The Pragmatic Programmers

grox.io
Learning

Programmer Passport

Phoenix LiveView

Bruce A. Tate

Edited by Jacquelyn Carter

Programmer Passport: Liveview

Bruce Tate

The Pragmatic Bookshelf
Raleigh, North Carolina

Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

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

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

ISBN-13: pending
Encoded using the finest acid-free high-entropy binary digits.
Book version: B1.0—Monthname, yyyy

Contents

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.

B1.0: July 15, 2020

- Initial beta release. This is our initial Phoenix with LiveView chapter. It marks a departure of traditional Programmer Passport programs. We'll be covering overarching concepts in these chapters, and covering more details in our videos. We think this strategy matches Groxio's teaching methods much more clearly. Let us know what you think! Drop us a line at info@grox.io.

CHAPTER 1

Lifecycle and Flow of a LiveView

LiveView is the most anticipated Elixir framework yet. The premise is seductive: you can build highly interactive, scalable applications without ever writing a line of custom JavaScript.

In this book, we'll look at how LiveView works in broad strokes, and then we'll examine how data moves between the client and server. This book will be heavily supplemented by the videos on Groxio. We'll use this book to explore the *theory*. We'll explore high-level concepts, with a few simple examples. You'll see how Phoenix generates code, learn strategies for designing your own code, and find out how to incorporate many different kinds of events.

In the videos, we'll explore detailed examples. You'll find out how to do common tasks. We'll start with the most basic applications so you can work with LiveView without having to focus too much on business logic. We'll integrate change validation through Ecto changesets without requiring a database. We'll graduate to a more complex application for memorizing things.

In both the book and videos, we'll organize our code into layers so you can address only a small bit of complexity at any given time. As you read now, we're going to stay at a high level. We'll walk through a basic LiveView program, starting at `mix phx.new` and going all the way through trying our working application in the development environment.

In this chapter, we'll concentrate on an overview of LiveView. We'll talk about the problem it solves, walk through how new requests work, and then explore how to make these programs interactive.

Let's get started. In order to best understand the problem LiveView solves, it's best to understand a bit of history. Let's look at how web programming was traditionally done.

The Evolution of LiveView

In a traditional web app, programmers wrote code for a web application server to take web requests. The server would process a request, perhaps reading from a database or other back end service, and then build a new web page based on the response. This model of web app development is *request-response* processing.

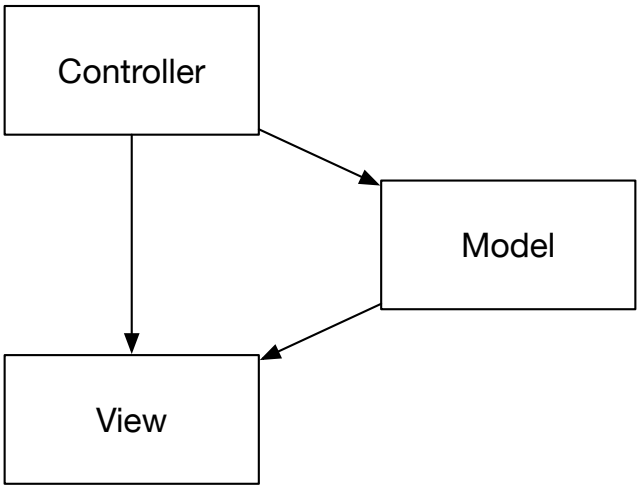
These frameworks all focus on *functions*. A request comes in, goes through some sort of process, and a response flows out. Even within object oriented frameworks, *requests are functions*, with requests and responses.

As these request functions became more complex, developers searched for better ways to organize the chaos, but it was the work done by a software lab within Xerox — the copier maker — that would provide the ultimate answers. Xerox scientists explored ways to organize client-server programs into manageable modules using a pattern called model-view-controller (MVC).[1] Though these techniques were built for native client-server applications instead of the web, they would influence web development for decades to come. Forward thinking programmers found these techniques and looked for ways to integrate them into frameworks to make development easier.

