We can use the public functions in our Catalog boundary to further reduce the `products_with_average_ratings` query before executing it. Let's update the signature of our `Catalog.products_with_average_ratings/0` function in `catalog.ex` to take an `age_group_filter` and apply our three reducers, like this:

```

def products_with_average_ratings(%{
  age_group_filter: age_group_filter
}) do
  Product.Query.with_average_ratings()
  |> Product.Query.join_users()
  |> Product.Query.join_demographics()
  |> Product.Query.filter_by_age_group(age_group_filter)
  |> Repo.all()
end

```

end

This code is beautiful in its simplicity. The CRC pipeline creates a base query for the constructor. Then, the reducers refine the query by joining the base to users, then to demographics, and finally filtering by age. We send the final form to the database to fetch results.

The code in the boundary simplifies things a bit by pattern matching instead of running full validations. If a malicious user attempts to force a value we don't support, this server will crash, just as we want it to. We also accept any kind of filter, but our code will default to unfiltered code if no supported filter shows up.

Now, we're ready to consume that code in the component.

Your Turn: Test Drive the Query

Before you run the query in IEx, open up `lib/pento_web/live/survey_results_live.ex` and comment out the call to the `get_products_with_average_ratings/0` function in the `assign_products_with_average_ratings/1`, like this:

```
defp assign_products_with_average_ratings(socket) do
  socket
  # |> assign(
  #   :products_with_average_ratings,
  #   Catalog.products_with_average_ratings())
end
```

We'll come back in a bit and make the necessary changes to this reducer's invocation of the `get_products_with_average_ratings` function. For now, we'll just comment it out so that the code compiles and you can play around with your new query.

Open up IEx with `iex -S mix` and run the new query to filter results by age. You will need to create a map that has the expected age filter. You should see a filtered list show up when you change between filters. Does your IEx log show the underlying SQL that's sent to the database?

Add the Age Group Filter to Component State

With a query filtered by age group in hand, it's time to weave the results into the live view. Before we can actually change data on the page, we'll need a filter in the socket when we `update/2`, a form to send the filter event, and the handlers to take advantage of it. Let's update our `SurveyResultsLive` component to:

- Set an initial age group filter in `socket assigns` to "all"

- Display a drop-down menu with age group filters in the template
- Respond to form events by calling the updated version of our `Catalog.products_with_average_ratings/1` function with the age group filter from `socket assigns`

First up, let's add a new reducer to `survey_results_live.ex`, called `assign_age_group_filter/1`:

```
defmodule PentoWeb.SurveyResultsLive do
  use PentoWeb, :live_component
  alias Pento.Catalog

  def update(assigns, socket) do
    {:ok,
     socket
     |> assign(assigns)
     |> assign_age_group_filter()
     |> assign_products_with_average_ratings()
     |> assign_dataset()
     |> assign_chart()
     |> assign_chart_svg()}}
  end

  def assign_age_group_filter(socket) do
    socket
    |> assign(:age_group_filter, "all")
  end
end
```

The reducer is getting longer, but no more complex thanks to our code layering strategy. We can read our initial `update/2` function like a storybook. The reducer adds the default age filter of “all”, and we're off to the races.

Now, we'll change `assign_products_with_average_ratings/1` function in `SurveyResultsLive` to use the new age group filter:

```
defp assign_products_with_average_ratings(
  %{assigns: %{age_group_filter: age_group_filter}} =
  socket) do
  assign(
    socket,
    :products_with_average_ratings,
    Catalog.products_with_average_ratings(
      %{age_group_filter: age_group_filter}
    )
  )
end
```

We pick up the new boundary function from `Catalog` and pass in the filter we set earlier. While you're at it, take a quick look at your page to make sure

everything is rendering correctly. We want to make sure everything is working smoothly before moving on.

Now, we need to build the form controls.

Send Age Group Filter Events

We're ready to add some event handlers to our component. First, we'll add the drop-down menu to the component's template and default the selected value to the `@age_group_filter` assignment to `survey_results_live.html.leex`, using the code below:

```
interactive_dashboard/pento/lib/pento_web/live/survey_results_live.html.leex
<form phx-change="age_group_filter" phx-target="@myself">
  <label>Filter by age group:</label>
  <select name="age_group_filter" id="age_group_filter">
    <%= for age_group <-
      ["all", "18 and under", "18 to 25", "25 to 35", "35 and up"] do %>
      <option
        value="@age_group" %>
        <%=if @age_group_filter == age_group, do: "selected" %> >
          <%=age_group%>
        </option>
      <% end %>
    </select>
  </form>
```

LiveView works best when we surround individual form helpers with a full form. We render a drop-down menu in a form. The component is stateful, so the form tag must have the `phx-target` attribute set to `@myself` for the component to receive events. The form also has the `phx-change` event.

To respond to this event, add a handler matching `"age_group_filter"` to `survey_results_live.ex`, like this:

```
interactive_dashboard/pento/lib/pento_web/live/survey_results_live.ex
def handle_event(
  "age_group_filter",
  %{"age_group_filter" => age_group_filter},
  socket
) do
  {:noreply,
   socket
   |> assign_age_group_filter(age_group_filter)
   |> assign_products_with_average_ratings()
   |> assign_dataset()
   |> assign_chart()
   |> assign_chart_svg()}
end
```

Now you can see the results of our hard work. Our event handler responds by updating the age group filter in socket assigns and then re-invoking the rest of our reducer pipeline. The reducer pipeline will operate on the new age group filter to fetch an updated list of products with average ratings. Then, the template is re-rendered with this new state. Let's break this down step by step.

First, we update socket assigns `:age_group_filter` with the new age group filter from the event. We do this by implementing a new version of our `assign_age_group_filter/2` function.

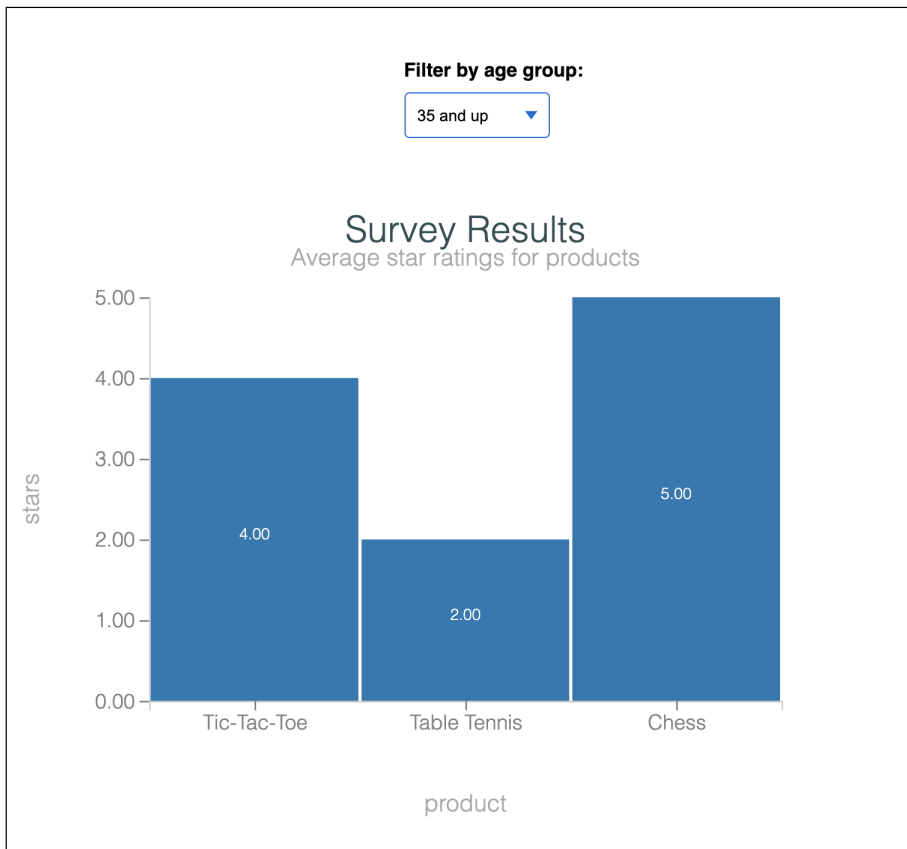
```
interactive_dashboard/pento/lib/pento_web/live/survey_results_live.ex
def assign_age_group_filter(socket, age_group_filter) do
  assign(socket, :age_group_filter, age_group_filter)
end
```

Then, we update socket assigns `:products_with_average_ratings`, setting it to a re-fetched set of products. We do this by once again invoking our `assign_products_with_average_ratings` reducer, this time it will operate on the updated `:age_group_filter` from socket assigns.

Lastly, we update socket assigns `:dataset` with a new Dataset constructed with our updated products with average ratings data. Subsequently, `:chart`, and `:chart_svg` are also updated in socket assigns using the new dataset. All together, this will cause the component to re-render the chart SVG with the updated data from socket assigns.

Now, if we visit `/admin-dashboard` and select an age group filter from the drop down menu, we should see the chart render again with appropriately filtered data:

Survey Results



Phew! That's a *lot* of powerful capability packed into just a few lines of code. Just as we promised, our neat reducer functions proved to be highly reusable. By breaking out individual reducer functions to handle specific pieces of state, we've ensured that we can construct and re-construct pipelines to manage even complex live view state.

This code should account for an important edge case before we move on. There might not be any survey results. Let's select a demographic with no associated product ratings. If we do this, we'll see the LiveView crash with the following error in the server logs:

```
[error] GenServer #PID<0.3270.0> terminating
** (FunctionClauseError) ...
(elixir 1.10.3) lib/map_set.ex:119: MapSet.new_from_list(nil, [nil: []])
(elixir 1.10.3) lib/map_set.ex:95: MapSet.new/1
(context 0.3.0) lib/chart/mapping.ex:180: Context.Mapping.missing_columns/2
```

```
...
(context 0.3.0) lib/chart/mapping.ex:139: Context.Mapping.validate_mappings/3
(context 0.3.0) lib/chart/mapping.ex:57: Context.Mapping.new/3
(context 0.3.0) lib/chart/barchart.ex:73: Context.BarChart.new/2
```

As you can see, we *can't* initialize a Contex bar chart with an empty dataset. There are a few ways we could solve this problem. Let's solve it like this. If we get an empty results set back from our `Catalog.products_with_average_ratings/1` query, then we should query for and return a list of product tuples where the first element is the product name and the second element is 0. This will allow us to render our chart with a list of products displayed on the x-axis and no values populated on the y-axis.

Assuming we have the following query:

```
interactive_dashboard/pento/lib/pento/catalog/product/query.ex
def with_zero_ratings(query \\ base()) do
  query
  |> select([p], {p.name, 0})
end
```

And context function:

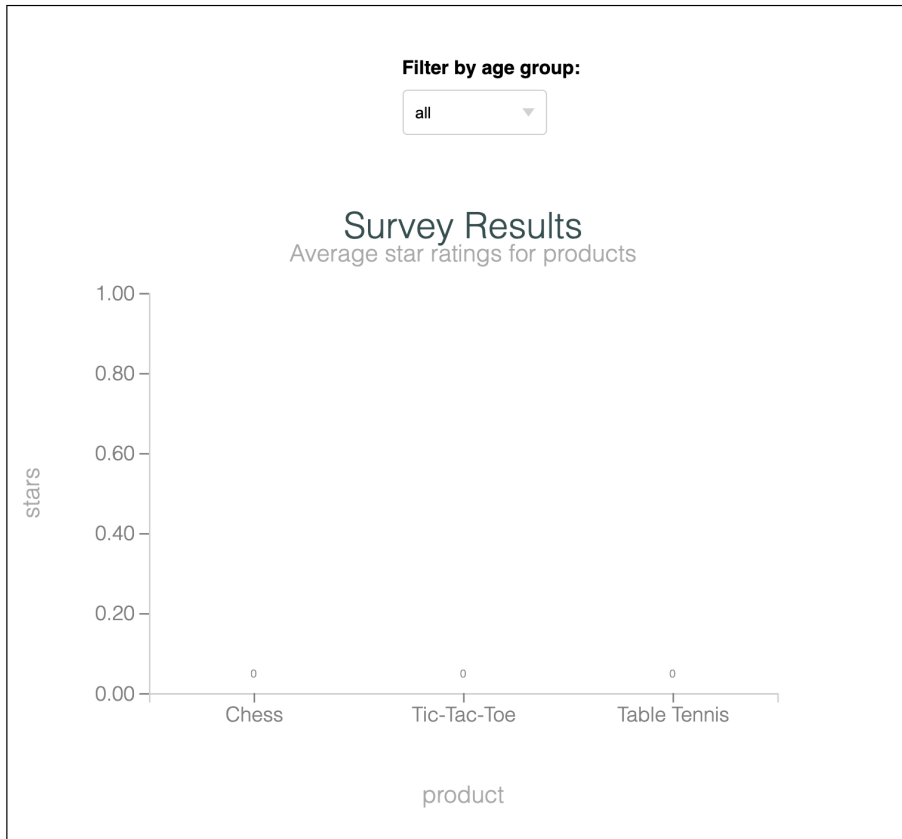
```
interactive_dashboard/pento/lib/pento/catalog.ex
def products_with_zero_ratings do
  Product.Query.with_zero_ratings()
  |> Repo.all()
end
```

We can update our LiveView to implement the necessary logic:

```
defp assign_products_with_average_ratings(
  %{assigns: %{age_group_filter: age_group_filter}} =
  socket
) do
  assign(
    socket,
    :products_with_average_ratings,
    get_products_with_average_ratings(%{age_group_filter: age_group_filter})
  )
end

defp get_products_with_average_ratings(filter) do
  case Catalog.products_with_average_ratings(filter) do
    [] ->
      Catalog.products_with_zero_ratings()
    products ->
      products
  end
end
```

Now, if we select an age group filter for which there are no results, we should see a nicely formatted empty chart:



Nice! With a few extra lines of code, we get exactly what we're looking for. We have a beautifully interactive dashboard for just a few lines of code beyond the static version. All that remains is to make this code more beautiful.

Refactor Chart Code with Macros

Our `SurveyResultsLive` component has a fair bit of charting support, in addition to the typical `LiveView` functions that set and change the socket. This kind of charting logic and configuration should live elsewhere so other components can take advantage of it as well.

Let's refactor the chart code by extracting common code into a `__using__` macro. In return for these efforts, your live view logic will be clean and re-usable. Here's how it works.

Refactor with `__using__`

At the top of every LiveView we've written so far, you see the call to use `PentoWeb, :live_view`. The `use` directive calls the `__using__` macro on the `PentoWeb` module. That code in turn returns *code* that is injected into our live view modules. Open up `lib/pento_web.ex` and take a look:

```
def live_view do
  quote do
    use Phoenix.LiveView,
      layout: {PentoWeb.LayoutView, "live.html"}

    unquote(view_helpers())
  end
end
...
defmacro __using__(which) when is_atom(which) do
  apply(__MODULE__, which, [])
end
```

At the bottom of the file, you'll see a `__using__` macro. Think of macros as Elixir code that writes and injects code. When a LiveView module calls `use PentoWeb, :liveview`, Elixir calls this `__using__` function with a `which` value of `:live_view`. Then, Phoenix calls the `live_view` function, and returns the code listed there. The `quote` macro surrounds code that should be injected, so that code will add a `use Phoenix.LiveView` with a few options. The `unquote(view_helpers())` code injects still more code, and so on.

If all of this seems a bit complicated to you, don't worry. You just need to understand that calling `use` with some module will make all of the functions of that module available in whichever module you are calling `use`.

We're going to do something similar. Future developers who want to use our charting functionality will call `use PentoWeb.BarChart` to inject all of the charting configuration code our module needs. Let's do that next.

Extract Common Helpers

First, we'll define a module `PentoWeb.BarChart` that wraps up our chart rendering logic:

```
interactive_dashboard/pento/lib/pento_web/bar_chart.ex
defmodule PentoWeb.BarChart do
  alias Context.{Dataset, BarChart, Plot}

  def make_bar_chart_dataset(data) do
    Dataset.new(data)
  end

  def make_bar_chart(dataset) do
```

```

dataset
|> BarChart.new()
end

def render_bar_chart(chart, title, subtitle, x_axis, y_axis) do
  Plot.new(500, 400, chart)
  |> Plot.titles(title, subtitle)
  |> Plot.axis_labels(x_axis, y_axis)
  |> Plot.to_svg()
end
end

```

We move the chart-specific functions from our LiveView to a common module. You can recognize the code that builds our dataset and bar chart, and the converter that renders them. We don't make any changes at this point.

Import the Charting Module

Next up, we need code that imports the common functions. Let's think about where we want the imported code to live. PentoWeb doesn't need access to the chart helpers. Our live view does. That means we need to *inject code* that imports PentoWeb.BarChart. Luckily, we have a quote function that does exactly that.

Open up the PentoWeb module in `lib/pento_web.ex` and add in a function called `chart_helpers/0` that injects our import function:

```

defp chart_helpers do
  quote do
    import PentoWeb.BarChart
  end
end

```

Perfect. The quote macro will tell Elixir to inject the BarChart functions. With the implementation of the `chart_helpers` function, our application has a place to pull in common functions, aliases, and configuration related to charting.

Now, we can call that code in the traditional way, with a use directive.

Inject the Code with `__using__`

The last job is to implement the public function that the PentoWeb's `__using__` macro definition will apply via the call to `use PentoWeb, :chart_live`, like this:

```

interactive_dashboard/pento/lib/pento_web.ex
def chart_live do
  quote do
    unquote(chart_helpers())
  end
end

```


Perfect. Now, the `chart_live` function will work perfectly with the `__using__` code, just like the `use PentoWeb, :live_view` expression you see at the top of each of each Phoenix live view. All that remains is to, um, use the macro.

Use the Macro

Go ahead and to delete the refactored functions from your live view. Then, add the new `use` directive to `SurveyResultsLive` component, as shown here:

```
interactive_dashboard/pento/lib/pento_web/live/survey_results_live.ex
```

```
defmodule PentoWeb.SurveyResultsLive do
  use PentoWeb, :live_component
  use PentoWeb, :chart_live
```

Remove `make_bar_chart_dataset/1`, `make_bar_chart/2` and `render_bar_chart/5` from the live view.

```
interactive_dashboard/pento/lib/pento_web/live/survey_results_live.ex
```

```
def assign_chart_svg(%{assigns: %{chart: chart}} = socket) do
  socket
  |> assign(
    :chart_svg,
    render_bar_chart(chart, title(), subtitle(), x_axis(), y_axis())
  )
end

defp title do
  "Survey Results"
end

defp subtitle do
  "Average star ratings for products"
end

defp x_axis do
  "product"
end

defp y_axis do
  "stars"
end
```

The result is pleasing. This kind of layering shields our users from dealing with charting complexity when they are working with the data that makes those charts work. Now, all of the code that renders a bar chart lives in `PentoWeb.BarChart`, while the code specific to how to render the bar chart *for the survey results component* remains in `SurveyResultsLive`. We could easily imagine our bar chart logic and configuration growing more complex—say, to accommodate custom color configuration, padding, orientation and more. Now,

should we want to accommodate that increased complexity, it has a logical home in the chart module.

With this new module and macro in place, you have yet another LiveView code organization tool in your kit. You can use macros to organize re-usable code that keeps your live views clean and concise.

This chapter has been pretty long, so it's time to wrap up.

Your Turn

We built a lot of new functionality in this chapter. Let's review.

You built a brand-new admin dashboard that displays survey results data with the help of the *Contex* library. *Contex* let's you render SVG charts on the server, which makes it the perfect fit for creating beautiful charts in LiveView. You took it a step further by making your survey results chart interactive. Gender and age group filters allowed your user to filter survey results by demographic info, and you once again used LiveView event handlers to manage these interactions. Finally, you did a bit of refactoring to keep your live view clean and concise with the use of macros.

By now, you've built a number of component-backed features, and you're starting to get the hang of using the reducer pattern and the core/boundary designations to quickly and easily decide where new code belongs. You've seen how these patterns allow you to move fast and write clean, organized code. Once again, we're left with a highly interactive feature that manages complex single-page app state with very little code. On top of that, you're now prepared to use server-side-rendered SVG to visualize data in LiveView.

Before we move on to the next chapter, it's your turn to get your hands dirty.

Give It A Try

The “filter by gender” code is present in the codebase. Choose the option that best reflects your confidence level.

If you're looking for an *easy* exercise, review the code to filter by gender that's already in the codebase. Take some time to walk through the code, starting in the query builder and context functions in the core and boundary, and making your way up to the LiveView.

If you're looking for an *intermediate* exercise, use the same pattern that we used to build the age filter to add a gender filter to your own code.

Next Time

Now we have a working dashboard, but it does not react in real-time to data that comes in from other parts of the system. In the next chapter, we'll use the Phoenix publish-subscribe interface to update our dashboard when new survey results come in. Then, we'll add a new component to the dashboard that reports on real-time user interactions with our products. Let's keep going!