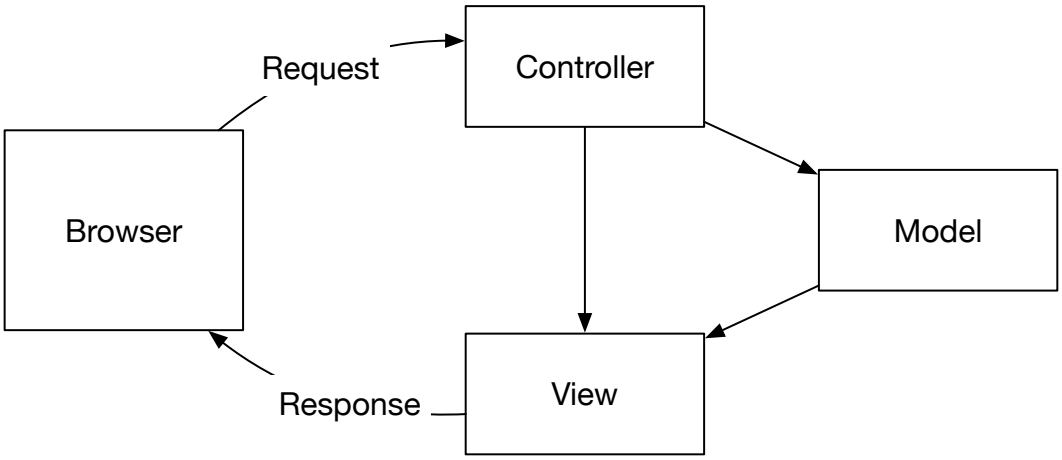
Enter Model-View-Controller, or MVC2

As web development evolved, programmers learned that they had to break their code into layers so they could understand and maintain their applications. The MVC pattern seperate the responsibilities in applications having a user interface between the model, view, and controller. These layers hold the business logic, presentation logic, and coordination between models and views. These layers are called model,view, and controller, as shown in the following figure.

1. https://wiki.c2.com/?ModelViewControllerHistory



Over time, programmers modified MVC to work on the web with a request-response flow. These developers (from many different backgrounds) eventually settled on a new name and programming model called *MVC2* or *model-2*. The following figure shows how it all works.



A user makes a request. Then, a *router* layer receives that request and routes it to a *controller*. The controller fetches external data, perhaps from a database, from the *model* layer, and then passes to a *view*. The *view* layer will use the state and perhaps a *template* to render the data.

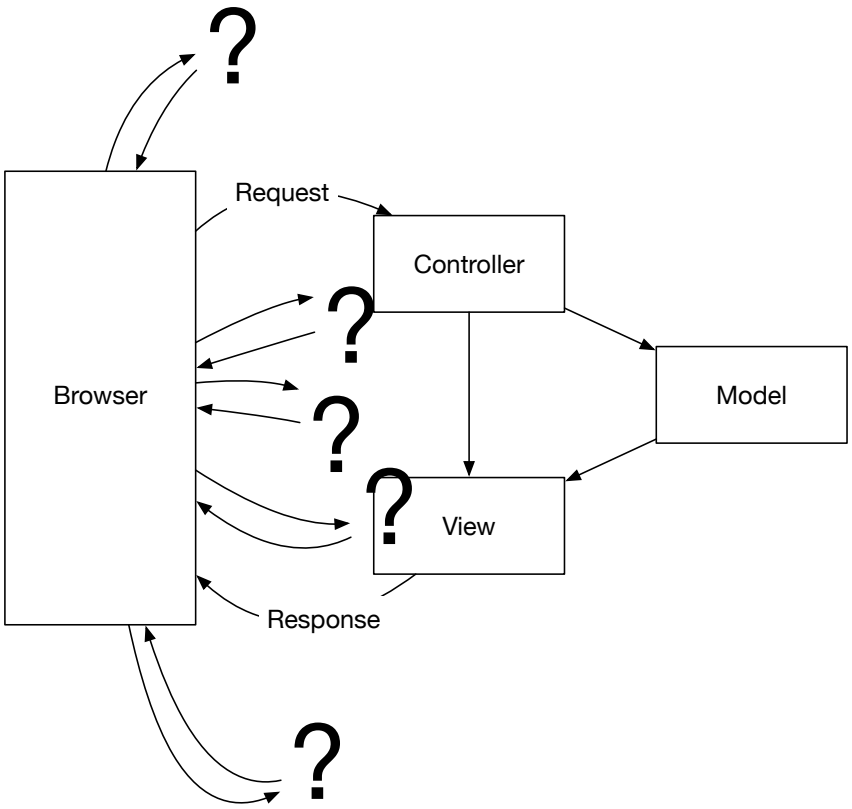
This programming model was tremendously popular, and exploded with the growth of the Java programming language. Model-2 is still broadly used today in many Java frameworks, Ruby on Rails, and even Elixir's Phoenix.