Build a Distributed Dashboard

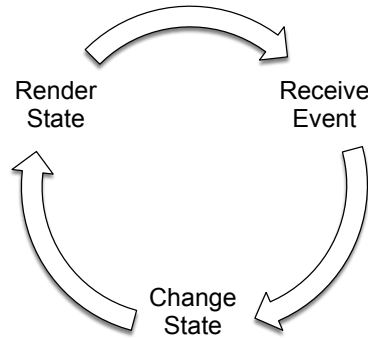
In the last chapter, you explored some of the capabilities that make Phoenix LiveView the perfect fit for single-page apps like dashboards. Components help organize pages into layers, and the LiveView workflow makes quick work of adding interactive controls.

So far, the live views you've built have focused on single users interacting with browsers. Way back in the first chapter of this book, you learned that live views are effectively distributed systems. By now, you should have a better sense of why that's true. JavaScript code on browser clients triggers events that transmit data to the server, and the servers respond, often after connecting to other services like databases. We're shielded from those details because LiveView has built the complicated parts for us.

In this chapter, you'll see that live views are not only distributed systems because of the way they manage state across the client and the server, but also because they are capable of reflecting the distributed state of your entire application. With the features you're about to build, you'll push LiveView and Phoenix by connecting views to other pages and database activity *not* triggered by the current user. Our application will be much more interactive, but we won't have to spend extraordinary effort to make it so. Rather than building the hard parts yourself, you'll rely on various Phoenix messaging frameworks. Let's talk about them now.

LiveView and Phoenix Messaging Tools

We're more than a half-way into this book, and you may be coming to appreciate the LiveView programming model. Let's revisit the LiveView flow figure that was first shown in [Chapter 1, Get To Know LiveView, on page 1](#):



Just like this figure shows, you’ve expressed each view with a data model that you stored in the socket. Your code changed the data with reducers tied to event handlers, and you built a template or a render function to convert the data in the socket to HTML pages and SVG graphics. The architecture neatly cleaves the concepts of *changing* data and *rendering* data.

This flow paves the way for success as we integrate other distributed elements into our Phoenix application. If you stop and think about it, it doesn’t really matter whether the events your live view handles are initiated by a user’s mouse click on a browser page or a boundary function that sends a message from some other area of your application. You’ll use a variety of tools to send events. They will flow through the views just as if they’d been sent by a user.

In this chapter, we’re going to use several Phoenix messaging libraries to trigger other kinds of events, and we’ll teach our live view to handle these events. In this way, you can build live views that are capable of reflecting the distributed state of your entire Phoenix application.

Using Phoenix.PubSub,¹ your boundary functions can *publish events* to send messages to every other process that expresses interest, including live views. Meanwhile, Phoenix.Presence² can notify you when users interact with your site.

We’re going to tie our single-page app to other services using the Phoenix.PubSub service, effectively making our dashboard reflect real world updates, regardless of their source. The impact will be striking. Users will see update in real-time, with excellent responsiveness. We’ll also take advantage of Phoenix Presence and integrate it into our live view for some real-time tracking of user activity around our app. Along the way, we’ll introduce some

1. https://hexdocs.pm/phoenix_pubsub/Phoenix.PubSub.html

2. <https://hexdocs.pm/phoenix/Phoenix.Presence.html>

new LiveView component capabilities and see how a parent live view can communicate updates to its child components.

Before we dive in, let's plan our attack.

As you recall, we've been working on a dashboard that charts survey results and allows users to interact with that chart by selecting demographics. We're going to extend this dashboard with a few new requirements.

You might have noticed that the dashboard doesn't automatically update when new results come in. The user must reload to see any newly submitted survey results. We'll fix that with the help of Phoenix PubSub. We also want to track user engagement by displaying a real-time list of users who are viewing products. We'll do so with the help of Phoenix Presence.

We'll begin by synchronizing AdminDashboardLive when new survey results data comes in. We'll use PubSub to send messages and we'll teach our live view to subscribe to those messages and handle them by updating the survey results chart component.

Then, we'll move on to the real-time user tracking feature. We'll build a new component that leverages Presence to display a live-updating list of which users are viewing which products at a given moment in time. Similar to how we'll build our PubSub-backed feature, we'll use Presence to send messages when a user is looking at a product, and we'll teach our live view to subscribe to those messages and handle them by updating the new user list component.

Let's get started.

Track Real-Time Survey Results with PubSub

First on the agenda is automatically updating the survey results chart component when a user completes a survey. Right now, users are entering demographics and survey results through the RatingLive.FormComponent. When we handle the event for a new survey rating in the parent SurveyLive live view, we need to notify AdminDashboardLive. The question is how.

You could try to do so with a direct message, but you'd need access to the AdminDashboardLive PID. Even if we had access, this view could crash and the PID would change. We could give names to the AdminDashboardLive process, but that would require more work and more synchronization. Fortunately, there's a better way.

Phoenix PubSub Implements the Publish/Subscribe Pattern

We're going to use Phoenix PubSub, a *publish/subscribe* implementation, to build the feature. Under the hood, a live view is just a process. Publish/subscribe is a common pattern for sending messages between processes in which messages are broadcast over a topic to dedicated subscribers listening to that topic. Let's see how it works.

Rather than sending a message directly from a sender to a receiver with `send/2`, you'll use a Phoenix PubSub server as an intermediary. Processes that need access to a topic announce their interest with a `subscribe/1` function. Then, sending processes broadcast a message through the PubSub service, over a given topic, which forwards the message to all subscribed processes.

This service is exactly what we need in order to pass messages between live views. Going through an intermediary is perfect for this use case. Neither `SurveyLive` nor `AdminDashboardLive` need to know about one another. They need only know about a common pub/sub topic. That's good news. All we need to do is use the `PubSub.broadcast/3` function to send a message over a particular topic and the `PubSub.subscribe/1` function to receive a message over a particular topic.

Plan the Feature

Our `AdminDashboardLive` process will use Phoenix PubSub to subscribe to a topic. This means that `AdminDashboardLive` will receive messages broadcast over that topic from *anywhere else* in our application. For our new feature, we'll broadcast "new survey results" messages from the `SurveyLive` live view. Then, we'll teach `AdminDashboardLive` how to handle these messages by updating the `SurveyResultsLive` component with the new survey results info.

By combining `LiveView`'s real-time functionality with PubSub's ability to pass messages across a distributed set of clients, we can seamlessly keep our live views up-to-date.

With that plan, we're ready to write some code. We'll start with a brief look at how PubSub is configured in your Phoenix application. Then, we'll set up our message broadcast and subscribe workflow. Finally, we'll teach the `AdminDashboardLive` how to update its `SurveyResultsLive` child component.

Configure Phoenix PubSub

It turns out that we don't need to do anything special to configure Phoenix PubSub. When we generated the initial application, the Phoenix application generator configured PubSub for us automatically:

```
distributed_dashboard/pento/config/config.exs
```

```
config :pento, PentoWeb.Endpoint,
  url: [host: "localhost"],
  secret_key_base: "aag4li5BIDcK1naKx/4wNENV/H34Npsh5FRN32/Ir1mURh12d0cGBsY3VLZKHKChaag4li5BIDcK",
  render_errors: [
    view: PentoWeb.ErrorView,
    accepts: ~w(html json),
    layout: false
  ],
  pubsub_server: Pento.PubSub,
  live_view: [signing_salt: "cqGw5PQX"]
```

Remember, the endpoint is the very first function a web request encounters. Here, our app's endpoint configures a PubSub server and names it `Pento.PubSub`. This server is just a registered process, and in Elixir, registered processes have names. The configuration sets the default adapter, `PubSub.PG2`. This adapter runs on Distributed Erlang—clients across distributed nodes of our app can subscribe to a shared topic and broadcast to that shared topic, because PubSub can directly exchange notifications between servers when configured to use the `Phoenix.PubSub.PG2` adapter. Building on this common robust infrastructure will save us a tremendous amount of time should we ever need this capability.

As a result of this configuration, we can access the PubSub library's `broadcast/3` and `subscribe/1` functions through `PentoWeb.Endpoint.broadcast/3` and `PentoWeb.Endpoint.subscribe/1`. We'll do exactly that as we incorporate message publishing and subscribing across the survey submission and survey results chart features.

Broadcast Survey Results

To make our results interactive, we need only make three tiny changes:

First, we'll need to broadcast a message over a topic when a user submits the survey within the `SurveyLive` view. Then, we'll subscribe the `AdminDashboardLive` view to that topic. Finally, we'll teach the `AdminDashboardLive` view to handle messages it receives over that topic by updating the `SurveyResultsLive` component.

Let's start with broadcasting our message. We'll send a `"rating_created"` message to the `"survey_results"` topic exactly when the `SurveyLive` live view receives a new rating, like this:

```
# lib/pento_web/live/survey_live.ex
alias PentoWeb.Endpoint
@survey_results_topic "survey_results"

defp handle_rating_created(
  %{assigns: %{products: products}} = socket,
```



```

        updated_product,
        product_index
    ) do

Endpoint.broadcast(@survey_results_topic, "rating_created", %{}) # I'm new!

socket
|> put_flash(:info, "Rating submitted successfully")
|> assign(
  :products,
  List.replace_at(products, product_index, updated_product)
)
end

```

We alias the endpoint to access the broadcast/3 message and add a new topic as a module attribute. Later, our dashboard will subscribe to the same topic. Most of the rest of the code is the same, except this line:

```
Endpoint.broadcast(@survey_results_topic, "rating_created", %{})
```

The endpoint's broadcast/3 sends the "rating_created" message over the @survey_results_topic topic with an empty payload. This function hands the message to an intermediary, the Pento.PubSub server, which in turn broadcasts the message with its payload to any process subscribed to the topic.

Now we're ready subscribe our dashboard to that topic.

Subscribe to Survey Results Messages

We want to use the broadcast of this message to tell the SurveyResultsLive component to update with a fresh list of filtered product ratings. So, you might want to subscribe the SurveyResultsLive component to the "survey_results" topic.

Think about it, though. When we subscribe to a topic, we do so on behalf of a process. Components don't run in their own processes—they share a process with their parent live view. In fact, components don't even implement a handle_info/2 function. That means any messages sent to the process will need to be handled by the parent live view, in this case AdminDashboardLive. That means we'll need to:

- Subscribe AdminDashboardLive to the "survey_results" topic.
- Implement a handle_info/2 function on AdminDashboardLive for the "rating_created" message.
- Use that function to tell the child SurveyResultsLive component to update.

You'll be surprised at how quickly it goes. Once again, the LiveView framework handles many of the details for us and exposes easy-to-use functions that we can leverage to build this workflow.

First, in `admin_dashboard_live.ex`, subscribe to the topic, like this:

```
defmodule PentoWeb.AdminDashboardLive do
  use PentoWeb, :live_view
  alias PentoWeb.Endpoint
  @survey_results_topic "survey_results"

  def mount(_params, _session, socket) do
    if connected?(socket) do
      Endpoint.subscribe(@survey_results_topic)
    end

    {:ok,
     socket
     |> assign(:survey_results_component_id, "survey-results")}
  end
end
```

A quick note on the usage of the `connected?/1` function. Remember, in the `LiveView` flow, `mount/3` gets called twice—once when the live view first mounts and renders as a static HTML response and again when the `WebSocket`-connected live view process starts up. We're calling `subscribe/1` *only if* the socket is connected, in the second `mount/3` call.

Now, when the `SurveyLive` live view broadcasts the `"rating_created"` message over this common topic, the `AdminDashboardLive` will receive the message. So, we'll need to implement a `handle_info/2` callback to respond to that message.

Implement a `handle_info/2` in the same file, like this:

```
distributed_dashboard/pento/lib/pento_web/live/admin_dashboard_live.ex
def handle_info(%{event: "rating_created"}, socket) do
  send_update(
    SurveyResultsLive,
    id: socket.assigns.survey_results_component_id
  ){:noreply, socket}
end
```

We want to respond to this message by updating the `SurveyResultsLive` component to display the latest data. So, we use the `send_update/3` function to send a message from the parent live view to the child component.

Update the Component

Let's take a moment to talk about the `send_update/3` function. `send_update/3` asynchronously updates the component with the specified module name and ID, where that component is running in the parent `LiveView`. Remember that in the previous chapter we stored the component ID in the parent live view's socket assigns. Here's where that pays off.

Once `send_update/3` is called, the component updates with any new assigns passed as the second argument to `send_update/3`, invoking the `preload/1` and `update/2` callback functions on that component. Our `SurveyResultsLive` component will invoke its `update/2` function, causing it to fetch the updated survey results from the database, thereby including any newly submitted product ratings.

We do have one problem, though. Recall that the reducer pipeline in our `update/2` function hard-codes the initial state of the `:gender_filter` and `:age_group_filter` to values of `"all"`. So, now, when our `update/2` function runs *again* as a result of the `AdminDashboardLive` receiving a message broadcast, we will set the `:gender_filter` and `:age_group_filter` keys in `socket assigns` to `"all"`, thereby losing whatever filter state was applied to the `SurveyResultsLive`'s socket by user interactions.

In order to fix this, the `assign_age_group_filter/1` and `assign_gender_filter/1` reducer functions need to get a littler smarter. If the socket already has a value at either of the `:age_group_filter` or `:gender_filter` keys, then it should retain that value. Otherwise, it should set the default value to `"all"`.

So, we'll implement additional function heads for these reducers that contain this logic:

```
distributed_dashboard/pento/lib/pento_web/live/survey_results_live.ex
def assign_age_group_filter(
  %{assigns: %{age_group_filter: age_group_filter}}
  = socket) do
  assign(socket, :age_group_filter, age_group_filter)
end

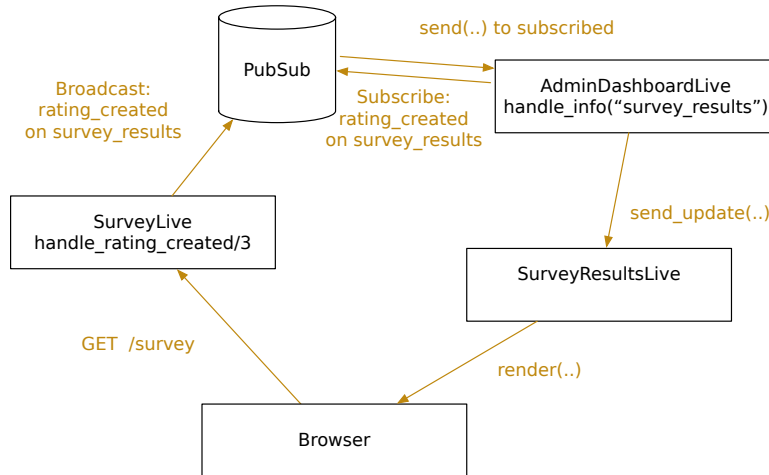
def assign_age_group_filter(socket) do
  assign(socket, :age_group_filter, "all")
end
```

That's the `age_group` filter. If the key/value pair is present in the socket, we match this first function head set the value of that key in `socket assigns` to the existing value. Otherwise, we fall through to the next matching function and set the key to `"all"`. Now, we can do the same thing to the `gender` filter:

```
distributed_dashboard/pento/lib/pento_web/live/survey_results_live.ex
def assign_gender_filter(
  %{assigns: %{gender_filter: gender_filter}}
  = socket) do
  assign(socket, :gender_filter, gender_filter)
end

def assign_gender_filter(socket) do
  assign(socket, :gender_filter, "all")
end
```

Perfect. Now, when a user submits a new product rating, a message will be broadcast over PubSub and the AdminDashboardLive view will receive that message and tell the SurveyResultsLive component to update. When that update happens, the component will reduce over the socket. Any filters in state will retain their values and the component will re-fetch products with their average ratings from the database. When the component re-renders, the users will see updated results. Putting it all together, we have something like this:



Give it a try by opening up one tab and pointing it at `/admin-dashboard`. Then, open up another browser tab or window, register as a new user, and submit a new survey. You should see the results in the “Survey Results” section of the `/admin-dashboard` page update *without* having to refresh the page.

That’s a lot of functionality all packed into, once again, just a few new lines of code. As a programmer, you get a beautiful programming model that accommodates PubSub messages the same way it handles LiveView events. Your users get connected, interactive applications that stay up-to-date when events occur anywhere in the world.

Next up, we’ll build a section into our dashboard to track user activity.

Track Real-Time User Activity with Presence

Web applications are full of rich interactions. You can think of these interactions as an active conversation between the user and your app. In Phoenix, those conversations are represented and managed by processes—often implemented with channels. By gathering up a list of active processes, we can show the active conversations happening on our site. This is exactly the

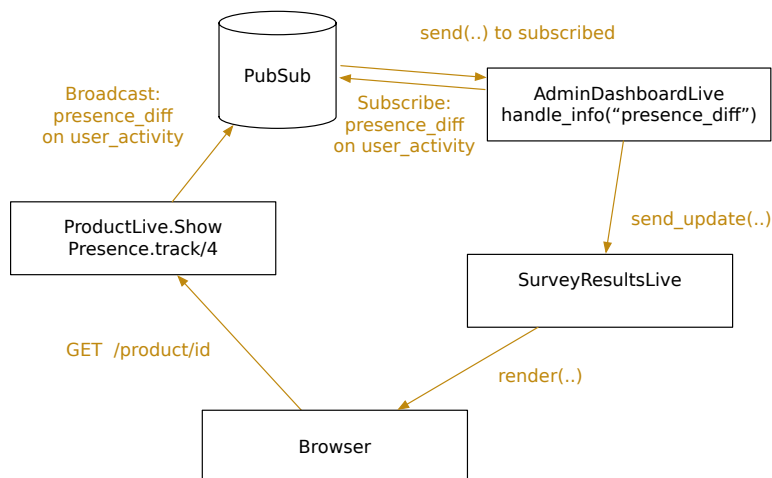
job of Phoenix Presence, a behaviour that provides the capabilities to track a user's conversation, or presence, within your application.

Tracking activity on a network is an easy problem to solve when everything is on one server. However, that is rarely the case. In the real world, servers are clustered together for performance and reliability, and connections between those servers sometimes fail. These problems make tracking presence by listing processes notoriously difficult.

Phoenix Presence solves these problems for us. It is built on top of Phoenix PubSub and leverages PubSub's distributed capabilities to reliably track processes across a distributed set of servers. It also uses a CRDT³ (Conflict-free Replicated Data Type) model to ensure that presence tracking will keep working when nodes or connections on our network fail.

We'll use Presence to give us insight as users interact with our application around the world. And because the Presence is backed by PubSub, the way we code the live views won't have to change at all.

When we're done, our dashboard will display a section that shows which users are viewing which products at a given moment. The list will update immediately as users visit and leave a Product Show live view, something like this:



This plan may seem ambitious, but it's surprisingly easy to do. To build this feature, we'll need to build the following:

3. <https://www.serverless.com/blog/crdt-explained-supercharge-serverless-at-edge>

PentoWeb.Presence

This module will define our *presence model*. It will implement the Phoenix Presence behaviour, define the data structure that will track user activity, and connect it to our app's PubSub server.

UserActivityLive component

We'll set up a component that renders a static list of users.

handle_info/3 message handler

A function on AdminDashboardLive live that tell the user activity component to update based on site user activity

We'll tie these entities together with a PubSub-backed Presence workflow. When a user visits a Product Show live view, that live view will use the `Presence.track/4` function to broadcast a user activity event over a topic. We'll subscribe AdminDashboardLive to that topic. Then, our `handle_info/3` function will take care of the rest, updating the user activity component, just like in our real-time survey results chart feature.

Set Up Presence

The Phoenix.Presence behaviour is an Elixir service based on OTP. It is used to notify applications via PubSub when processes or channels representing online presences come and go. Since a live view is just a process under the hood, we can use the Phoenix Presence API to track user activity within a live view. Then, Presence will publish details about presences that come and go.

We'll define our own module that uses this behavior. Let's take a look at that module definition now.

```
distributed_dashboard/pento/lib/pento_web/presence.ex
```

```
defmodule PentoWeb.Presence do
  use Phoenix.Presence,
    otp_app: :pento,
    pubsub_server: Pento.PubSub
```

The PentoWeb.Presence module defines our *presence model*. A presence model is the data structure that tracks information about active users on our site, and the functions that process changes in that model. So far, there's not much happening, but let's call out the details.

First, we use the Presence behaviour. As you've already seen, that behaviour calls the `__using__` macro on the Phoenix.Presence module. Notice the arguments we pass in. You might recognize Pento.PubSub as the publish/subscribe server for our application, while the `otp_app: :pento` key/value pair specifies the OTP application that holds our app's configuration.

Right now, the module is sparse. As our needs grow, we'll have functions to track new users. We just need to do one more thing to make sure our application can use this new Presence module. We have to add `PentoWeb.Presence` module to our application's children so that the Presence process starts up when our app starts up, as part of our application's supervision tree. Open up `lib/pento/application.ex` and add the module to the list of children defined in the `start` function, like this:

```
distributed_dashboard/pento/lib/pento/application.ex
def start(_type, _args) do
  children = [
    # Start the Ecto repository
    Pento.Repo,
    # Start the Telemetry supervisor
    PentoWeb.Telemetry,
    # Start the PubSub system
    {Phoenix.PubSub, name: Pento.PubSub},
    PentoWeb.Presence, # Add this line!
    # Start the Endpoint (http/https)
    PentoWeb.Endpoint
    # Start a worker by calling: Pento.Worker.start_link(arg)
    # {Pento.Worker, arg}
  ]
end
```

Let's move on to new user tracking now.

Track User Activity

To track presence, we need to answer a couple of basic questions. First, who is the user? We'll need to determine exactly which data we'll use to track the user's identity. The second is when are they present? We'll need to pick the right point in time to hook in our presence model. Let's answer the first question first. In [Chapter 2, Phoenix and Authentication, on page 31](#), the authentication service we generated placed a `user_token` in the session when a user logged in. We can use that token to fetch a `user_id`. As for *when* the user is considered to be "present", we want to track which users are viewing which products. So, the user becomes present when they are looking at the product page in `ProductLive.Show`.

Before we do anything else, we need to update the `ProductLive.Show` `LiveView` to fetch the current user using the session token. We have access to the session token in the session argument that is passed to `mount/3`. So, we'll update our `mount/3` function in `lib/pento_web/live/product_live/show.ex` to grab the token from the session and store it in socket assigns, like this:

```
distributed_dashboard/pento/lib/pento_web/live/product_live/show.ex
def mount(_params, %{"user_token" => token}, socket) do
```

```

    {:ok, assign(socket, :user_token, token)}
  end

```

Remember, the `handle_params/3` callback fires right after `mount/3`. We can use it to track the user's presence for the specified product id. Also, remember `handle_params/3` will fire twice for a new page, once when the initial page loads and once when the page's WebSocket connection is established. If the `:live_action` is `:show` and the socket is connected, we'll use the token to find the current user, like this:

```

alias PentoWeb.Presence
alias Pento.Accounts

def handle_params(%{"id" => id}, _, socket) do
  product = Catalog.get_product!(id)
  maybe_track_user(product, socket)

  {:noreply,
   socket
   |> assign(:page_title, page_title(socket.assigns.live_action))
   |> assign(:product, product)}
end

def maybe_track_user(
  product,
  %{assigns: %{live_action: :show, user_token: user_token}} = socket
) do
  if connected?(socket) do
    user = Accounts.get_user_by_session_token(user_token)
    # do tracking here!
  end
end

def maybe_track_user(product, socket), do: nil

```

In our `handle_params/3` function, we look up the product and then add a function, `maybe_track_user/2`, to conditionally track the user's presence. The word `maybe` is a convention that marks the function as conditional—we only want to do the user presence tracking if the live view is loading with the `:show` (as opposed to the `:edit`) live action, and if the live view is connected over WebSockets. Let's look inside that function now.

The `maybe_track_user/2` function picks up the user token from the socket. Then, if the live view is connected, we track the user. We use the generated authentication service code to look up the user based on the token, and leave the actual tracking for later.

Now we've prepared the live view's plumbing for tracking. We need to decide exactly what data we want to show with each user, so let's think about the user interface we ultimately want to display on our admin dashboard. We

want a list of product names, and a list of users interacting with each product. Presence allows us to store a top-level key pointing to a map of metadata. We'll use the product name as the top-level key and the metadata map will contain the list of “present” users who are viewing that product. Our Presence data structure will ultimately look like this:

```
%{
  "Chess" => %{metas: [%{
    users: [
      %{email: "robert@email.com"},
      %{email: "joe@email.com"}]}}}
}
```

The `Presence.track/4` gives us the means to store and broadcast exactly that. We call `.track/4` with:

- The PID of the process we want to track, the Product Show live view
- A PubSub message topic used to broadcast messages
- A key representing the presence, in this case the product name
- The metadata to track for each presence, in this case the list of users

Let's dive into the usage of the `track/4` function now.

To track a user for a product name, assuming that product name does not yet exist in Presence, you might do this:

```
topic = "user_activity"
Presence.track(
  some_pid,
  topic,
  product.name,
  %{users: [%{email: user.email}}}
)
```

Presence would store this data:

```
%{
  "Chess" => %{
    metas: [%{users: [%{email: "robert@email.com"}]}]
  }
}
```

Notice how the last argument we provided to `track/4` becomes part of the Presence data store's list of `:metas`—the metadata for the given presence.

But, let's say the Chess key already exists. You can't just replace the payload, or you'll erase the existing list of users. You need to update it instead. That means the tracking logic will need to accommodate existing lists of users. It will first check for the key of the given product name in the Presence store.

If it is there, the code must update the existing value, appending the new user. Otherwise, the code will simply create the initial list of users.

We'll accomplish this with the help of the Presence API's `get_by_key/2` function. This function returns the map of Presence metadata for the given topic/key pair.

```
case Presence.get_by_key("user_activity", product.name) do
  [] ->
    Presence.track(
      pid,
      "user_activity",
      product.name, %{users: [%{email: user.email}]}
    )

  %{metas: [%{users: active_users_for_product}]} ->
    Presence.update(pid, "user_activity", product.name, %{
      users: [%{email: user.email} | active_users_for_product ]
    })
end
```

We could put that code directly in the `ProductLive.Show` live view, but we already have the perfect home for this logic—the `PentoWeb.Presence` module. Let's add a function, `track_user/3`, like this:

```
distributed_dashboard/pento/lib/pento_web/presence.ex
alias PentoWeb.Presence
alias Pento.Accounts
@user_activity_topic "user_activity"

def track_user(pid, product, token) do
  user = Accounts.get_user_by_session_token(token)

  case Presence.get_by_key(@user_activity_topic, product.name) do
    [] ->
      Presence.track(
        pid,
        @user_activity_topic,
        product.name,
        %{users: [%{email: user.email}]}
      )

    %{metas: [%{users: active_users_for_product}]} ->
      Presence.update(pid, @user_activity_topic, product.name, %{
        users: [%{email: user.email} | active_users_for_product ]
      })
  end
end
```

Notice that we've also moved the call to `Accounts.get_user_session_by_token/1` into this new Presence module function.

Now, replace the comment you left in the `handle_params/3` function in `ProductLive.Show`, with this:

```
distributed_dashboard/pento/lib/pento_web/live/product_live/show.ex
```

```
def maybe_track_user(
  product,
  %{assigns: %{live_action: :show, user_token: user_token}} = socket
) do
  if connected?(socket) do
    Presence.track_user(self(), product, user_token)
  end
end

def maybe_track_user(_product, _socket), do: nil
```

Beautiful. The code calls the custom `Presence` function we built with the PID of the current live view, the product, and the user.

Now that we're tracking user presence for a given product, let's move on to the work of displaying those presences and making sure they update in real-time.

Display User Tracking

We've come a surprisingly long way with user activity tracking, but there's still a bit of work to do. We'll implement a component, `UserActivityLive`, that will use its `update/2` callback to ask `Presence` for the list of products and their present users. It will store this list in state via the socket assigns. Then, we'll render that list in our component's template.

Let's kick things off by defining our component. We know that the component needs to fetch a list of presences when it first renders. Later, we'll teach the component to update whenever a new presence is added to the `PentoWeb.Presence` data store. As you might guess, we'll have the parent live view, `AdminDashboardLive`, receive a message when this happens and respond by telling the component to update. So, we want to use the component's `update/2` function to fetch the presence list and store it in state, rather than the `mount/3` function. This way we ensure that the presence list is re-fetched when the component updates later on. More on this update flow later. Let's build our `update/2` function now.

We'll rely on the `Presence.list/1` function to fetch a list of users by topic, like this:

```
distributed_dashboard/pento/lib/pento_web/live/user_activity_live.ex
```

```
defmodule PentoWeb.UserActivityLive do
  use PentoWeb, :live_component
  alias PentoWeb.Presence
  @user_activity_topic "user_activity"
```

```

def update(_assigns, socket) do
  {:ok,
   socket
  }> assign_user_activity()
end

def assign_user_activity(socket) do
  presence_map = Presence.list(@user_activity_topic)

  user_activity =
    presence_map
    |> Enum.map(fn {product_name, _data} ->
      users =
        get_in(presence_map, [product_name, :metas])
        |> List.first()
        |> get_in([:users])

      {product_name, users}
    end)

  assign(socket, :user_activity, user_activity)
end
end

```

As usual, we extract the code to build a user activity list to a reducer function called `assign_user_activity/1`. That function's only job is to fetch a list of users from `PentoWeb.Presence`, and assign it to the `:user_activity` key. Let's look at it in detail.

We get a list of presences on the topic defined in `@user_activity_topic` using the `Presence.list/1` function. We didn't write that function. It comes as part of the `Phoenix.Presence` behaviour. Remember, the `Presence` data looks like this:

```
"Chess" => %{metas: [%{users: [%{email: "robert@email.com"}, ...]}}}
```

The keys are product names, and the values are `Presence` metadata. So, we iterate over the key/value pairs in that map, collecting the metadata. We use `get_in/2` and `List.first/1` to drill into the data structure and extract the user data we need in the form of a list of tuples where the first element is the product name and the second element is the list of present users for that product. We then assign that data to the socket, and we're ready to render.

The template iterates over this list to display the product names and their present users, like this:

```
distributed_dashboard/pento/lib/pento_web/live/user_activity_live.html.leex
<div class="user-activity-component">
  <h2>User Activity</h2>
  <p>Active users currently viewing games</p>
  <div>
    <%= for {product_name, users} <- @user_activity do %>

```

```

    <h3><%= product_name %></h3>
    <ul>
      <%= for user <- users do %>
        <li><%= user.email %> </li>
      <% end %>
    </ul>
  <% end %>
</div>
</div>

```

There are no surprises in this template. Two `for` comprehensions iterate over first the products in `@user_activity` and then their users. Then, we render the name of the product followed by a list of users, and we're done.

The last step is to render this component. We'll need an `id` to make it stateful, so we need to add the new `id` to `lib/pento_web/live/admin_dashboard_live.ex`:

```

...
{:ok,
 socket
 |> assign(:survey_results_component_id, "survey-results")
 |> assign(:user_activity_component_id, "user-activity")}
...

```

The `AdminDashboardLive` live view needs to hold on to awareness of the component's ID so that it can use it to tell the component to update later on. More on that in a bit.

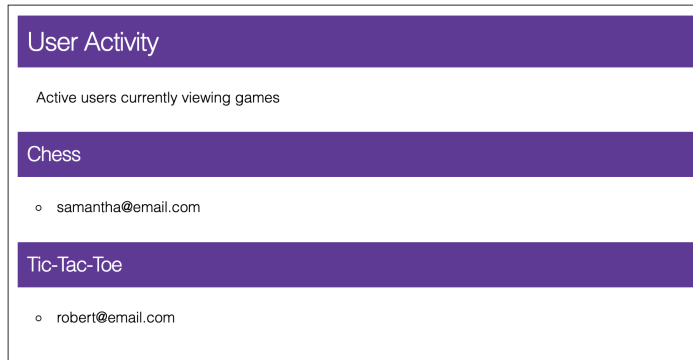
Now, the `AdminDashboardLive` template can render the component:

```

distributed_dashboard/pento/lib/pento_web/live/admin_dashboard_live.html.leex
<%= live_component @socket,
  PentoWeb.UserActivityLive,
  id: @user_activity_component_id %>

```

The code is simple and direct. It renders a component, passing only the new `id` from `@user_activity_component_id`. Now, you can try it out. Open a few different browser sessions for different users and navigate each to a product show page. Then, point yet another browser to `/admin-dashboard`, and you'll see the user activity component in all of its glory, like this:



Now, our site admins can see users engaging with products. So far, so good. There's a problem, though. When new users interact with the site, you won't be able to see them. Similarly, if a user navigates away from a given product's show page, the user activity list won't update in real-time. Admins need to refresh the page in order to get the latest list of active users. Fortunately, there's an easy remedy, and it has to do with PubSub.

Subscribe to Presence Changes

Recall that when we defined our `PentoWeb.Presence` module, we configured it to use our application's PubSub server. This means that, whenever we change the state of the data in the Presence data store, for example with a call to the `track/4` function, a "presence_diff" event will get broadcast over the specified topic.

So, all we need to do is subscribe the `AdminDashboardLive` view to the "user_activity" topic we provided in our call to `Presence.track/4`. Then, we'll implement a `handle_info/2` function in `AdminDashboardLive` and teach it to respond to messages over this topic by updating the `UserActivityLive` component. When the component updates, it will call `update/2` again, which will re-fetch the latest list of present users.

Let's put the plan into action.

Add a module attribute with the "user_activity" topic to `AdminDashboardLive`, and update the `mount/3` to subscribe to this topic:

```
# lib/pento_web/live/admin_dashboard_live.ex
...
@survey_topic "survey_results"
@user_activity_topic "user_activity"

def mount(_params, _session, socket) do
  if connected?(socket) do
    Endpoint.subscribe(@survey_topic)
    Endpoint.subscribe(@user_activity_topic)
```

```

    end
  ...
end
...

```

With that done, all that remains is responding to the PubSub broadcasts via `handle_info/2`. Let's finish this feature, and put a bow on it.

Respond to Presence Events

Now that `AdminDashboardLive` is subscribed to the "user_activity" topic, we'll implement the `handle_info/2` function for the "presence_diff" event, like this:

```

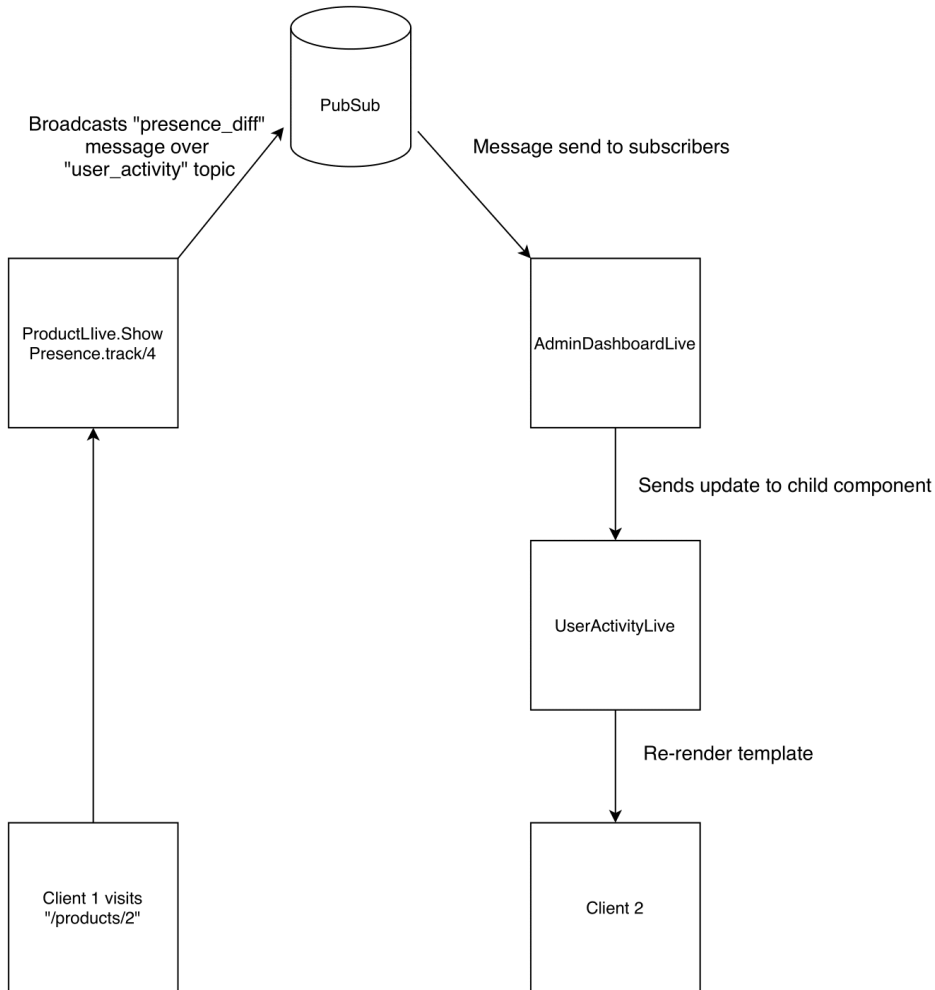
distributed_dashboard/pento/lib/pento_web/live/admin_dashboard_live.ex
def handle_info(%{event: "presence_diff"}, socket) do
  send_update(
    UserActivityLive,
    id: socket.assigns.user_activity_component_id
  ){:noreply, socket}
end

```

We call a basic `send_update/3` function, providing the component name and ID. This will tell the `UserActivityLive` component to update, invoking its `update/2` function.

Remember, the `update/2` function already invokes the `assign_user_activity/1` reducer. That function fetches a fresh list of user presences per product, so we're done!

With a few dozen lines of code, we've implemented an interactive distributed solution for tracking user activity. It's a solution that will work equally well on a standalone server or a worldwide distributed cluster. The following figure shows what's happening.



This figure shows exactly what happens when a new user visits a product page. First, the `Presence.track/4` function is invoked. This tracks the given user based on the running process, updating the Presence data store accordingly. With this change to Presence state, the Presence service sends out a message via PubSub. When that happens, the `AdminDashboardLive` view tells the `UserActivityLive` component to update.

With just a few lines of code to respond to a PubSub message, the `UserActivityLive` component updates! That's the beauty of Presence, and of LiveView. Presence and PubSub allow us to supercharge our live view with the ability to reflect the state of our distributed application, while writing very little new code.

It's been a short chapter, but an intense one. It's time to wrap up.

Your Turn

Developers can extend single-page apps to react to distributed events with incremental effort. Phoenix PubSub and Presence bring the powerful capabilities of distributed Elixir to LiveView. They seamlessly integrate into LiveView code to allow you to build live views that represent the state of your entire application. You can even maintain your beautifully layered LiveView components alongside these technologies by using the `send_update/3` function to communicate distributed state changes to child components. LiveView components and Phoenix PubSub work together to support complex, distributed state management systems with ease.

Now, you can put these skills to work.

Give It A Try

This problem lets you use Presence and PubSub to update a view.

- Use PubSub and Presence to track the number of people taking a survey.
- Add a new component to the admin dashboard view to display this total list of survey-taking users.
- What happens when a user navigates away from a survey page? Did your list of survey-taking users update on its own, without you writing any new code to support this feature? Think through why this is.

Next Time

With a working distributed dashboard, the admin features of the site are now complete. Next, we build a set of test cases to make sure the site does not regress as new features are released. We'll use the CRC strategy to build test cases that are organized, easy-to-read, and that scale well to cover a wide range of scenarios. Keep this ball rolling by turning the page!

Test Your Live Views

By now, you've seen most of what LiveView has to offer. You've used generators to build and customize a full-fledged CRUD feature set. You've built individual forms with and without schemas behind them to express inputs and validation. You've composed complex views with simpler components. You've even extended live views with Phoenix PubSub for real-time updates in your distributed system.

So far, our workflow has consisted of writing tiny bits of code and verifying them by running IEx sessions and looking at browser windows. This flow works well in this book because it offers excellent opportunities for teaching dense concepts. In reality, most developers *build tests as they go*. By writing tests, you'll gain the ability to make significant changes with confidence that your tests will catch breakages as they happen. In this chapter, you'll finally get to write some tests.

Testing for live views is easier than testing for most web frameworks for several reasons. First, the CRC pattern lends itself nicely to robust unit testing because we can write individual tests for the small, single-purpose functions that compose into the CRC workflow. LiveView's test tooling makes a big difference too. Though LiveView is young, the LiveViewTest module offers a set of convenience functions to exercise live views without fancy JavaScript testing frameworks. You'll use this module directly in your ExUnit tests, which means that all of your live view tests can be written in Elixir. As a result, your live view tests will be fast, concurrent, and stable, which differs markedly from the experience of working with headless browser testing tools that introduce new external dependencies and can make consistency difficult to achieve.

Tests exist to instill confidence, and unstable tests erode that confidence. Building as much of your testing story as possible on pure Elixir will pay

dividends in your confidence and help you move quickly when building your LiveView applications.

In this chapter, we’re not going to spend much time beyond the narrow slice of testing where ExUnit meets our LiveView code. If you want to know more about Elixir testing, check out [Testing Elixir \[LM21\]](#) by Andrea Leopardi and Jeffrey Matthias. If you’re writing full applications using LiveView, you’ll eventually need to take a deeper dive into Elixir testing, and that book is a great place to start.