Users Demand Interactive Pages

Eventually, users demanded more than the stodgy, static request-response applications. Google pushed the state of the art with Google Maps, and Google

Mail. Twitter and Facebook shared live timelines that updated without user interaction. MVC2 was too limited to deliver those new applications. Over time, frameworks sought ways to build APIs to make *partial requests* to the server where each request changes.

Seemingly overnight, all web applications became distributed. Each interactive web page had JavaScript code to send requests to the web application server, retrieve the results, and stitch them into the web page. On the server, a single page might have dozens of tiny parts, with each tiny user interaction working like its own web request. Many different programming models emerged to handle these tiny requests, as in the following chaotic image.



The question marks in this figure show individual web requests that all service the same individual page. Each one requires its own MVC2 design. Many different frameworks in several different languages attempted to solve this problem. Building interactive applications became difficult, almost unmanageable. Dozens, even hundreds, of JavaScript frameworks emerged to weave order out of this chaos. Unfortunately, the management of these JavaScript

libraries became a problem of its own, and each web project became an increasingly distributed mess.

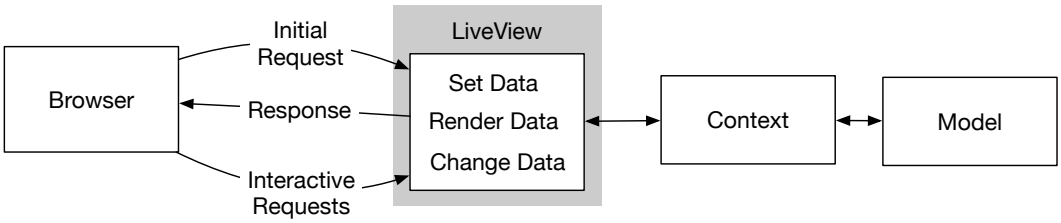
This is where we are today. MVC2, a framework that was never built to handle tiny requests and responses, is breaking under the weight of the interactive single-page app. Many tiny interactive requests from the same page require broader skills across multiple languages. Something has to give.

Enter LiveView

Phoenix LiveView is a whole new programming model built from the ground up to handle single-page flows. Rather than building *functions* satisfying independent requests, LiveView is centered around *state*. Developers work in two distinct dimensions. They render the state with a function. Then, other programs—including the browser—can change the state with events.

In this series, we're going to work through some basic applications, and then ramp up the complexity slowly over time. Throughout this book when we refer to to the LiveView library, we'll use the camel-case word `LiveView`. When we're talking about an interactive view built with this library, we'll use two lower case words *live view*.

Let's talk about how it works. LiveView assumes the burden of the layers between the client and server, including the layer of JavaScript that runs on the browser. With LiveView taking the responsibility of distributed requests between the JavaScript client and Elixir server, programming gets much simpler. The following figure tells the story.



Here's what's happening. LiveView is a much more interactive model than model-2. A *live view* is an interactive view that surrounds a bit of data. When a live view gets an initial request, it establishes some data, and then renders an initial HTML response. Each `render` is a pure function that accepts only the data for a live view. The initial response is pure HTML, so the initial render is easy to optimize for search engines and the like.

Once the live view renders the initial request, it listens for events. Rather than these events triggering a render, the events trigger a *change in state*. Then, each state change triggers a render. Did you get the difference between

live views and MVC2? It all depends on clearly defining the responsibilities of rendering and state change.

There's a subtle but important layer separating the responsibilities of changing state and rendering. With LiveView, a programmer has to consider only two major kinds of functions, and the tiny helper functions that support them. The `render` functions transform state to HTML (or some other kind of string). Events trigger handlers, which change data. If this design seems a little hazy right now, don't worry. We're going to get plenty of practice working with them.

In the sections that follow, we'll talk about how these two concepts work. We will write very little code, but you'll immediately see the the interactive nature of what's happening. We'll start with the initial request to a LiveView.