For now, we’ll test the survey results feature on the admin dashboard page to expose you to the testing techniques you’ll need when building live views.

What Makes CRC Code Testable?

Think of a test as a scientific experiment. The target of the experiment is a bit of code, and the thesis is that the code is working. Logically, each test is an experiment that does three things:

- Set up preconditions
- Provide a stimulus
- Compare an actual response to expectations

That definition is pretty broad, and covers a wide range of testing strategies and frameworks. We’re going to write three tests, of two specific types. Both types of tests will follow this broad pattern. One of the tests will be a *unit test*. We’ll write it to verify the behavior of the independent functions that set up the socket. We’ll also write two *integration tests* which will let us verify the interaction between components: one to test interactions *within* a live view process, and another to verify interactions *between* processes.

You might be surprised that we *won’t* be testing JavaScript. A big part of the LiveView value proposition is that it pushes much of the JavaScript interactions into the infrastructure, so we don’t have to deal with them. Because the Pento application has no custom JavaScript integrations, we don’t have to worry about testing JavaScript if we trust the LiveView JavaScript infrastructure.

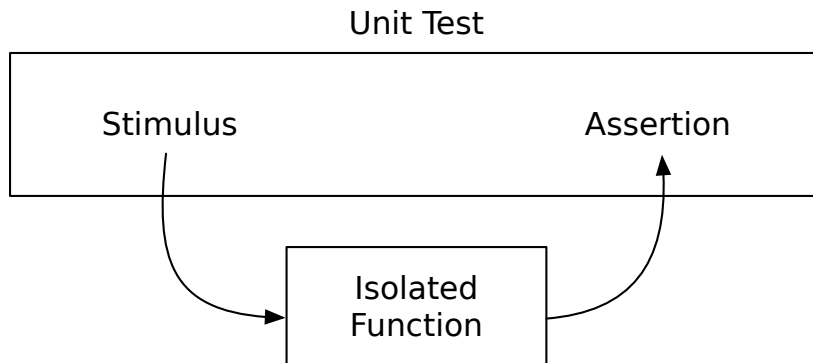
Instead, the integration tests we write will interact with LiveView machinery to examine the impact of page loads and events that flow through a live view. A good example of such a test is simulating a button click and checking the impact on the re-rendered live view template. Integration tests have the benefit of catching *integration problems*—problems that occur at the integration

points between different pieces of your system, in this case, the client and the server.

These integration tests are certainly valuable, but they can be brittle. For example, if the user interface changes the button into a link, then your test must be updated as well. That means this type of test is costly in terms of long-term maintenance. Sometimes it pays to isolate specific functions with complex behavior—like our live view reducer functions—and write pure tests for them. Such tests are called *unit tests* because they test one specific unit of functionality. Let's discuss a testing strategy that addresses both integrated and isolated tests.

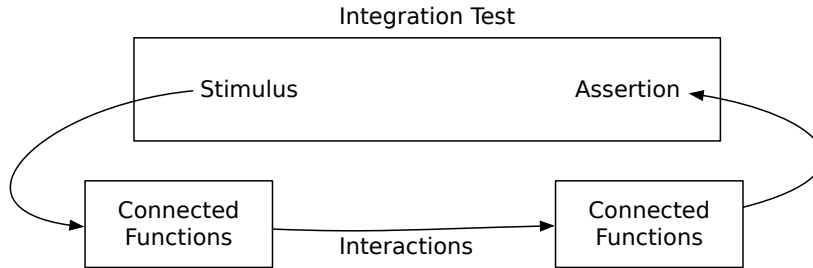
Isolation Vs. Integration

Pure unit tests call one function at a time, and then check expectations with one or more assertions. If you've designed your code well, you should find lots of opportunities for unit tests. By filling up your application's functional core with pure, predictable functions, and by adhering to the CRC pattern, you'll find yourself with many small, isolated functions that can be tested with small, isolated unit tests, as follows.



Unit tests encourage *depth*. Such tests don't require much ceremony so programmers can write more of them and cover more scenarios quickly and easily. Unit tests also allow *loose coupling* because they don't rely on specific interactions. Building code that's friendly to unit tests also lets you take advantage of other techniques like property based testing. This technique uses generated data to verify code and makes it even easier to create unit tests that cover an in-depth range of inputs. Read more about it in [Property-Based Testing with PropEr, Erlang, and Elixir \[Heb19\]](#) by Fred Hebert.

In contrast, integration tests check the *interaction between application elements*, like this:



As the figure shows, integration tests check interactions between different parts of the same system. These kind of tests offer testing *breadth* by exercising a wider swath of your application. The cost is tighter coupling, since integration tests rely on specific interactions between parts of your system. Of course, that coupling will exist whether we test it or not.

So, which types of tests should you use? In short, good developers need both. In this chapter, you'll start with some unit tests written with pure ExUnit. Then, you'll move on to two different types of integration tests. One will use LiveViewTest features to interact with your live view, and another will use LiveViewTest along with plain Elixir message passing to simulate PubSub messages.

Start with Unit Tests

One of the best ways to make writing unit tests testing easier is to start with single-purpose, decoupled functions. The CRC pipelines we built throughout the first part of this book are perfect for unit tests. You could choose to test each constructor, reducer, and converter individually *as functions* by directly calling them within an ExUnit test without any of the LiveView test machinery. That's a unit test.

By exercising individual complex functions in unit tests with many different inputs you can exhaustively cover corner cases by that may be prone to failure. Then, you can write a smaller number of integration tests to confirm that the complex interactions of the system work as you expect them to.

For example, a mortgage calculator is likely to have many tests on the function that computes financial values, but only a few tests to make sure that those values show up correctly on the page when a user submits a request.

That's the approach we'll take in order to test the SurveyResultsLive component. We'll focus on a few of this component's functions that are the most complex and likely to fail: the ones that underpin the component's ability to obtain and filter survey results. Along the way, you'll write advanced unit tests composed of reducer pipelines. Then, we'll move on to the integration tests.

Unit Test Test Survey Results State

We'll begin with some unit tests that cover the `SurveyResultsLive` component's ability to manage survey results data in state. First up is the `assign_products_with_average_ratings/2` reducer function, which needs to handle both an empty survey results dataset and one with existing product ratings. First, we'll make sure the reducer creates the correct socket state when no product ratings exist. Start by defining a test module in `test/pento_web/live/survey_results_live_test.exs`, like this:

```
testing/pento/test/pento_web/live/survey_results_live_test.exs
defmodule PentoWeb.SurveyResultsLiveTest do
  use Pento.DataCase
  alias PentoWeb.SurveyResultsLive
```

Note the `use Pento.DataCase` line. This pulls in the `Pento.DataCase` behaviour which provides access to the `ExUnit` testing functions and provides our test with a connection to the application's test database.

You'll also notice that our module aliases the `SurveyResultsLive` component. That's the component we're testing in this module. We need to perform a few other aliases too. We'll use them to establish some fixtures and helper functions to simplify the creation of test data, like this:

```
testing/pento/test/pento_web/live/survey_results_live_test.exs
alias Pento.{Accounts, Survey, Catalog}

@create_product_attrs %{
  description: "test description",
  name: "Test Game",
  sku: 42,
  unit_price: 120.5
}

@create_user_attrs %{
  email: "test@test.com",
  password: "passwordpassword"
}

@create_user2_attrs %{
  email: "another-person@email.com",
  password: "passwordpassword"
}

@create_demographic_attrs %{
  gender: "female",
  year_of_birth: DateTime.utc_now.year - 15
}

@create_demographic2_attrs %{
  gender: "male",
  year_of_birth: DateTime.utc_now.year - 30
}
```

```

defp product_fixture do
  {:ok, product} = Catalog.create_product(@create_product_attrs)
  product
end

defp user_fixture(attrs \\ @create_user_attrs) do
  {:ok, user} = Accounts.register_user(attrs)
  user
end

defp demographic_fixture(user, attrs \\ @create_demographic_attrs) do
  attrs =
    attrs
    |> Map.merge(%{user_id: user.id})
  {:ok, demographic} = Survey.create_demographic(attrs)
  demographic
end

defp rating_fixture(stars, user, product) do
  {:ok, rating} = Survey.create_rating(%{
    stars: stars,
    user_id: user.id,
    product_id: product.id
  })

  rating
end

defp create_product(_) do
  product = product_fixture()
  %{product: product}
end

defp create_user(_) do
  user = user_fixture()
  %{user: user}
end

defp create_rating(stars, user, product) do
  rating = rating_fixture(stars, user, product)
  %{rating: rating}
end

defp create_demographic(user) do
  demographic = demographic_fixture(user)
  %{demographic: demographic}
end

defp create_socket(_) do
  %{socket: %Phoenix.LiveView.Socket{}}
end

```

Test fixtures create test data, and ours use module attributes to create User, Demographic, Product, and Rating records, followed by a few helpers that call on

our fixtures and return the newly created records. You'll see these helper functions, and their return values, in action in a bit.

Now that our test module is defined and we've implemented helper functions to create test data, we're ready to write our very first test. We'll start with a test that verifies the socket state when there are no product ratings. Open up a describe block and add a call to the `setup/1` function with the list of helpers that will create a user, product, and socket struct, like this:

```
testing/pento/test/pento_web/live/survey_results_live_test.exs
describe "Socket state" do
  setup [:create_user, :create_product, :create_socket]

  setup %{user: user} do
    create_demographic(user)
    user2 = user_fixture(@create_user2_attrs)
    demographic_fixture(user2, @create_demographic2_attrs)
    [user2: user2]
  end
end
```

Let's break it down. The `describe` function groups together a block of tests. Before each one of them, ExUnit will run the `setup` callbacks. Think of both `setup` functions as reducers. Both take an accumulator, called the context, which holds a bit of state for our tests to share. The first `setup` call provides a list of atoms. Each one is the name of a named `setup` function.¹ A `setup` function returns a map of data to merge into the context. The second `setup` function is a reducer that further transforms the context.

The named `setup` functions each create bits of data to add to the context. If you look at the `create_socket` named `setup` function, you'll see that it's nothing more than a pure Elixir function returning an empty `LiveView` socket to add to the context. By returning `%{socket: %Phoenix.LiveView.Socket{}}`, the `create_socket` `setup` function will add this key/value pair to the shared test context data structure. The other named `setup` functions are similar.

After running the named `setups`, ExUnit calls the `setup/1` function in which we establish the demographic records for two test users. The function is called with an argument of the context and the return value of this function likewise gets added to the context map—this time the key/value pairs from the returned keyword list are added to the context map. The result is that our code builds a map, piece by piece, and passes it into each test in the `describe` block.

We're finally ready to write the unit test. Create a test block within the `describe` block that matches the context we created in the named `setup`. For this test,

1. https://hexdocs.pm/ex_unit/ExUnit.Callbacks.html#setup/1

we only need the socket from the context map, so we'll pull it out using pattern matching, like this:

```
test "no ratings exist", %{socket: socket} do
  # coming soon!
end
```

Let's pause and think through what we're testing here and try to understand what behavior we *expect* to see. This test covers the function `assign_products_with_average_ratings/1` when no product ratings exist. If it's working correctly, the socket should contain a key of `:products_with_average_ratings` that points to a value that looks something like this:

```
[{"Test Game", 0}]
```

The result tuples should still exist, but with a rating of 0. That's our expectation. We'll setup our test assertion like this:

```
test "no ratings exist", %{socket: socket} do
  socket =
    socket
    |> SurveyResultsLive.assign_products_with_average_ratings()
  assert
    socket.assigns.products_with_average_ratings ==
      [{"Test Game", 0}]
end
```

Our test won't work as-is, though. The `assign_products_with_average_ratings/1` function expects that both the `:age_filter` and `:gender_filter` keys are present in `socket.assigns`. So, we'll need to establish those keys with the help of a reducer pipeline like this:

```
testing/pento/test/pento_web/live/survey_results_live_test.exs
```

```
test "no ratings exist", %{socket: socket} do
  socket =
    socket
    |> SurveyResultsLive.assign_age_group_filter()
    |> SurveyResultsLive.assign_gender_filter()
    |> SurveyResultsLive.assign_products_with_average_ratings()
  assert socket.assigns.products_with_average_ratings == [{"Test Game", 0}]
end
```

Perfect. We use the same reducers to set up the socket state in the test as we used in the live view itself. That's a sign that the code is structured correctly. Building a component with small, single-purpose reducers let us test some complex corner cases with a focused unit test. Testing a socket with no user ratings is a good example of the kinds of scenarios unit tests handle well.

Let's quickly add another, similar, test of the `assign_products_with_average_ratings/1` reducer's behavior when ratings *do* exist, like this:

```
testing/pento/test/pento_web/live/survey_results_live_test.exs
```

```
test "ratings exist", %{socket: socket, product: product, user: user} do
  create_rating(2, user, product)
  socket =
    socket
    |> SurveyResultsLive.assign_age_group_filter()
    |> SurveyResultsLive.assign_gender_filter()
    |> SurveyResultsLive.assign_products_with_average_ratings()
  assert socket.assigns.products_with_average_ratings == [{"Test Game", 2.0}]
end
```

Thanks to the composability of our reducer functions, writing tests is quick and easy and can be handled entirely in the world of ExUnit. The same functions that set up the socket within the live views also set up the socket in our tests. We haven't even brought in any `LiveViewTest` functions, but our tests are already delivering value. This block of code is exactly the type of test that might catch regressions that a refactoring exercise might leave behind.

Cover Corner Cases in Unit Tests

The next core behavior to test is the survey results chart's ability to filter results based on age and gender. The `assign_age_group_filter/1` function manages the age group key and we'll focus our attention there now. Testing the ability of this reducer to manage the `:age_group_filter` piece of socket state could require significant setup code within integration tests, but several quick assertions in a unit test can make quick work of the problem.

The function's behavior is relatively complex. We'll need to cover several different scenarios:

- When the socket has no `:age_group_filter` key, `assign_age_group_filter/1` should add an `:age_group_filter` key with `all`.
- When a socket has the `18 and under` value for `:age_group_filter`, `assign_age_group_filter/1` should *not* replace it with `all`.
- Calling `assign_products_with_average_ratings/1` when the socket has an `:age_group_filter` of `18 and under` should add the correct, filtered product ratings to the socket.

Thanks to the reusable and composable nature of our reducers, we can construct a test pipeline that allows us to exercise and test each of these scenarios in one beautiful flow.

Create Unit Tests to Clarify Concepts

The line between unit and integration tests is not always clear. Sometimes, a function tests an isolated *concept* that's broken into multiple closely related functions. In these scenarios, sometimes it's helpful to build a multi-stage test. Let's see how it works.

Open up `test/pento_web/live/survey_results_live_test.exs` and add a test block within the existing `describe`:

```
test "ratings are filtered by age group", %{
  socket: socket,
  user: user,
  product: product,
  user2: user2} do

  create_rating(2, user, product)
  create_rating(3, user2, product)

  # coming soon!
end
```

The test uses our helper function to create two ratings. The first is for a user in the 18 and under demographic and the other is not.

Now, we're ready to construct our reducer pipeline and test it. We'll start by testing the first of the three scenarios we outlined. We'll test that, when called with a `socket` that does not contain an `:age_group_filter` key, the `assign_age_group_filter/1` reducer returns a `socket` that sets that key to a value of `"all"`. Call `SurveyResultsLive.assign_age_group_filter/1` with the `socket` from the test context, and establish your assertions, like this:

```
test "ratings are filtered by age group",
  %{socket: socket, user: user, product: product, user2: user2} do
  create_rating(2, user, product)
  create_rating(3, user2, product)

  socket =
    socket
    |> SurveyResultsLive.assign_age_group_filter()

  assert socket.assigns.age_group_filter == "all"
end
```

Run the test by specifying the test file and line number, and you'll see it pass:

```
[pento] → mix test test/pento_web/live/survey_results_live_test.exs:109
Excluding tags: [:test]
Including tags: [line: "109"]

.

Finished in 0.1 seconds
```

3 tests, 0 failures, 2 excluded

Randomized with seed 48183

Clean and green. Now we're ready to test our second scenario. When the `assign_age_group_filter/1` function is called with a socket that already contains an `:age_group_filter` key, it should retain the value of that key. We'll test this scenario by updating the *same* socket from our existing test to use the 18 and under filter, like this:

```
test "ratings are filtered by age group",
  %{socket: socket, user: user, product: product, user2: user2} do
    create_rating(2, user, product)
    create_rating(3, user2, product)

    socket =
      socket
      |> SurveyResultsLive.assign_age_group_filter()

    assert socket.assigns.age_group_filter == "all"

    socket =
      update_socket(socket, :age_group_filter, "18 and under")
      |> SurveyResultsLive.assign_age_group_filter()

    assert socket.assigns.age_group_filter == "18 and under"
  end

defp update_socket(socket, key, value) do
  %{socket | assigns: Map.merge(socket.assigns, Map.new([{key, value}])))}
end
```

The `update_socket` helper function sets the `:age_group_filter` to 18 and under and pipes the result into `assign_age_group_filter/1` before running the last assertion.

Tie Stages Together in a Pipeline

This code works, but we can do better. The same pipes you built in the first part of this book will work well in unit tests as well. You need only write a tiny custom helper function to glue the example together. Open up the test file and add the following below the `update_socket/3` helper:

```
testing/pento/test/pento_web/live/survey_results_live_test.exs
```

```
defp assert_keys(socket, key, value) do
  assert socket.assigns[key] == value
  socket
end
```

We created an assertion reducer. This function is a bit different than typical reducers. Rather than transforming the socket, this reducer's job is to call the `assert` macro, and then return the socket unchanged. The job of the function

is twofold. It calls the assertion, and keeps the integrity of the pipeline intact by returning the element with which it was called.

Now, we can assemble our test pipeline like this:

```
testing/pento/test/pento_web/live/survey_results_live_test.exs
test "ratings are filtered by age group", %{
  socket: socket,
  user: user,
  product: product,
  user2: user2} do
  create_rating(2, user, product)
  create_rating(3, user2, product)

  socket
  |> SurveyResultsLive.assign_age_group_filter()
  |> assert_keys(:age_group_filter, "all")
  |> update_socket(:age_group_filter, "18 and under")
  |> SurveyResultsLive.assign_age_group_filter()
  |> assert_keys(:age_group_filter, "18 and under")
  |> SurveyResultsLive.assign_gender_filter()
  |> SurveyResultsLive.assign_products_with_average_ratings()
  |> assert_keys(:products_with_average_ratings, [{"Test Game", 2.0}])
end
```

That's much better! The test now unfolds like a story. Each step is a reducer with a socket accumulator. Then, we use our new helper to check each key.

We can chain further reducers and assertions onto our pipeline to test the final scenario. The `assign_products_with_average_ratings/1` function should populate the socket with the correct product ratings, given the provided filters, like this:

```
testing/pento/test/pento_web/live/survey_results_live_test.exs
defp assert_keys(socket, key, value) do
  assert socket.assigns[key] == value
  socket
end
```

There are no surprises here. The extra assertion looks like it belongs. Building in this kind of conceptual density without sacrificing readability is what Elixir is all about.

Now, if you run all of the tests, in this file, you'll see them pass:

```
[pento] → mix test test/pento_web/live/survey_results_live_test.exs
...

Finished in 0.2 seconds
3 tests, 0 failures

Randomized with seed 543381
```

The composable nature of our reducer functions makes them highly testable. It's easy to test the functionality of a single reducer under a variety of circumstances, or to string together any set of reducers to test the combined functionality of the pipelines that support your live view's behavior. With a little help from our `assert_keys/3` function, we constructed a beautiful pipeline to test a set of scenarios within one easy-to-read flow.

Now that we've written a few unit tests that validate the behavior of the reducer building blocks of our live view, let's move on to testing LiveView features and behaviors with the help of the `LiveViewTest` module.

Integration Test LiveView Interactions

Where unit tests check isolated bits of code, integration tests verify the interactions between parts of a system. In this section, we'll write an integration test that validates interactions within *a single live view*. We'll focus on testing the behavior of the survey results chart filter. In the next section, we'll write another integration test that checks interactions *between processes* that manage live updates when new ratings come in.

We'll write both tests *without any JavaScript*. This statement should get some attention from anyone used to the overhead of bringing in an external JavaScript dependency to write integration tests that are often slow and flaky. So, we'll say it again, louder this time. You don't need JavaScript to test LiveView!

We'll use the `LiveViewTest` module's special LiveView testing functions to simulate liveView connections without a browser. Your tests can mount and render live views, trigger events, and then execute assertions against the rendered view. That's the whole LiveView lifecycle.

You might be concerned about leaving JavaScript untested, but remember. The JavaScript that supports LiveView is part of the LiveView framework itself, so you also don't have to leverage JavaScript to *test* your live views. You can trust that the JavaScript in the framework does what it's supposed to do, and focus your attention on testing the specific behaviors and features that you built into your own live view, in pure Elixir.

As a result, the integration tests for LiveView are quick and easy to write and they run fast and concurrently. Once again, LiveView maintains a focused mindset on the server, in pure Elixir. Let's write some tests.

Write an Integration Test

We've unit tested the individual pieces of code responsible for our component's filtering functionality. Now it's time to test that same filtering behavior by

exercising the overall live view. We'll write one test together to introduce LiveView's testing capabilities. Then, we'll leave it up to you to add more tests to cover additional scenarios. Our test will simulate a user's visit to /admin-dashboard, followed by their filter selection of the 18 and under age group. The test will verify an updated survey results chart that displays product ratings from users in that age group.

Because components run in their parent's processes, we'll focus our tests on the AdminDashboardLive view, which is the SurveyResultsLive component's parent. We'll use LiveViewTest helper functions to run our admin dashboard live view and interact with the survey results component. Along the way, you'll get a taste for the wide variety of interactions that the LiveViewTest module allows you to test.

Let's begin by setting up a LiveView test for our AdminDashboardLive view.

Set Up The LiveView Test Module

It's best to segregate unit tests and integration tests into their own modules, so create a new file test/pento_web/live/admin_dashboard_live_test.exs and key this in:

```
testing/pento/test/pento_web/live/admin_dashboard_live_test.exs
defmodule PentoWeb.AdminDashboardLiveTest do
  use PentoWeb.ConnCase

  import Phoenix.LiveViewTest
  alias Pento.{Accounts, Survey, Catalog}

  @create_product_attrs %{
    description: "test description",
    name: "Test Game",
    sku: 42,
    unit_price: 120.5
  }

  @create_demographic_attrs %{
    gender: "female",
    year_of_birth: DateTime.utc_now.year - 15
  }

  @create_demographic_over_18_attrs %{
    gender: "female",
    year_of_birth: DateTime.utc_now.year - 30
  }

  @create_user_attrs %{email: "test@test.com", password: "passwordpassword"}
  @create_user2_attrs %{email: "test2@test.com", password: "passwordpassword"}
  @create_user3_attrs %{email: "test3@test.com", password: "passwordpassword"}

  defp product_fixture do
    {:ok, product} = Catalog.create_product(@create_product_attrs)
    product
  end
end
```

```

defp user_fixture(attrs \\ @create_user_attrs) do
  {:ok, user} = Accounts.register_user(attrs)
  user
end

defp demographic_fixture(user, attrs) do
  attrs =
    attrs
    |> Map.merge(%{user_id: user.id})
  {:ok, demographic} = Survey.create_demographic(attrs)
  demographic
end

defp rating_fixture(user, product, stars) do
  {:ok, rating} = Survey.create_rating(%{
    stars: stars,
    user_id: user.id,
    product_id: product.id
  })

  rating
end

defp create_product(_) do
  product = product_fixture()
  %{product: product}
end

defp create_user(_) do
  user = user_fixture()
  %{user: user}
end

defp create_demographic(user, attrs \\ @create_demographic_attrs) do
  demographic = demographic_fixture(user, attrs)
  %{demographic: demographic}
end

defp create_rating(user, product, stars) do
  rating = rating_fixture(user, product, stars)
  %{rating: rating}
end

```

We're doing a few things here. First, we define our test module. Then, we use the `PentoWeb.ConnCase` behavior that will allow us to route to live views using the test connection. Using this behaviour gives our tests access to a context map with a key of `:conn` pointing to a value of the text connection. We also import the `LiveViewTest` module to give us access to LiveView testing functions. Finally, we throw in some fixtures we will use to create our test data.

Now that our module is set up, go ahead and add a describe block to encapsulate the feature we're testing—the survey results chart functionality:


```

testing/pento/test/pento_web/live/admin_dashboard_live_test.exs
describe "Survey Results" do
  setup [:register_and_log_in_user, :create_product, :create_user]

  setup %{user: user, product: product} do
    create_demographic(user)
    create_rating(user, product, 2)

    user2 = user_fixture(@create_user2_attrs)
    create_demographic(user2, @create_demographic_over_18_attrs)
    create_rating(user2, product, 3)

    :ok
  end
end

```

Two calls to `setup/1` seed the test database with a product, users, demographics, and ratings. One of the two users is in the 18 and under age group and the other is in another age group. Then, we create a rating for each user.

We're also using a test helper created for us way back when we ran the authentication generator—`register_and_log_in_user/1`. This function creates a context map with a logged in user, a necessary step because visiting the `/admin-dashboard` route requires an authenticated user.

Now that our setup is completed, we'll write the body of the test.

Test The Survey Chart Filter

As with the other testing module, this one will group tests together into a common describe block. Add a test within the describe, like this:

```

test "it filters by age group", %{conn: conn} do
  # coming soon!
end

```

We'll fill in the details of our test after making a plan. We need to:

- Mount and render the live view
- Find the age group filter drop down menu and select an item from it
- Assert that the re-rendered survey results chart has the correct data and markup

This is the pattern you'll apply to testing live view features from here on out. Run the live view, target some interaction, test the rendered result. This pattern should sound a bit familiar. Earlier on in this chapter we said that all of the types of tests will adhere to this pattern:

- Set up preconditions
- Provide a stimulus
- Compare an actual response to expectations

LiveView tests are not different.

To mount and render the live view, we'll use the `LiveViewTest.live/2` function. This function spawns a simulated LiveView process. We call the function with the test context struct and the path to the live view we want to run and render:

```
test "it filters by age group", %{conn: conn} do
  {:ok, view, _html} = live(conn, "/admin-dashboard")
end
```

The call to `live/2` returns a three element tuple with `:ok`, the LiveView process, and the rendered HTML returned from the live view's call to `render/1`. We don't need to access that HTML in this test, so we ignore it.

Remember, components run in their parent's process. That means the test *must* start up the `AdminDashboardLive` view, rather than rendering *just* the `SurveyResultsLive` component. By spawning the `AdminDashboardLive` view, we're *also* rendering the components that the view is comprised of. This means our `SurveyResultsLive` component is up and running and is rendered within the `AdminDashboardLive` view represented by the returned view variable. So, we'll be able to interact with elements within that component and test that it re-renders appropriately within the parent live view, in response to events. This is the correct way to test LiveView component behavior within a live view page.

Testing LiveView Components

To test the rendering of a component in isolation, you can use the `LiveViewTest.render_component/2` function. This will render and return the markup of the specified component, allowing you to write assertions against that markup. This is useful in writing unit tests for stateless components. To test the behavior of a component—i.e. how it is mounted within a parent live view and how events impact its state—you'll need to run the parent live view with the `live/2` function and target events at DOM elements contained within the component.

The test has a running live view, so we're ready to select the 18 and under age filter. Let's interact with our running live view to do exactly that.

Simulate an Event

The test can trigger LiveView interactions using helper functions from `LiveViewTest`—all you need to do is identify the page element you want to interact with. For a comprehensive look at the rapidly growing list of such functions, check the `LiveViewTest` documentation.²

2. https://hexdocs.pm/phoenix_live_view/Phoenix.LiveViewTest.html#functions

We'll use the `element/3` function to find the age group drop-down on the page. First, we'll add a unique ID attribute to the form element so that we can find it with the `element/3` function, as you can see here:

```
testing/pento/lib/pento_web/live/survey_results_live.html.leex
```

```
<form
  phx-change="age_group_filter"
  phx-target="<%= @myself%>"
  id="age-group-form">
```

Now we can target this element with the `element/3` function like this:

```
test "it filters by age group", %{conn: conn} do
  {:ok, view, _html} = live(conn, "/admin-dashboard")
  html =
    view
    |> element("#age-group-form")
end
```

The `element/3` function accepts three arguments—the live view whose element we want to select, any query selector, and some optional text to narrow down the query selector even further. If no text filter is provided, it must be true that the query selector returns a single element.

Now that we've selected our element, let's take a closer look. Add the following to your test in order to inspect it:

```
test "it filters by age group", %{conn: conn} do
  {:ok, view, _html} = live(conn, "/admin-dashboard")
  html =
    view
    |> element("#age-group-form")
    |> IO.inspect
end
```

Then, run the test and you'll see the element inspected into the terminal:

```
[pento] → mix test test/pento_web/live/admin_dashboard_live_test.exs:75
Compiling 1 file (.ex)
Excluding tags: [:test]
Including tags: [line: "75"]

...
#Phoenix.LiveViewTest.Element<
  selector: "#age-group-form",
  text_filter: nil,
  ...
>
.
Finished in 0.3 seconds
```

Nice! We can see that the `element/3` function returned a `Phoenix.LiveViewTest.Element` struct. Let's use it to fire a form change event that selects the 18 and under option, like this:

```
test "it filters by age group", %{conn: conn} do
  {:ok, view, _html} = live(conn, "/admin-dashboard")
  html =
    view
    |> element("#age-group-form")
    |> render_change(%{"age_group_filter" => "18 and under"})
end
```

The `render_change/2` function is one of the functions you'll use to simulate user interactions when testing live views. It takes an argument of the selected element, along with some params, and triggers a `phx-change` event.

`testing/pento/lib/pento_web/live/survey_results_live.html.leex`

```
<form
  phx-change="age_group_filter"
  phx-target="<%= @myself%>"
  id="age-group-form">
```

The `phx-change` attribute of the given element determines the name of the event and the `phx-target` attribute determines which component gets the message. Recall that the age group form element we selected looks like this:

`testing/pento/lib/pento_web/live/survey_results_live.html.leex`

```
<form
  phx-change="age_group_filter"
  phx-target="<%= @myself%>"
  id="age-group-form">
```

So, we'll send the message `"age_group_filter"` to the target `@myself`, which is the `SurveyResultsLive` component. The `phx-change` event will fire with the params we provided to `render_change/2`. This event will trigger the associated handler, thus invoking the reducers that update our socket, eventually re-rendering the survey results chart with the filtered product rating data. To refresh your memory:

`testing/pento/lib/pento_web/live/survey_results_live.ex`

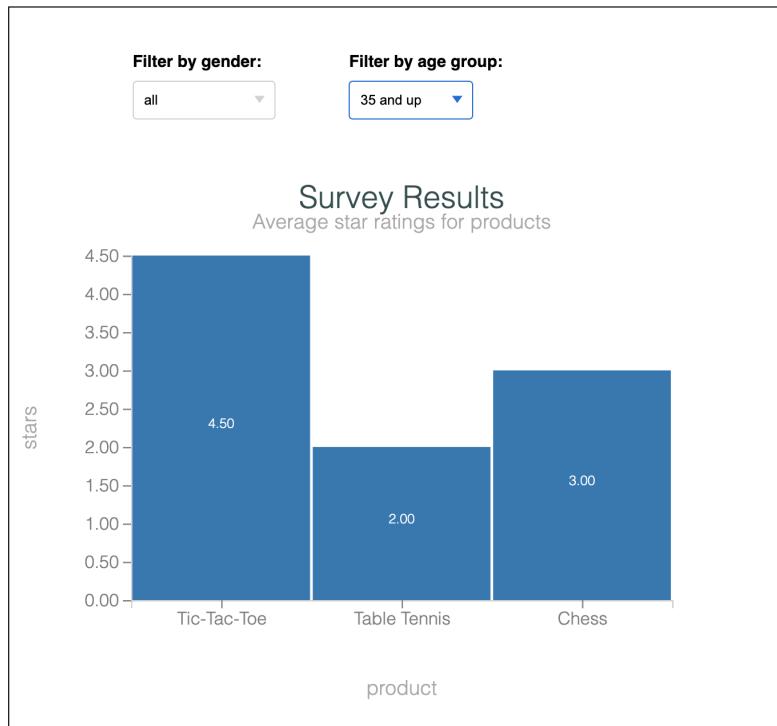
```
def handle_event(
  "age_group_filter",
  %{"age_group_filter" => age_group_filter},
  socket) do
  {:noreply,
   socket
   |> assign_age_group_filter(age_group_filter)
   |> assign_products_with_average_ratings()
   |> assign_dataset()
  }
```

```
|> assign_chart()
|> assign_chart_svg()
end
```

Now that we have our test code in place to trigger the form event, and we know how we expect our component to behave when it receives that event, we're ready to write our assertions.

The call to `render_change/2` will return the re-rendered template. Let's add an assertion that the re-rendered chart displays the correct data. Recall that the bars in our survey results chart are labeled with the average star rating for the given product, like this:

Survey Results



So, we'll need to write an assertion that looks for the correct average star rating to be present on the bar for a given game in the selected age group. But how will we select the correct page element in order to write our assertion?

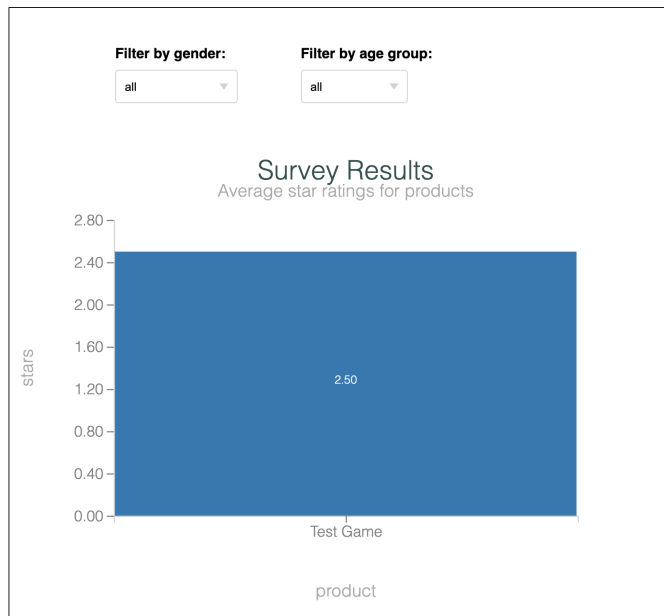
This is a great time to make use of another `LiveViewTest` convenience. The `open_browser/1` function lets us inspect a browser page at a given point in the

run of a test. Let's use it now to inspect the view so we can get a better sense of what test assertion we need to write. Add the following to your test:

```
test "it filters by age group", %{conn: conn} do
  {:ok, view, _html} = live(conn, "/admin-dashboard")
  html =
    view
    |> open_browser()
    |> element("#age-group-form")
    |> render_change(%{"age_group_filter" => "18 and under"})
end
```

Now, run the test via `mix test test/pento_web/live/admin_dashboard_live_test.exs:75` and you should see your default browser open and display the following page:

Survey Results



You can open up the element inspector in order to select the “Test Game” column’s label, like this:



Now you know exactly what element to select—a `<title>` element that contains the expected average star rating.

So, what should that average star rating be? Revisit the test data we established in our setup block here:

```
setup %{user: user, product: product} do
  create_demographic(user)
  create_rating(user, product, 2)

  user2 = user_fixture(@create_user2_attrs)
  create_demographic(user2, @create_demographic_over_18_attrs)
  create_rating(user2, product, 3)
end
:ok
```

You can see that we created two ratings for the test product—a 2 star rating for the user in the “18 and under” age group and a 3 star rating for the other user. So, if we filter survey results by the “18 and under” age group, we would expect the “Test Game” bar in our chart to have a title of 2.0. Let’s add our assertion here:

```
testing/pento/test/pento_web/live/admin_dashboard_live_test.exs
test "it filters by age group", %{conn: conn} do
  {:ok, view, _html} = live(conn, "/admin-dashboard")
  params = %{ "age_group_filter" => "18 and under" }
  assert view
    |> element("#age-group-form")
    |> render_change(params) =~ "<title>2.00</title>"
end
```

Now, you can run your test and it will pass! The `LiveViewTest` module provided us with everything we needed to mount and render a connected live view, target elements within that live view—even elements nested within child

components—and assert the state of the view after firing DOM events against those elements.

The test code, like much of the Elixir and LiveView code we’ve been writing, is clean and elegantly composed with a simple pipeline. All of the test code is written in Elixir with `ExUnit` and `LiveViewTest` functions. This made it quick and easy for us to conceive of and write our test. Our test runs fast, and it’s highly reliable. We didn’t need to bring in any JavaScript dependencies or undertake any onerous setup to test our LiveView feature. LiveView tests allow us to focus on the live view behavior we want to test—we don’t need JavaScript because we trust that the JavaScript in the LiveView framework will work the way it should.

We only saw a small subset of the `LiveViewTest` functions that support LiveView testing here. We used `element/3` and `render_change/2` to target and fire our form change event. There are many more `LiveViewTest` functions that allow you to send any number of DOM events—blurs, form submissions, live navigation and more.

We won’t get into all of those functions here. Instead, we’ll let you explore more of them on your own. There is one more testing task we’ll tackle together though. In the last chapter, you provided real-time updates to the admin dashboard with the help of `PubSub`. `LiveViewTest` allows us to test this distributed real-time functionality with ease.

Verify Distributed Realtime Updates

Testing message passing in a distributed application can be painful, but `LiveViewTest` makes it easy to test the `PubSub`-backed real-time features we’ve built into our admin dashboard. That is because LiveView tests interact with views via process communication. Because `PubSub` uses simple Elixir message passing, testing a live view’s ability to handle such messages is a simple matter of using `send/2`.

In this section, we’ll write a test to verify the admin dashboard’s real-time updates that fire when it receives a `"rating_created"` message. We’ll use a call to `send/2` to deliver the appropriate message to the view and then use the `render` function to test the result.

Set Up the Test

The real-time survey results chart test will follow the same LiveView test pattern we used earlier on. Remember, these are the steps:

- Mount and render the connected live view

- Interact with that live view—in this case, by sending the `rating_created` message to the live view
- Re-render the view and verify changes in the resulting HTML

Add the test to `test/pento_web/live/admin_dashboard_live_test.exs` within the current `describe` block:

```
test "it updates to display newly created ratings",
  %{conn: conn, product: product} do
  # coming soon!
end
```

That's a basic test that receives the connection and a product. Now, spawn the live view with `live/2`, like this:

```
test "it updates to display newly created ratings",
  %{conn: conn, product: product} do
  {:ok, view, html} = live(conn, "/admin-dashboard")
end
```

Add a Rating

Before we target our interaction and establish some assertion, let's think about what changes should occur on the page. Thanks to our `setup` block, we already have one product with two ratings—one with a star rating of 2 and the other with a star rating of 3. So, we know our survey results chart will render a bar for the “Test Game” product with a label of 2.50. We can verify this assumption with the help of the `open_browser/0` function, like so:

```
test "it updates to display newly created ratings",
  %{conn: conn, product: product} do
  {:ok, view, html} = live(conn, "/admin-dashboard")
  open_browser(view)
end
```

Perfect. Run the test like this to see the browser state:

```
[pento] → mix test test/pento_web/live/admin_dashboard_live_test.exs:84
```

You'll see this page open in the browser:

Survey Results



Now, you can see that the chart does in fact have a bar with a `<title>` element containing the text 2.50. That's the initial value, but it will change. We'll create a new rating to change this average star rating title and then send the `rating_created` message to the live view. Finally, we'll check for the changed `<title>` element.

Before making any changes though, the test should verify the initial 2.50 title element, like this:

```
testing/pento/test/pento_web/live/admin_dashboard_live_test.exs
test "it updates to display newly created ratings",
  %{conn: conn, product: product} do
  {:ok, view, html} = live(conn, "/admin-dashboard")
  assert html =~ "<title>2.50</title>"
```

It's a basic assertion to validate the starting state of the page. Now, let's create a new user, demographic and rating with a star value of 3, like this:

```
testing/pento/test/pento_web/live/admin_dashboard_live_test.exs
test "it updates to display newly created ratings",
  %{conn: conn, product: product} do
  {:ok, view, html} = live(conn, "/admin-dashboard")
  assert html =~ "<title>2.50</title>"
  user3 = user_fixture(@create_user3_attrs)
  create_demographic(user3)
```

```
create_rating(user3, product, 3)
```

Perfect. We're ready to trigger the live view interaction by sending the event to the view.

Trigger an Interaction with send/2

Recall that new ratings trigger PubSub "rating_created" messages to be broadcast over the "survey_results" topic with an empty payload. Since the AdminDashboardLive live view is subscribed to that same topic, it will receive a message that looks like this:

```
%{event: "rating_created", payload: %{}}
```

The AdminDashboardLive view implements the following handle_info/2 event handler for this event:

```
testing/pento/lib/pento_web/live/admin_dashboard_live.ex
def handle_info(%{event: "rating_created"}, socket) do
  send_update(
    SurveyResultsLive,
    id: socket.assigns.survey_results_component_id
  ){:noreply, socket}
end
```

In order to test the admin dashboard's ability to handle this message and update the template appropriately, we can manually deliver the same message with send/2, like this:

```
testing/pento/test/pento_web/live/admin_dashboard_live_test.exs
test "it updates to display newly created ratings",
  %{conn: conn, product: product} do
  {:ok, view, html} = live(conn, "/admin-dashboard")
  assert html =~ "<title>2.50</title>"
  user3 = user_fixture(@create_user3_attrs)
  create_demographic(user3)
  create_rating(user3, product, 3)

  send(view.pid, %{event: "rating_created"})
  :timer.sleep(2)
```

Notice that we've added a sleep to give the live view time to receive the message, handle it, and re-render before executing any assertions.