The Initial Request

Let's build a basic LiveView project. We'll create a Phoenix application called Dazzle that plays with a counter to render a string a few different ways. Building something so simple will let us focus on how LiveViews work rather than the design of our application. Let's build a project. Luckily, this project is going to generate a little LiveView code that will give us something to study.

Install Phoenix version 1.5.[2] Then, create a Phoenix project:

```
[liveview] ➔ mix phx.new dazzle --live
... create project files ...

Fetch and install dependencies? [Yn] Y
... asset installation ...
... instructions to start server ...

[liveview] ➔ cd dazzle
[dazzle] ➔ mix ecto.create
The database for Dazzle.Repo has been created
...

[dazzle] ➔ mix phx.server
...
webpack is watching the files…
...
```

We've shortened these listings, but you get the idea. If all goes well, you can point your browser to `http://localhost:4000` and see the familiar Phoenix startup screen. If you get an error, don't panic. While LiveView is pretty young, the community is vibrant. Just paste a chunk of your error into Google, and the

2. https://hexdocs.pm/phoenix/installation.html

excellent LiveView community will point you to the source of the problem and potential solutions.

You might not be able to tell right now, but this initial page is a live view. Let's talk about some of configuration that sets up a live view. When you pointed your browser to http://localhost:4000, you started a cascade of functions. Don't worry. They are not hidden within many layers of framework code. They're all explicit. You can see every line of code that Elixir touched.

Elements of a LiveView

As you might imagine, the LiveView programming model is simple. You'll need to consider these steps:

Configure the route: Phoenix connections start with an endpoint, a bit of code and configuration that describes the communication protocols, the security configuration, and various policies that every request must honor. Each individual type of request needs a route you'll add to router.ex. Together, these bits of code combine with a few other files to configure the functions Phoenix needs to accept requests.

Establish the data structure

Live views implement data, and you need to establish the data that goes into your LiveView. Each new request will go through a mount function.

Transform the socket to HTML with render/1

Live views transform data to HTML. Each time the state changes, LiveView will call your render/1 function. The first invocation will send a pure HTML page to the client. Subsequently, LiveView will send only changes down to the client when state changes.

Receive events and change data with handlers

Each event, whether a mouse click, a key press, or a simple process message, calls a handler. Each of these handlers transforms the state in a socket, triggering a render/1 call.

Keep in mind that after you get past the router, all LiveView code is composed of functions. You can organize these functions any way you want. As we do more complex tasks with our live views, we'll use more sophisticated code layering techniques.

Let's look at a detailed example. We'll implement a counter, but your back end code can do anything you want. Later chapters will add in some complexity so you'll have a better idea of how to design your code. We'll build a basic

project piece by piece, starting with an endpoint, and ending with a fully interactive live view.

Start with the Endpoint

Remember, each Phoenix request starts with an endpoint. Our requests will go first to lib/dazzle_web/endpoint.ex. You don't need to understand everything that's happening in this file, but I do want to point out a couple of lines of code to you. Open up the endpoint file. Near the top, you'll find this line:

```
socket "/live",
       Phoenix.LiveView.Socket,
       websocket: [connect_info: [session: @session_options]]
```

This line of code establishes a socket for communication between the LiveView pages on the browser and your code on the server. This design is nice because you won't need to establish a new route for each tiny interaction between the client and server. All LiveView communication will go over this socket.

Next, look at one of the last lines of code in the file:

```
plug DazzleWeb.Router
```

The plug term may seem strange to you, but don't worry. A plug is a function. Each plug in this file takes an argument, a structure called Plug.Conn. The plug function transforms the connection struct in some way, and then returns the transformed plug. The Router plug is the plug that chooses which code to execute based on the URL. We'll look at the router next.

The Endpoint Calls the Router

As we move from the endpoint to the router, we'll shift our attention to the next bit of configuration, the lib/dazzle_web/router.ex module. When you open that file, you'll see more plugs. Remember, these plugs are functions that take and return Plug.Conn. Here are the lines that matter:

```
pipeline :browser do
    plug :accepts, ["html"]
    plug :fetch_session
    plug :fetch_live_flash
    plug :put_root_layout, {DazzleWeb.LayoutView, :root}
    plug :protect_from_forgery
    plug :put_secure_browser_headers
  end

  ...

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