We've sent the message, so all that remains is checking the result.

Verify the Result

To view the result, we'll call render. Then, we'll execute an assertion, like this:

```

testing/pento/test/pento_web/live/admin_dashboard_live_test.exs
test "it updates to display newly created ratings",
  %{conn: conn, product: product} do
  {:ok, view, html} = live(conn, "/admin-dashboard")
  assert html =~ "<title>2.50</title>"
  user3 = user_fixture(@create_user3_attrs)
  create_demographic(user3)
  create_rating(user3, product, 3)

  send(view.pid, %{event: "rating_created"})
  :timer.sleep(2)
  assert render(view) =~ "<title>2.67</title>"

```

We render the view, and then execute the assertion that verifies the updated template. It's finally time to run this last test.

Let it fly:

```

[pento] → mix test test/pento_web/live/admin_dashboard_live_test.exs
..
Finished in 0.4 seconds
2 tests, 0 failures
Randomized with seed 678757

```

We've tested a distributed operation, and then verified the result. With that, you've seen a lot of what live view tests can do. Before we go, we'll give you a chance to get your hands dirty.

Your Turn

LiveView makes it easy to write both unit tests and integration tests. Unit tests call individual functions within a live view in *isolation*. Integration tests exercise interactions *between* functions. Both are important, and LiveView's design makes it easy to do both.

Using the CRC pattern within a live view yields many single-purpose functions that are great testing targets. Unit tests use reducers to set up precise test conditions, and then compare those results against expectations in an assertion. Integration tests use the LiveViewTest module to mount and render a view. Then, these tests interact with elements on a page through the specialized functions provided by LiveViewTest to verify behavior with assertions.

We only saw a handful of LiveView test features in this chapter, but you're already equipped to write more.

Give It a Try

These tasks will give you a chance to explore unit and integration tests in the context of components.

- Build a unit test that calls `render_component/3`³ directly. Test that the stateless `RatingLive.IndexComponent` renders the product rating form when no product rating exists.
- Write another test to verify that the component renders the correct rating details when ratings do exist.
- Test the stateful `DemographicLive.FormComponent` by writing a test for the parent live view. Ensure that submitting a new demographic form updates the page to display the saved demographic details.

Next Time

This chapter completes our brief tour of testing, and closes out Part 3, *Extending LiveView*. In the next part, you'll get to create a new LiveView feature without relying on the help of any generators. We'll build a game to show how a multi-layer system interacts across multiple views, starting with a core layer that plots, rotates, and moves points.

3. https://hexdocs.pm/phoenix_live_view/Phoenix.LiveViewTest.html#render_component/3

Part IV

Graphics and Custom Code Organization

In Part IV, we'll focus on organizing custom code built from scratch. We'll start with a chapter to provide a detailed look at CRC in Elixir, including a core layer for moving and dropping game pieces. Next, we'll work with a LiveView layer that will render graphics, and finally a boundary layer that handles uncertainty.

Build The Game Core

By now, you have all of the building blocks you need to build clean, maintainable LiveView applications that handle a wide variety of use-cases. In this part of this book, you'll put together what you've learned to build an interactive in-browser game in LiveView, from the ground up.

LiveView might not be the perfect fit for complex in-browser games with lots of interaction or low latency environments. For such scenarios, it's best to write the full game in a client side language like JavaScript. But it *is* a good fit for the simple logic game we'll be building if you're not particularly concerned with latency. With LiveView, we can use the CRC pattern to model a game-like domain and present the user with a way to manage changes to that domain. Building a game will give you an opportunity to put into practice just about everything you've learned so far. It's the perfect way to wrap up your adventures with LiveView.

In this chapter, you'll start with our game's functional core, and you'll use the CRC pattern to model the game's basic pieces and interactions.

Remember, the functional core of your application represents certainty. It's the place for all of your pure, predictable code that always behaves the same way, given the same inputs. We'll construct our core out of small, pure functions that will compose into elegant pipelines. These functions will be single-purpose, so they'll be easy to test. And their composable nature makes them flexible—we'll call on the same functions in different orders to construct pipelines that do different things. This will allow us to layer up complex game functionality out of a few simple building blocks.

Before we dive into the details of these building blocks, let's make a plan for our functional core, and talk a bit about the game we'll be building.

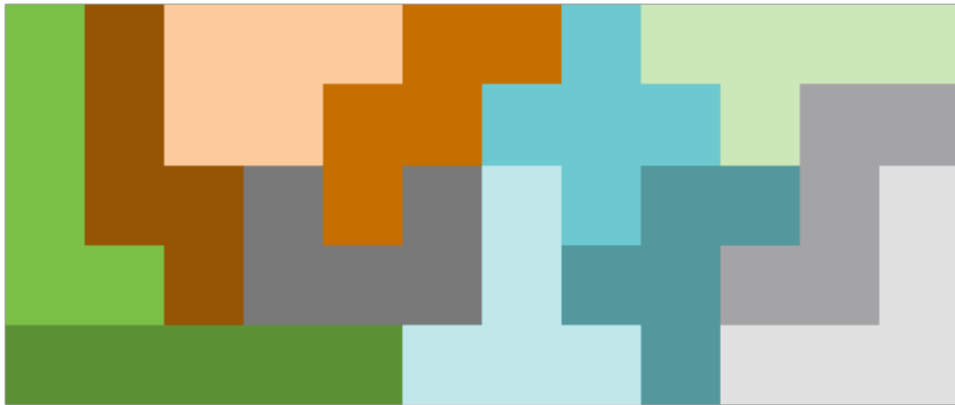
The Plan

In this last part of the book, we'll be building the game of Pentominoes—the favorite game of legendary CBS News anchor Walter Kronkite. Pentominoes is something like a cross between Tetris and a puzzle. The player is presented with a set of shapes called “pentominoes” and a game board of a certain size. The player must figure out how to place all of the pentominoes on the board so that they fit together to evenly cover all of the available space, like a puzzle.

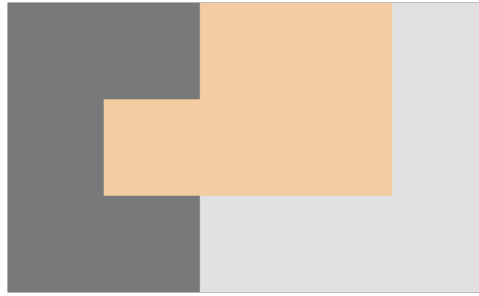
Each pentomino is a shape comprised of five even squares joined together. There are 12 basic shapes:



Assuming you're playing a round of Pentominoes with a large rectangular board and *all* of the available pentimino shapes, you might end up with a finished puzzle that looks like this:



Now, let's say you were playing a round of Pentominoes with a small rectangular board and just three shapes—a :c, a :v, and a :p. You might end up with a finished puzzle that looks like this:



Now that you have a basic idea of how the game works, let's talk about the core concepts of the game that we'll model in code.

The Game Board

Each round of pentominoes solves a different puzzle. This means that each round presents the player with a board of a different size and a new set of pentomino pieces to place.

We'll model the game board with a module that produces Board structs. Each board struct will have the following attributes:

- `points`: All of our puzzles will be rectangles of different shapes. The puzzle shape will be a list of points that make up the grid of our puzzle board.
- `palette`: The set of pentomino shapes that must be placed onto the board in order to complete the puzzle.
- `completed_pentos`: The pentomino shapes that have already been placed on the board. This will update as the user places more shapes.
- `active_pentos`: The pentomino from the palette that the user has selected and is actively in the process of placing on the board.

We'll return to the Board module and its functionality later on this chapter. For now, let's move on to a high-level overview of our next game fundamental, the pentominoes pieces themselves.

The Pentominoes Pieces

We'll define a module, `Pentomino`, that produces pentominoes structs. Each pentomino struct will have the following attributes:

- `name`: The type of shape, for example `:i` or `:p`.
- `rotation`: The number of degrees that the shape has been rotated to.
- `reflected`: A true/false value to indicate whether the user flipped the shape over to place it on the board.
- `location`: The location of the shape on the board grid.

Along with the Pentomino module that represents the placement of a shape on the board, we'll also define another module, Shape, that wraps up the attributes of individual pentominoes shapes. If Pentomino is responsible for representing a shape on the board, Shape is responsible for modeling a given pentomino shape. Each shape struct will have the following attributes:

- points: The five points that comprise the given shape.
- color: The color associated with the given shape.

We'll take a closer look at the relationship between a Pentomino struct and a Shape struct later on, and we'll see how to use them to model the placement of pentominoes on the game board. Before we move on however, we have one more game primitive to discuss.

The Pentominoes Points

Each shape is comprised of a set of five points of equal size, and each point will occupy a spot on the board grid. We'll model these points with the Point module. It will produce $\{x,y\}$ point tuples that represent the x, y coordinates of the point on the game board. It will also implement a set of reducer functions to move these points according to the location, rotation, and reflection of the shape to which a point belongs. In order to place our pentominoes on the board, we will apply functions to rotate, reflect, and move that pentomino to all five individual points that make up the pentomino. If that seems a little confusing right now, don't worry. We'll build this out step by step in the following sections.

Now that we have a high-level understanding of the basic building blocks of our game, let's start building the functional core in earnest. Each of the modules we've outlined so far will be implemented in our application's core, and we'll use the CRC pattern to build them. Our core modules will have constructors that create and return the module's data type, reducers to manipulate that data, and converters to transform the data into something that can be consumed by other parts of our application. While not all of our core modules will implement each of these CRC elements, you will see all of them come into play in the rest of this chapter.

Represent a Shape With Points

The user interface will render each pentomino with a set of points. Before we draw our points on the board, we need to understand *where* we will be placing those points. In order to be able to correctly calculate the location of each

point in a shape, given that shape's reflection and rotation, we're going to take the following approach.

- Always plot each shape in the center of a 5x5 grid that will occupy the top-left of any given game board.
- Calculate the location of each point in the shape given its rotation and reflection *within* that 5x5 square.
- Only *then* will we apply the location to move the pentomino's location on the wider board.

We'll dig into this process and the reasoning behind it in greater detail later on. For now, you just need to understand that every shape is comprised of a set of five points, and those five points are located by default in the center of 5x5 square which is positioned like this:

		X									
		1	2	3	4	5	6	7	8	9	10
Y	1	1, 1	2, 1	3, 1	4, 1	5, 1	6, 1	7, 1	8, 1	9, 1	10, 1
	2	1, 2	2, 2	3, 2	4, 2	5, 2	6, 2	7, 2	8, 2	9, 2	10, 2
	3	1, 3	2, 3	3, 3	4, 3	5, 3	6, 3	7, 3	8, 3	9, 3	10, 3
	4	1, 4	2, 4	3, 4	4, 4	5, 4	6, 4	7, 4	8, 4	9, 4	10, 4
	5	1, 5	2, 5	3, 5	4, 5	5, 5	6, 5	7, 5	8, 5	9, 5	10, 5
	6	1, 6	2, 6	3, 6	4, 6	5, 6	6, 6	7, 6	8, 6	9, 6	10, 6
	7	1, 7	2, 7	3, 7	4, 7	5, 7	6, 7	7, 7	8, 7	9, 7	10, 7
	8	1, 8	2, 8	3, 8	4, 8	5, 8	6, 8	7, 8	8, 8	9, 8	10, 8
	9	1, 9	2, 9	3, 9	4, 9	5, 9	6, 9	7, 9	8, 9	9, 9	10, 9
	10	1, 10	2, 10	3, 10	4, 10	5, 10	6, 10	7, 10	8, 10	9, 10	10, 10

Let's do some quick prototyping. Consider the :p shape. If we want to place the :p shape in the center of the 5x5 square, it will look like this:

		X									
		1	2	3	4	5	6	7	8	9	10
1		1, 1	2, 1	3, 1	4, 1	5, 1	6, 1	7, 1	8, 1	9, 1	10, 1
2		1, 2	2, 2	3, 2	4, 2	5, 2	6, 2	7, 2	8, 2	9, 2	10, 2
3		1, 3	2, 3	3, 3	4, 3	5, 3	6, 3	7, 3	8, 3	9, 3	10, 3
4		1, 4	2, 4	3, 4	4, 4	5, 4	6, 4	7, 4	8, 4	9, 4	10, 4
Y 5		1, 5	2, 5	3, 5	4, 5	5, 5	6, 5	7, 5	8, 5	9, 5	10, 5
6		1, 6	2, 6	3, 6	4, 6	5, 6	6, 6	7, 6	8, 6	9, 6	10, 6
7		1, 7	2, 7	3, 7	4, 7	5, 7	6, 7	7, 7	8, 7	9, 7	10, 7
8		1, 8	2, 8	3, 8	4, 8	5, 8	6, 8	7, 8	8, 8	9, 8	10, 8
9		1, 9	2, 9	3, 9	4, 9	5, 9	6, 9	7, 9	8, 9	9, 9	10, 9
10		1, 10	2, 10	3, 10	4, 10	5, 10	6, 10	7, 10	8, 10	9, 10	10, 10

So, the :p shape can be represented by a list of the following points:

```
[{3, 2}, {4, 2},
 {3, 3}, {4, 3},
 {3, 4}]
```

Now that you've seen how a set of points is used to depict a shape on the board, let's build out our very first core module, the Point module.

Define the Point Constructor

Create a file, `lib/pento/game/point.ex` and implement the module head like this:

```
core/pento/lib/pento/game/point.ex
defmodule Pento.Game.Point do
```

Now, define the constructor function. The core entity that the Point module creates and manages is the point tuple. The first element of the tuple is the x coordinate of the point and the second value is the y coordinate. So, our constructor function will take in two arguments, the x and y values of the point, and return the point tuple, as you can see here:

```
core/pento/lib/pento/game/point.ex
def new(x, y) when is_integer(x) and is_integer(y), do: {x, y}
```

Simple enough. The guards make sure each point has valid data, as long as we create points with the new constructor. If bad data comes in, we just let it crash because we can't do anything about that error condition. Now that we have a constructor to create points, let's build some reducers to manipulate those points.

Move Points Up and Down with Reducers

Later, we'll build logic that moves the pentomino shape on the board by applying a change of location to each point in that shape. Right now, we'll start from the ground up by giving our Point module the ability to move an individual point by some amount.

In the Point module, define the move/2 function like this:

```
core/pento/lib/pento/game/point.ex
def move({x, y}, {change_x, change_y}) do
  {x + change_x, y + change_y}
end
```

Here we have a classic reducer. It takes in a first argument of the data type we are manipulating, and a second argument of some input with which to manipulate it. Then, it returns a new entity that is the result of applying the manipulation to the data. In this case, we take in an amount by which to move each x and y coordinate, and return a new tuple that is the result of adding those values to the current x and y coords.

You can already see how easy it will be to create pipelines of movements that change the location of a given point. Since the move/2 reducer takes in a point tuple and returns a point tuple, we can string calls to move/2 together into flexible pipelines.

Tack on an end on your module and let's see it in action. Open up an IEx session and key this in to create and move a point:

```
iex> alias Pento.Game.Point
Pento.Game.Point
iex> Point.new(2, 2) |> Point.move({1, 0})
{3, 2}
```

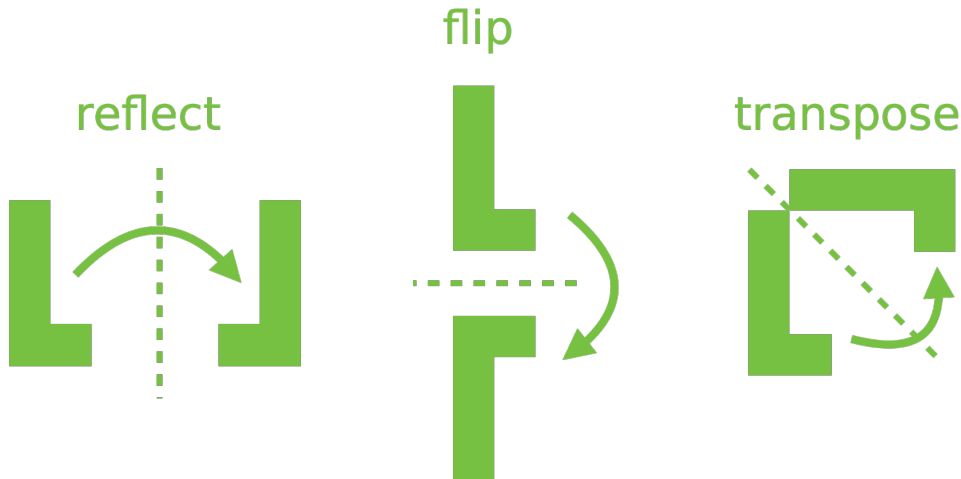
Then, try moving a point through a pipeline of movements:

```
iex> Point.new(2, 2) |> Point.move({1, 0}) |> Point.move({0, 1})
{3, 3}
```

With this reducer in place, we will be able to iterate over the points in a shape and move each point by some amount. In this way, we will move an entire shape according to user input.

Move Points Geometrically with Reducers

Now we have the ability to move points up and down by changing a point's x and y value with the `move/2` reducer. That's not enough, though. Users can rotate shapes or flip them over to fit them into the board and solve the puzzle. Here are some of the basic geometric manipulations our users will need to do:



Applying a change in position to a shape means applying that change to each point in the shape. Once again, we'll start from the ground up by giving our `Point` module the ability to make each type of change. The `reflect`, `flip`, and `transpose` functions will serve as primitives that allow us to do more complex operations like rotations. Later, we'll see this code in action in the context of manipulating the overall shape.

We'll start with the “transpose” movement. Think of transpose as flipping a point over a diagonal line that runs from the upper right to the lower left. In order to transpose the orientation of a shape on the board, we'll need to swap the x and y coordinates of each point in the shape. We'll define a reducer in the `Point` module that takes in a point tuple and returns a new tuple with the x and y values switched.

Open up the `Point` module and define a `transpose/1` function that looks like this:

```
core/pento/lib/pento/game/point.ex
```

```
def transpose({x, y}), do: {y, x}

def rotate(point, 0), do: point
def rotate(point, 90), do: point |> reflect |> transpose
def rotate(point, 180), do: point |> reflect |> flip
def rotate(point, 270), do: point |> flip |> transpose
```

```

def center(point), do: move(point, {-3, -3})

def prepare(point, rotation, reflected, location) do
  point
  |> rotate(rotation)
  |> maybe_reflect(reflected)
  |> move(location)
  |> center
end
end

```

Now let's turn our attention to the “flip” movement. In order to flip the orientation of a shape on the board, we will need to apply the following transformation to each point in the shape:

- Leave the x coordinate alone
- Subtract the value of the y coordinate from 6.

Here is where our approach of plotting each shape within an initial 5x5 grid comes into play. We take this approach so that we always know how to apply the “flip” (and later the “reflect”) transformation on a given shape. If we know that each shape is centered in a 5x5 grid, then we know that flipping it means applying this transformation to each point:

```
{x, 6 - y}
```

This makes it easy to build a single-purpose reducer that knows how to flip a point. By first applying any and all transpose, flip, or reflect transformations to all of the points in a shape centered in a 5x5 grid, we are able to calculate the correct location of each point in accordance with its orientation. Only then can we place the shape (and each of its points) in a provided location on the wider board. We'll take a closer look at this process of manipulating the overall shape and locating it on the board later on. For now, understand that starting with a 5x5 grid let's us define flip/1 and reflect/1 reducers that will always correctly place points according to the orientation of the given shape.

With this in mind, you can define a flip/1 reducer function that takes in a first argument of a point tuple, and returns a new tuple like this:

```

core/pento/lib/pento/game/point.ex
def flip({x, y}), do: {x, 6-y}

def transpose({x, y}), do: {y, x}

def rotate(point, 0), do: point
def rotate(point, 90), do: point |> reflect |> transpose
def rotate(point, 180), do: point |> reflect |> flip
def rotate(point, 270), do: point |> flip |> transpose

def center(point), do: move(point, {-3, -3})

```

```

def prepare(point, rotation, reflected, location) do
  point
  |> rotate(rotation)
  |> maybe_reflect(reflected)
  |> move(location)
  |> center
end
end

```

Now for our last geometric manipulation—the “reflect” movement. Reflecting a shape means applying this transformation to each point:

$\{6-x, y\}$

Go ahead and define a `reflect/1` reducer that takes in an argument of a point tuple and returns a new tuple, like this:

```

core/pento/lib/pento/game/point.ex
def reflect({x, y}), do: {6-x, y}

```

Let’s look at one example of applying one of these reducers to each point in a shape in order to better understand how we will use our reducers to manipulate shapes on the board. Let’s say you have our `:p` shape from the earlier example placed in the center of the 5x5 grid, like this:

		X									
		1	2	3	4	5	6	7	8	9	10
Y	1	1, 1	2, 1	3, 1	4, 1	5, 1	6, 1	7, 1	8, 1	9, 1	10, 1
	2	1, 2	2, 2	3, 2	4, 2	5, 2	6, 2	7, 2	8, 2	9, 2	10, 2
	3	1, 3	2, 3	3, 3	4, 3	5, 3	6, 3	7, 3	8, 3	9, 3	10, 3
	4	1, 4	2, 4	3, 4	4, 4	5, 4	6, 4	7, 4	8, 4	9, 4	10, 4
	5	1, 5	2, 5	3, 5	4, 5	5, 5	6, 5	7, 5	8, 5	9, 5	10, 5
	6	1, 6	2, 6	3, 6	4, 6	5, 6	6, 6	7, 6	8, 6	9, 6	10, 6
	7	1, 7	2, 7	3, 7	4, 7	5, 7	6, 7	7, 7	8, 7	9, 7	10, 7
	8	1, 8	2, 8	3, 8	4, 8	5, 8	6, 8	7, 8	8, 8	9, 8	10, 8
	9	1, 9	2, 9	3, 9	4, 9	5, 9	6, 9	7, 9	8, 9	9, 9	10, 9
	10	1, 10	2, 10	3, 10	4, 10	5, 10	6, 10	7, 10	8, 10	9, 10	10, 10

This shape is made up of the following points:


```
points = [{3, 2}, {4,2}, {3, 3}, {4, 3}, {3, 4}]
```

So, if we iterate over this list of points and apply the `Point.reflect/1` reducer to each one, we end up with the resulting list:

```
points = [{3, 2}, {2,2}, {3, 3}, {2, 3}, {3, 4}]
```

Mapped onto the board, we see this:

		X									
		1	2	3	4	5	6	7	8	9	10
Y	1	1, 1	2, 1	3, 1	4, 1	5, 1	6, 1	7, 1	8, 1	9, 1	10, 1
	2	1, 2	2, 2	3, 2	4, 2	5, 2	6, 2	7, 2	8, 2	9, 2	10, 2
	3	1, 3	2, 3	3, 3	4, 3	5, 3	6, 3	7, 3	8, 3	9, 3	10, 3
	4	1, 4	2, 4	3, 4	4, 4	5, 4	6, 4	7, 4	8, 4	9, 4	10, 4
	5	1, 5	2, 5	3, 5	4, 5	5, 5	6, 5	7, 5	8, 5	9, 5	10, 5
	6	1, 6	2, 6	3, 6	4, 6	5, 6	6, 6	7, 6	8, 6	9, 6	10, 6
	7	1, 7	2, 7	3, 7	4, 7	5, 7	6, 7	7, 7	8, 7	9, 7	10, 7
	8	1, 8	2, 8	3, 8	4, 8	5, 8	6, 8	7, 8	8, 8	9, 8	10, 8
	9	1, 9	2, 9	3, 9	4, 9	5, 9	6, 9	7, 9	8, 9	9, 9	10, 9
	10	1, 10	2, 10	3, 10	4, 10	5, 10	6, 10	7, 10	8, 10	9, 10	10, 10

By applying our point reducer transformations to each point in a shape, we will move and orient the shape according to input from the user.

Before we move on, let's do a little more exploration of the code we've built so far. The beautiful thing about our reducers is that we can string them into any combination of pipelines in order to transform points. Open up `IEx`, alias `Pento.Game.Point` and try out some of these pipelines:

```
iex> Point.new(2, 2) |> Point.move({1, 0})
{3, 2}
iex> Point.new(1, 1) |> Point.reflect
{5, 1}
iex> Point.new(1, 1) |> Point.flip
{1, 5}
iex> Point.new(1, 1) |> Point.flip |> Point.transpose
{5, 1}
```

Next up, we'll use these point movement reducers to create the “rotate point” flow.

Combine Reducers to Rotate Points

We've built out a set of point movement primitives that we'll use to manipulate shapes on the Pentominoes board. But users won't directly provide “flip” or “transpose” input. Instead, they will rotate a shape in increments of ninety degrees. So, we'll define a set of rotate/2 reducers that apply the correct pipeline of flip/1 and transpose/1 transformations in accordance with the given degrees of rotation.

Rotating shapes in increments of ninety degrees means that we will need to apply the following transformations:

- Rotate zero degrees: Do nothing.
- Rotate ninety degrees: Reflect and then transpose.
- Rotate one-hundred and eighty degrees: Reflect and then flip.
- Rotate two-hundred and seventy degrees: Flip and then transpose.

Thanks to the composable nature of our reducer functions, building a set of rotate/2 functions that pattern matches to each of these rotation values and pipes the given point into the correct set of reducers should be easy. Open up the Point module and define these functions:

```
core/pento/lib/pento/game/point.ex
def rotate(point, 0), do: point
def rotate(point, 90), do: point |> reflect |> transpose
def rotate(point, 180), do: point |> reflect |> flip
def rotate(point, 270), do: point |> flip |> transpose
```

Applying these rotate/2 functions to an :I shape, for example, should give you something like this:



Rotate 0



Rotate 90



Rotate 180



Rotate 270

Let's walk through what happens if we pipe each of the points in our original :p shape through the `rotate/2` function with a degrees argument of 90. Open up IEx and type this in:

```
iex> alias Pento.Game.Point
iex> points = [{3, 2}, {4,2}, {3, 3}, {4, 3}, {3, 4}]
iex> Enum.map(points, &Point.rotate(&1, 90))
[{2, 3}, {2, 2}, {3, 3}, {3, 2}, {4, 3}]
```

Let's break this down one step at a time. Calling `rotate/2` with a second argument of 90 invokes the following pipeline under the hood:

```
point |> reflect |> transpose
```

Taking this one step at a time, calling `reflect/1` on each point in the :p points applies the {6-x, y} transformation to each point in the list, returning this:

```
iex> Enum.map(points, &Point.reflect(&1))
[{3, 2}, {2, 2}, {3, 3}, {2, 3}, {3, 4}]
```

Then, calling the transpose/1 reducer with each point in this resulting list swaps each point's x and y values, giving us this:

```
iex> reflected_points = [{3, 2}, {2, 2}, {3, 3}, {2, 3}, {3, 4}]
iex> Enum.map(reflected_points, &Point.transpose(&1))
[{2, 3}, {2, 2}, {3, 3}, {3, 2}, {4, 3}]
```

Putting it all together, calling the rotate/2 reducer with our :p points and an argument of 90 degrees moves the shape to this new location on the board:

		X									
		1	2	3	4	5	6	7	8	9	10
Y	1	1, 1	2, 1	3, 1	4, 1	5, 1	6, 1	7, 1	8, 1	9, 1	10, 1
	2	1, 2	2, 2	3, 2	4, 2	5, 2	6, 2	7, 2	8, 2	9, 2	10, 2
	3	1, 3	2, 3	3, 3	4, 3	5, 3	6, 3	7, 3	8, 3	9, 3	10, 3
	4	1, 4	2, 4	3, 4	4, 4	5, 4	6, 4	7, 4	8, 4	9, 4	10, 4
	5	1, 5	2, 5	3, 5	4, 5	5, 5	6, 5	7, 5	8, 5	9, 5	10, 5
	6	1, 6	2, 6	3, 6	4, 6	5, 6	6, 6	7, 6	8, 6	9, 6	10, 6
	7	1, 7	2, 7	3, 7	4, 7	5, 7	6, 7	7, 7	8, 7	9, 7	10, 7
	8	1, 8	2, 8	3, 8	4, 8	5, 8	6, 8	7, 8	8, 8	9, 8	10, 8
	9	1, 9	2, 9	3, 9	4, 9	5, 9	6, 9	7, 9	8, 9	9, 9	10, 9
	10	1, 10	2, 10	3, 10	4, 10	5, 10	6, 10	7, 10	8, 10	9, 10	10, 10

By applying our rotate/2 reducer to each point in the :p shape, we are moving the shape around within the 5x5 grid. This ensures that we are correctly orienting the shape in the known area of the 5x5 grid. From there, we can place the shape in a given location on the wider board. Let's move on to that task now.

Prepare a Point For Rendering

Let's continue using our :p shape as an example. Imagine we have a :p shape that starts out centered in the middle of the 5x5 square with the default points values:

```
iex> [{3, 2}, {4, 2}, {3, 3}, {4, 3}, {3, 4}]
```

Let's apply the following pipeline of transformations to each point in the list:

```
iex> [{3, 2}, {4, 2}, {3, 3}, {4, 3}, {3, 4}] \
```

```
|> Enum.map(&Point.rotate(&1, 90)) \  
|> Enum.map(&Point.reflect(&1))  
[{4, 3}, {4, 2}, {3, 3}, {3, 2}, {2, 3}]
```

Breaking this down one step at a time, applying the rotate/2 reducer to each point in the shape, just like we did earlier, gives us this:

		X									
		1	2	3	4	5	6	7	8	9	10
Y	1	1, 1	2, 1	3, 1	4, 1	5, 1	6, 1	7, 1	8, 1	9, 1	10, 1
	2	1, 2	2, 2	3, 2	4, 2	5, 2	6, 2	7, 2	8, 2	9, 2	10, 2
	3	1, 3	2, 3	3, 3	4, 3	5, 3	6, 3	7, 3	8, 3	9, 3	10, 3
	4	1, 4	2, 4	3, 4	4, 4	5, 4	6, 4	7, 4	8, 4	9, 4	10, 4
	5	1, 5	2, 5	3, 5	4, 5	5, 5	6, 5	7, 5	8, 5	9, 5	10, 5
	6	1, 6	2, 6	3, 6	4, 6	5, 6	6, 6	7, 6	8, 6	9, 6	10, 6
	7	1, 7	2, 7	3, 7	4, 7	5, 7	6, 7	7, 7	8, 7	9, 7	10, 7
	8	1, 8	2, 8	3, 8	4, 8	5, 8	6, 8	7, 8	8, 8	9, 8	10, 8
	9	1, 9	2, 9	3, 9	4, 9	5, 9	6, 9	7, 9	8, 9	9, 9	10, 9
	10	1, 10	2, 10	3, 10	4, 10	5, 10	6, 10	7, 10	8, 10	9, 10	10, 10

Then, we call reflect/1 with each of these new points, performing the {6-x,y} transformation and returning this:

		X									
		1	2	3	4	5	6	7	8	9	10
Y	1	1, 1	2, 1	3, 1	4, 1	5, 1	6, 1	7, 1	8, 1	9, 1	10, 1
	2	1, 2	2, 2	3, 2	4, 2	5, 2	6, 2	7, 2	8, 2	9, 2	10, 2
	3	1, 3	2, 3	3, 3	4, 3	5, 3	6, 3	7, 3	8, 3	9, 3	10, 3
	4	1, 4	2, 4	3, 4	4, 4	5, 4	6, 4	7, 4	8, 4	9, 4	10, 4
	5	1, 5	2, 5	3, 5	4, 5	5, 5	6, 5	7, 5	8, 5	9, 5	10, 5
	6	1, 6	2, 6	3, 6	4, 6	5, 6	6, 6	7, 6	8, 6	9, 6	10, 6
	7	1, 7	2, 7	3, 7	4, 7	5, 7	6, 7	7, 7	8, 7	9, 7	10, 7
	8	1, 8	2, 8	3, 8	4, 8	5, 8	6, 8	7, 8	8, 8	9, 8	10, 8
	9	1, 9	2, 9	3, 9	4, 9	5, 9	6, 9	7, 9	8, 9	9, 9	10, 9
	10	1, 10	2, 10	3, 10	4, 10	5, 10	6, 10	7, 10	8, 10	9, 10	10, 10

Nothing new here so far. We've simply applied a pipeline of reducer transformations to each point in a shape. Now, let's say the provided location of the overall shape on the wider board is {5, 5}. All we need to do is take the updated list of points and apply the `Point.move/2` reducer to each one. Let's add to our pipeline now, like this:

```
iex> [{3, 2}, {4, 2}, {3, 3}, {4, 3}, {3, 4}] \
  |> Enum.map(&Point.rotate(&1, 90)) \
  |> Enum.map(&Point.reflect(&1)) \
  |> Enum.map(&Point.move(&1, {5, 5}))
[{9, 8}, {9, 7}, {8, 8}, {8, 7}, {7, 8}]
```

Recall that the `move/2` reducer takes in a first argument of a point and a second argument of `{x_change, y_change}`. It returns a new point tuple that is the result of adding the `x_change` value to the `x` coordinate and the `y_change` value to the `y` coordinate.

This leaves us with a board that looks like this:

		X									
		1	2	3	4	5	6	7	8	9	10
Y	1	1, 1	2, 1	3, 1	4, 1	5, 1	6, 1	7, 1	8, 1	9, 1	10, 1
	2	1, 2	2, 2	3, 2	4, 2	5, 2	6, 2	7, 2	8, 2	9, 2	10, 2
	3	1, 3	2, 3	3, 3	4, 3	5, 3	6, 3	7, 3	8, 3	9, 3	10, 3
	4	1, 4	2, 4	3, 4	4, 4	5, 4	6, 4	7, 4	8, 4	9, 4	10, 4
	5	1, 5	2, 5	3, 5	4, 5	5, 5	6, 5	7, 5	8, 5	9, 5	10, 5
	6	1, 6	2, 6	3, 6	4, 6	5, 6	6, 6	7, 6	8, 6	9, 6	10, 6
	7	1, 7	2, 7	3, 7	4, 7	5, 7	6, 7	7, 7	8, 7	9, 7	10, 7
	8	1, 8	2, 8	3, 8	4, 8	5, 8	6, 8	7, 8	8, 8	9, 8	10, 8
	9	1, 9	2, 9	3, 9	4, 9	5, 9	6, 9	7, 9	8, 9	9, 9	10, 9
	10	1, 10	2, 10	3, 10	4, 10	5, 10	6, 10	7, 10	8, 10	9, 10	10, 10

This calculation is a little off though. Remember that we placed our original :p shape in the center of the 5x5 grid at the top-left of our board, meaning the center of the :p shape occupied the {3,3} location on the board. We always take {3,3} as the starting point of the center of any shape we put on the board. This way, we can reliably apply the correct math to calculate the orientation of the shape given the provided rotation, reflection, etc. So, when we move our correctly orientated shape to its final location, we're actually off by 3. In order to ensure that the shape is moved to the given location, we need to take the results of applying move(point, {5, 5}) and subtract 3 from every point's coordinates. We'll call this action "centering the point" in order to account for the {3,3} offset we began with.

Let's implement a center/1 reducer to that for us now:

```
core/pento/lib/pento/game/point.ex
def center(point), do: move(point, {-3, -3})
```

Now, we'll add a call to center/1 to the end of our pipeline:

```
iex> [{3, 2}, {4, 2}, {3, 3}, {4, 3}, {3, 4}] \
  |> Enum.map(&Point.rotate(&1, 90)) \
  |> Enum.map(&Point.reflect(&1)) \
  |> Enum.map(&Point.move(&1, {5, 5})) \
  |> Enum.map(&Point.center(&1))
[{6, 5}, {6, 4}, {5, 5}, {5, 4}, {4, 5}]
```

This would leave us with a board that looks like this:

		X									
		1	2	3	4	5	6	7	8	9	10
Y	1	1, 1	2, 1	3, 1	4, 1	5, 1	6, 1	7, 1	8, 1	9, 1	10, 1
	2	1, 2	2, 2	3, 2	4, 2	5, 2	6, 2	7, 2	8, 2	9, 2	10, 2
	3	1, 3	2, 3	3, 3	4, 3	5, 3	6, 3	7, 3	8, 3	9, 3	10, 3
	4	1, 4	2, 4	3, 4	4, 4	5, 4	6, 4	7, 4	8, 4	9, 4	10, 4
	5	1, 5	2, 5	3, 5	4, 5	5, 5	6, 5	7, 5	8, 5	9, 5	10, 5
	6	1, 6	2, 6	3, 6	4, 6	5, 6	6, 6	7, 6	8, 6	9, 6	10, 6
	7	1, 7	2, 7	3, 7	4, 7	5, 7	6, 7	7, 7	8, 7	9, 7	10, 7
	8	1, 8	2, 8	3, 8	4, 8	5, 8	6, 8	7, 8	8, 8	9, 8	10, 8
	9	1, 9	2, 9	3, 9	4, 9	5, 9	6, 9	7, 9	8, 9	9, 9	10, 9
	10	1, 10	2, 10	3, 10	4, 10	5, 10	6, 10	7, 10	8, 10	9, 10	10, 10

Now, our :p shape is correctly orientated according to the given rotation and reflection, and it is correctly placed with its center in the provided location of {5, 5}.

So far, we've strung together our own bespoke pipeline of reducers by iterating over a list of points and calling various combinations of rotate/2, reflect/1 and move/2. Let's take a step back and think about how such a pipeline will be used in the context of placing a shape on the board.

We know that we will define a module, Shape, that produces structs with the attributes:

- points: The list of points the make up the shape
- rotation: The number of degrees the shape user has rotated the shape, between 0 and 270 in increments of 90
- reflected: A true or false value indicating if the user has reflected the shape
- location: The desired location of the shape on the board

We'll want a way to apply each of these attributes to a given point. We'll create a pipeline of reducers in a function, Point.prepare/4, to do exactly that. In lib/pento/game/point.ex, define this function:


```
core/pento/lib/pento/game/point.ex
def prepare(point, rotation, reflected, location) do
  point
  |> rotate(rotation)
  |> maybe_reflect(reflected)
  |> move(location)
  |> center
end
end
```

Note that we're using a new reducer function, `maybe_reflect/2`. Here's what that function looks like:

```
core/pento/lib/pento/game/point.ex
def maybe_reflect(point, true), do: reflect(point)
def maybe_reflect(point, false), do: point
```

The `maybe_reflect/2` reducer takes in a first argument of a point and a second argument of true or false. It calls the `reflect/1` reducer if the second argument is true and simply returns the unchanged point if it is false.

This `prepare/4` function is where the CRC pattern really shines. We can manipulate *one point* in the pentomino, according to the set of rules that we will later encapsulate in the `Shape` struct. Then, we can use the same `prepare/4` function to move *all of the points* according to the same rules.

Later we'll use `Point.prepare/4` to move all the points in a pentomino shape at once. Now, let's move on to building out that `Shape` module.

Group Points Together in Shapes

The `Shape` module is responsible for modeling pentomino shapes. It will wrap up the attributes that describe a shape and implement some functions that manage the behavior of shapes. In this section, we'll define the `Shape` module and implement its constructor function. We'll use the `Point.prepare/4` function we just built to prepare a shape for rendering in the correct location, given some input.

Represent Shape Attributes

Let's begin with the attributes that describe a shape. Each shape will have a distinct color and a list of points. The `Shape` module will wrap up these attributes in a struct. We'll build this out first. Create a file, `lib/pento/game/shape.ex` and define the module like this:

```
defmodule Pento.Game.Shape do
  defstruct color: :blue, shape: :x, points: []
end
```

Our module implements a call to `defstruct` to define what keys the struct will have, along with their default values. Now, if we open up `IEx`, we can create new `Shape` structs like this:

```
iex> alias Pento.Game.Shape
Pento.Game.Shape
iex> Shape.__struct__
%Pento.Game.Shape{color: :blue, points: [], shape: :x}
```

Great. Now we have a module that produces structs that represent shapes. Let's make our module a little smarter. We'll implement a set of functions that represent each shape's points and colors.

Add these functions to your module to represent the colors associated to each shape:

```
core/pento/lib/pento/game/shape.ex
defp color(:i), do: :dark_green
defp color(:l), do: :green
defp color(:y), do: :light_green
defp color(:n), do: :dark_orange
defp color(:p), do: :orange
defp color(:w), do: :light_orange
defp color(:u), do: :dark_gray
defp color(:v), do: :gray
defp color(:s), do: :light_gray
defp color(:f), do: :dark_blue
defp color(:x), do: :blue
defp color(:t), do: :light_blue
```

Then, add these functions to represent the list of points that make up each shape:

```
core/pento/lib/pento/game/shape.ex
defp points(:i), do: [{3, 1}, {3, 2}, {3, 3}, {3, 4}, {3, 5}]
defp points(:l), do: [{3, 1}, {3, 2}, {3, 3}, {3, 4}, {4, 4}]
defp points(:y), do: [{3, 1}, {2, 2}, {3, 2}, {3, 3}, {3, 4}]
defp points(:n), do: [{3, 1}, {3, 2}, {3, 3}, {4, 3}, {4, 4}]
defp points(:p), do: [{3, 2}, {4, 3}, {3, 3}, {4, 2}, {3, 4}]
defp points(:w), do: [{2, 2}, {2, 3}, {3, 3}, {3, 4}, {4, 4}]
defp points(:u), do: [{2, 2}, {4, 2}, {2, 3}, {3, 3}, {4, 3}]
defp points(:v), do: [{2, 2}, {2, 3}, {2, 4}, {3, 4}, {4, 4}]
defp points(:s), do: [{3, 2}, {4, 2}, {3, 3}, {2, 4}, {3, 4}]
defp points(:f), do: [{3, 2}, {4, 2}, {2, 3}, {3, 3}, {3, 4}]
defp points(:x), do: [{3, 2}, {2, 3}, {3, 3}, {4, 3}, {3, 4}]
defp points(:t), do: [{2, 2}, {3, 2}, {4, 2}, {3, 3}, {3, 4}]
```

Great. Our `Shape` module knows what each kind of shape looks like. Now we're ready to implement the constructor function.

Define the Shape Constructor

A shape will be constructed with a provided name, rotation, reflection, and location. The Shape constructor will use the name to select the correct set of points and the correct color for the shape. It will apply the rotation, reflection and location values to each point and return an updated points list that is correctly oriented and placed on the board.

This last part might sound challenging, but we already built out all of the code we need in the previous section. The `Point.prepare/4` function takes in a point, rotation, reflection, and location and does all the work of orienting, moving and centering a point on the board. All our constructor needs to do is iterate over the points that make up the shape and call `Point.prepare/4` with each one. This will return a new list of correctly updated points. Let's build it.

First, remember to alias the `Pento.Game.Point` module like this:

```
core/pento/lib/pento/game/shape.ex
defmodule Pento.Game.Shape do
  alias Pento.Game.Point
```

Then, implement the constructor as shown here:

```
core/pento/lib/pento/game/shape.ex
def new(name, rotation, reflected, location) do
  points =
    name
    |> points()
    |> Enum.map(&Point.prepare(&1, rotation, reflected, location))

  %__MODULE__{:points: points, :color: color(name), :name: name}
end
```

Go ahead and test out your new constructor function. Open up IEx and key this in:

```
iex> Pento.Game.Shape.new(:p, 90, true, {5, 5})
%Pento.Game.Shape{
  color: :orange,
  points: [{6, 5}, {5, 4}, {5, 5}, {6, 4}, {4, 5}],
  shape: :p
}
```

You can see how the layers of our functional core are starting to come together. Given some information about a pentomino—its name, rotation, reflection, and location—we generate a struct that represents the shape's color, with all of the points in the right place thanks to the reducer pipeline

in the `Point.prepare/4` function. This struct wraps up everything we'll need to render the shape on the board later on.

We need one more abstraction before we're ready to build our board.

Track and Place a Pentomino

The `Shape` constructor takes in a rotation, reflection, and location value and returns the struct that represents the orientation and location of that shape on the board. This rotation, reflection, and location will be the result of a user selecting a shape and moving it around the board before finally dropping it into place. A user may do any combination of rotating the shape, reflecting the shape, or moving it up and down on the board before deciding to place it. So, we need a way to track all of these user inputs in one place before feeding the final rotation, reflection, and location values into our `Shape` constructor. We'll implement the `Pentomino` module to do exactly that.

Like the `Shape` module, the `Pentomino` module will produce structs that know the shape name, rotation, reflection, and location. Most importantly, however, the `Pentomino` module will implement a series of reducers that we can string together given a set of user inputs to change the rotation, reflection, and location of the pentomino. Then, the pentomino will be converted into a shape in order to be placed on the board at the correct set of points.

If you're thinking that this process—creating a `Pentomino` struct, transforming it with a series of reducers, and then converting it into a shape that can be displayed to the user—sounds a lot like CRC, you'd be right!

Let's implement our new module.

Represent Pentomino Attributes

Create a new file, `lib/pento/game/pentomino.ex` and define a module that looks like this:

```
defmodule Pento.Game.Pentomino do
  @names [:i, :l, :y, :n, :p, :w, :u, :v, :s, :f, :x, :t]
  @default_location {8, 8}

  defstruct [
    name: :i,
    rotation: 0,
    reflected: false,
    location: @default_location
  ]
end
```

Great. Now if you load up IEx, you should be able to create a new pentomino with the default values, like this:

```
iex> alias Pento.Game.Pentomino
Pento.Game.Pentomino
iex> Pentomino.__struct__
%Pento.Game.Pentomino{
  location: {8, 8},
  name: :i,
  reflected: false,
  rotation: 0
}
```

Now we're ready to define the constructor.

Define the Pentomino Constructor

Our constructor is simple. All it needs to do is implement a function, `new/1`, that takes in an argument of a keyword list and uses it to return a new `Pentomino` struct. Go ahead and add this function in now:

```
core/pento/lib/pento/game/pentomino.ex
def new(fields \\ []), do: __struct__(fields)
```

Now, recompile your IEx session and practice creating a new pentomino, like this:

```
iex> recompile()
iex> Pentomino.new(location: {11,5}, name: :p, reflected: true, rotation: 270)
%Pento.Game.Pentomino{
  location: {11, 5},
  name: :p,
  reflected: true,
  rotation: 270
}
```

With the constructor in place, we can move on to the reducers.

Manipulate Pentominoes with Reducers

A pentomino struct will hold the state of the pentomino as we transform that state with input from the user. A user will be able to:

- Rotate the pentomino in increments of 90 degrees
- Flip the pentomino
- Move the pentomino up, down, left, or right one square at a time

Each of these interactions will update the pentomino's rotation, reflected, or location attributes respectively. When a user is done moving the pentomino and is ready to place it on the board, we will take the final values of each of

these attributes and use them to create a Shape struct that knows its correct points locations.

Let's start with the rotate/1 reducer. This reducer will take in an argument of a pentomino and update its rotation attribute in increments of 90 by performing the following calculation: `rem(degrees + 90, 360)`. This will ensure that we rotate the pentomino 90 degrees at a time, returning the value to 0 rather than exceeding 270. Open up the Pentomino module and add this in:

```
core/pento/lib/pento/game/pentomino.ex
def rotate(%{rotation: degrees}=p) do
  %{ p | rotation: rem(degrees + 90, 360)}
end
```

Here, we return a new pentomino struct with all of the original struct's values, along with the updated `:rotation` value.

Next up, implement the flip/1 reducer. This reducer will take in an argument of a pentomino and return a new struct with an updated `:reflected` value that is the opposite of the present value. So, if the pentomino is *not* flipped, flipping it will set `:reflected` to true. If it *is* flipped, flipping it *again* will set `:reflected` to false. Open up the Pentomino module and add this in:

```
core/pento/lib/pento/game/pentomino.ex
def flip(%{reflected: reflection}=p) do
  %{ p | reflected: not reflection}
end
```

Lastly, we'll implement a set of reducers to move the pentomino up, down, left, and right by one square at a time. In other words, moving a pentomino up should change its location by `{x, y-1}`, and so on.

A pentomino's location represents the x and y coordinates of the center point of the pentomino's shape. We already have a function that knows how to move a point—`Point.move/1`. We'll re-use it here. First, open up the Pentomino module and alias `Pento.Game.Point` at the top of the module. Then, add in these reducers:

```
core/pento/lib/pento/game/pentomino.ex
def up(p) do
  %{ p | location: Point.move(p.location, {0, -1})}
end

def down(p) do
  %{ p | location: Point.move(p.location, {0, 1})}
end

def left(p) do
  %{ p | location: Point.move(p.location, {-1, 0})}
end
```

```
def right(p) do
  %{ p | location: Point.move(p.location, {1, 0})}
end
```

That's it for our reducers. Now for the final step—creating the converter function.

Convert a Pentomino to a Shape

A pentomino knows its rotation, reflection, and location based on a set of user input. In order to place that pentomino on the board, we will convert it to a shape. Recall that creating a new shape struct with a call to the `Shape.new/4` constructor does a few things:

- Get the list of default points that make up the given shape.
- Iterate over that list of points and call `Point.prepare/4` to apply the provided rotation, reflection, and location to each point in the shape. Collect the newly updated list of properly oriented and located points.
- Return a shape struct that knows its name, color, and this updated list of points.

In this way, we convert a pentomino into a shape that can be placed on the board, with the correct orientation, in the correct location, with the correct color.

Let's build that constructor now. Open up the `Pentomino` module and add this in:

```
core/pento/lib/pento/game/pentomino.ex
def to_shape(pento) do
  Shape.new(pento.name, pento.rotation, pento.reflected, pento.location)
end
end
```

Now, it's time to test drive it.

Test Drive The Pentomino CRC Pipeline

Let's run through a few examples of the `Pentomino CRC` pipeline. We'll create a new pentomino struct with the constructor, apply some transformations with various combinations of reducers, and convert the pentomino into a shape that knows its points and can be placed on a board.

Open up `IEx` and try this out:

```
iex> Pentomino.new(name: :i) |> Pentomino.rotate |> Pentomino.rotate
%Pento.Game.Pentomino{
  location: {8, 8},
```

```

    name: :i,
    reflected: false,
    rotation: 180
  }

```

We’ve constructed a new pentomino and rotated it twice. Just like with our previous reducer pipelines, we can string together any combination of reducers in order to change the state of our entity. By calling `Pentomino.rotate/1` twice, we first update the default rotation from 0 to 90 and then from 90 to 180.

Let’s keep adding to our reducer pipeline, as shown here:

```

iex> Pentomino.new(name: :i) \
    |> Pentomino.rotate \
    |> Pentomino.rotate \
    |> Pentomino.down
%Pento.Game.Pentomino{
  location: {8, 9},
  name: :i,
  reflected: false,
  rotation: 180
}

```

This time, we add an additional step to our reducer pipeline, the call to `Pentomino.down/1`. This changes the location by applying the transformation: `{x, y+1}`.

Finally, let’s convert our pentomino to a shape, like this:

```

iex> Pentomino.new(name: :i) \
    |> Pentomino.rotate \
    |> Pentomino.rotate \
    |> Pentomino.down \
    |> Pentomino.to_shape
%Pento.Game.Shape{
  color: :dark_green,
  points: [{8, 11}, {8, 10}, {8, 9}, {8, 8}, {8, 7}],
  shape: :i
}

```

We end up with a shape that has a correctly populated `:points` value given the rotation, reflection, and location of the transformed pentomino.

Now that we can create pentominoes, move them, and convert them to shapes that can be placed on a board, it’s time to build out the last piece of our functional core—the game board.

Track a Game in a Board

The Board module will be responsible for tracking a game. It will produce structs that describe the attributes of a board and implement a constructor function for creating new boards when a user starts a game of Pentominoes. Later, it will implement reducers to manage the behavior of a board, including commands to select a pento to move, rotate, or reflect a pento, drop a piece into place, and more.

Represent Board Attributes

A board struct will have the following attributes:

- `points`: The points that make up the shape of the empty board that the user will fill up with pentominoes. All of our shapes will be rectangles of different sizes.
- `completed_pentos`: The list of pentominoes that the user has placed on the board.
- `palette`: The provided pentominoes that the user has available to solve the puzzle.
- `active_pento`: The currently selected pentomino that the user is moving around the board.

Let's start by defining our module, along with a `defstruct` to define the Board structs to have these attributes. Create a new file, `lib/pento/game/board.ex`, add this in and then close your module with an `end`:

```
core/pento/lib/pento/game/board.ex
defmodule Pento.Game.Board do
  defstruct [
    active_pento: nil,
    completed_pentos: [],
    palette: [],
    points: []
  ]
end
```

Next up, we'll diverge from the CRC pattern a bit and define a function that returns the list of puzzle shapes we will support:

```
core/pento/lib/pento/game/board.ex
def puzzles(), do: ~w[default wide widest medium tiny]a
```

This function will be called later on in LiveView when we generate a new game for a user to play.

Now we're ready to implement the constructor.

Define the Board Constructor

Define a constructor function, `new/2`, that takes in two arguments, an atom that indicates the palette size (how many pentominoes to give the user), and a list of points that will form the board grid. Here's a look at the `new/2` function:

```
core/pento/lib/pento/game/board.ex
def new(palette, points) do
  %__MODULE__{:palette: palette(palette), :points: points}
end

def new(:tiny), do: new(:small, rect(5, 3))
def new(:widest), do: new(:all, rect(20, 3))
def new(:wide), do: new(:all, rect(15, 4))
def new(:medium), do: new(:all, rect(12, 5))
def new(:default), do: new(:all, rect(10, 6))
```

Here, we define two functions called `new`. The `new/1` function will be called with a board size atom. Each `new/1` function calls the `new/2` constructor with a different palette size and list of board points depending on the provided board size.

Let's take a look at the `palette/1` and `rect/2` functions used by our constructors now.

Define Constructor Helper Functions

The `rect/2` function takes in a board width and height and uses a for loop to return the list of points that make up a rectangle of that size. Open up the `Board` module and add this in:

```
core/pento/lib/pento/game/board.ex
defp rect(x, y) do
  for x <- 1..x, y <- 1..y, do: {x, y}
end
```

Now, define two versions of a `palette/1` function that pattern match on the available `:all` or `:small` atoms to return a list of pentomino shapes, like this:

```
core/pento/lib/pento/game/board.ex
defp palette(:all), do: [:i, :l, :y, :n, :p, :w, :u, :v, :s, :f, :x, :t]
defp palette(:small), do: [:u, :v, :p]
end
```

These functions work together to produce the correct set of board points and list of pentomino shapes for a given board size. In this way, we can start to build up a game given some input describing the board. You can start to think about how a board's list set of coordinates and list of pentominoes could be

displayed to the user so that a user can select pentominoes, move them, and place them on the board. Let's talk about those interactions now.

Test Drive the Board Constructor

Open up IEx and create a tiny board, like this:

```
iex> alias Pento.Game.Board
Pento.Game.Board
iex> Board.new(:tiny)
%Pento.Game.Board{
  active_pento: nil,
  completed_pentos: [],
  palette: [:u, :v, :p],
  points: [{1, 1},{1, 2},{1, 3},{2, 1},{2, 2},{2, 3},...{5, 3}]
}
iex>
```

You can see that we've created a board struct with the correct list of points describing the board rectangle, the correct (small) list of pentominoes for the user to place, and the correct default values to active_pentos and completed_pentos.

Manipulate The Board with Reducers

We won't actually be building the Board reducers in this chapter. We'll take that on in the next chapter as we build out our game live view. But let's think through this problem a bit now.

Users will be able to do a few things with pentominoes on the board:

- choose: Pick an active pentomino from the palette to move around the board. This should update a board struct's active_pento attribute.
- drop: Place a pentomino in a location on the board. The should update the list of placed pentominoes in the completed_pentos attribute of a given board struct.

We'll also need to make Board smart enough to know if a pentomino *can* be dropped in a given location. We'll support these interactions with our boards:

- legal?: Returns true if the location a user wants to drop a pentomino in is in fact on the board.
- droppable?: Returns true if the location a user wants to drop a pentomino in is in fact unoccupied by another pieces.

Lastly, we'll want to give the Board module the ability to tell us if the game is over. We'll implement this behavior:

- `status`: Indicates if all of the pentominoes in the palette have been placed on the board. In other words, are all of the pentominoes in the palette listed in the `completed_pentos` list of placed pieces?

Your Turn

In this chapter, you built out a solid functional core for our new Pentominoes game feature. You layered together three pieces of our core functionality to represent pentomino pieces, manipulate their orientation and location, and convert them into something that can be placed on a pentomino game board.

The `Point` module represents points. It constructs `{x, y}` tuples and manipulates them with reducers. We strung these reducers together into a pipeline in the `Point.prepare/4` function.

The `Shape` module builds structs that have a color, name, and the list of points in that shape. It uses `Point.prepare/4` to apply these attributes to every point in the shape.

The `Pentomino` module tracks the transformations to a shape that a user will apply during game play. Its constructor produces structs with the shape name, rotation, reflection, and location attributes. Its reducers take in some user input and update these attributes. Its converter returns shapes with the correctly located set of points given these attributes.

We've covered a lot of complex logic in this chapter, and the CRC pattern made it all possible. By creating a set of modules that modeled the different elements of our game, we were able to break out the logical concerns of game play. By applying the CRC pattern to each of these game elements, we were able to neatly model the attributes and behaviors of the game. By layering these game elements, we assembled the near-complete behavior of a round of pentominoes.

Now, it's time for you to work with these concepts on your own.

Give It a Try

These exercises let you better understand the core by writing tests. You'll also have a chance to extend the core by adding some new kinds of boards, and creating a reducer to add an optional direction to turn a pentomino.

- Write tests for the core. Focus on writing unit tests for the individual reducer functions of our core modules. Maybe reach for the reducer testing pipelines we used in [Chapter 10, Test Your Live Views, on page 255](#) to keep your tests clean, flexible, and highly readable.

- Add a skewed rectangle¹ puzzle, and a function called `Board.skewed_rect/2`, like `Board.rect/2` to support it.
- Add an optional direction argument to `Pentomino.rotate/1` that allows both clockwise and counterclockwise rotation.

Up Next

We're going to depart a bit from the approach we've taken in building out features so far, and turn our attention the UI *before* we build the application boundary. With our functional core firmly in place, we've perfectly set the stage for building out the UI in LiveView. In the next chapter, we'll focus on rendering a board and shapes by composing a series of LiveView components with the help of the Surface library. We'll build *almost* the full game-play functionality, adding in new Board reducers along the way. Only then will we build the application boundary that will enforce some necessary game rules and validations on the UI.

1. <http://puzzler.sourceforge.net/docs/pentominoes.html>

Render Graphics With SVG

Content to be supplied later.

Establish Boundaries and APIs

Content to be supplied later.

Bibliography

- [Alm18] Ulisses Almeida. *Learn Functional Programming with Elixir*. The Pragmatic Bookshelf, Raleigh, NC, 2018.
- [Heb19] Fred Hebert. *Property-Based Testing with PropEr, Erlang, and Elixir*. The Pragmatic Bookshelf, Raleigh, NC, 2019.
- [IT19] James Edward Gray, II and Bruce A. Tate. *Designing Elixir Systems with OTP*. The Pragmatic Bookshelf, Raleigh, NC, 2019.
- [LM21] Andrea Leopardi and Jeffrey Matthias. *Testing Elixir*. The Pragmatic Bookshelf, Raleigh, NC, 2021.
- [McC15] Chris McCord. *Metaprogramming Elixir*. The Pragmatic Bookshelf, Raleigh, NC, 2015.
- [Tho18] Dave Thomas. *Programming Elixir 1.6*. The Pragmatic Bookshelf, Raleigh, NC, 2018.
- [TV19] Chris McCord, Bruce Tate and José Valim. *Programming Phoenix 1.4*. The Pragmatic Bookshelf, Raleigh, NC, 2019.
- [WM19] Darin Wilson and Eric Meadows-Jönsson. *Programming Ecto*. The Pragmatic Bookshelf, Raleigh, NC, 2019.

Thank you!

How did you enjoy this book? Please let us know. Take a moment and email us at support@pragprog.com with your feedback. Tell us your story and you could win free ebooks. Please use the subject line “Book Feedback.”

Ready for your next great Pragmatic Bookshelf book? Come on over to <https://pragprog.com> and use the coupon code BUYANOTHER2021 to save 30% on your next ebook.

Void where prohibited, restricted, or otherwise unwelcome. Do not use ebooks near water. If rash persists, see a doctor. Doesn't apply to *The Pragmatic Programmer* ebook because it's older than the Pragmatic Bookshelf itself. Side effects may include increased knowledge and skill, increased marketability, and deep satisfaction. Increase dosage regularly.

And thank you for your continued support.

The Pragmatic Bookshelf



SAVE 30%!
Use coupon code
BUYANOTHER2021

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by professional developers for professional developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

This Book's Home Page

<https://pragprog.com/book/liveview>

Source code from this book, errata, and other resources. Come give us feedback, too!

Keep Up to Date

<https://pragprog.com>

Join our announcement mailing list (low volume) or follow us on twitter @pragprog for new titles, sales, coupons, hot tips, and more.

New and Noteworthy

<https://pragprog.com/news>

Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this ebook, perhaps you'd like to have a paper copy of the book. Paperbacks are available from your local independent bookstore and wherever fine books are sold.

Contact Us

Online Orders: <https://pragprog.com/catalog>

Customer Service: support@pragprog.com

International Rights: translations@pragprog.com

Academic Use: academic@pragprog.com

Write for Us: <http://write-for-us.pragprog.com>

Or Call: +1 800-699-7764