Programming Phoenix LiveView

Interactive Elixir Web Programming
Without Writing Any JavaScript

The Pragmatic Bookshelf
Raleigh, North Carolina

The Pragmatic Programmers

Programming Phoenix LiveView

Interactive Elixir Web Programming Without Writing Any JavaScript

Bruce A. Tate and Sophie DeBenedetto

edited by Jacquelyn Carter

Programming Phoenix LiveView

Interactive Elixir Web Programming
Without Writing Any JavaScript

Bruce A. Tate
Sophie DeBenedetto

The Pragmatic Bookshelf
Raleigh, North Carolina

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*.

ISBN-13: 978-1-68050-821-5
Encoded using the finest acid-free high-entropy binary digits.
Book version: B1.0—February 25, 2021

CHAPTER 7

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 nothing 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 its 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

```elixir
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

```elixir
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.Form-Component, 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:

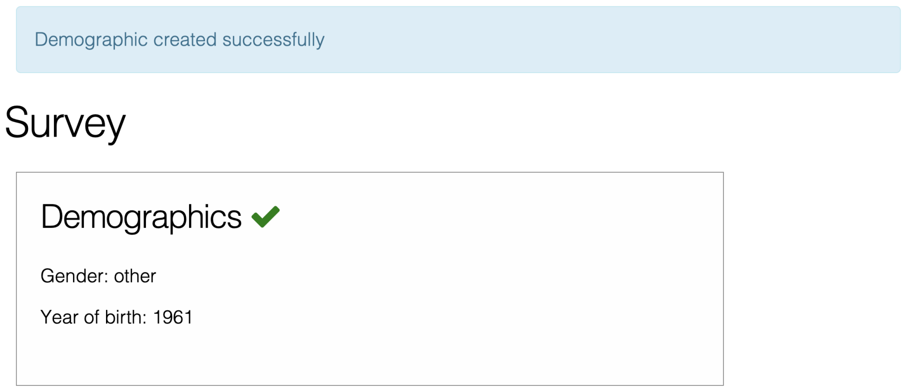
```
def mount(_params, %{"user_token" => token}, socket) do
  {:ok,
    socket
    |> assign_user(token)
    |> assign_demographic()}
end
```

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—i.e. 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.

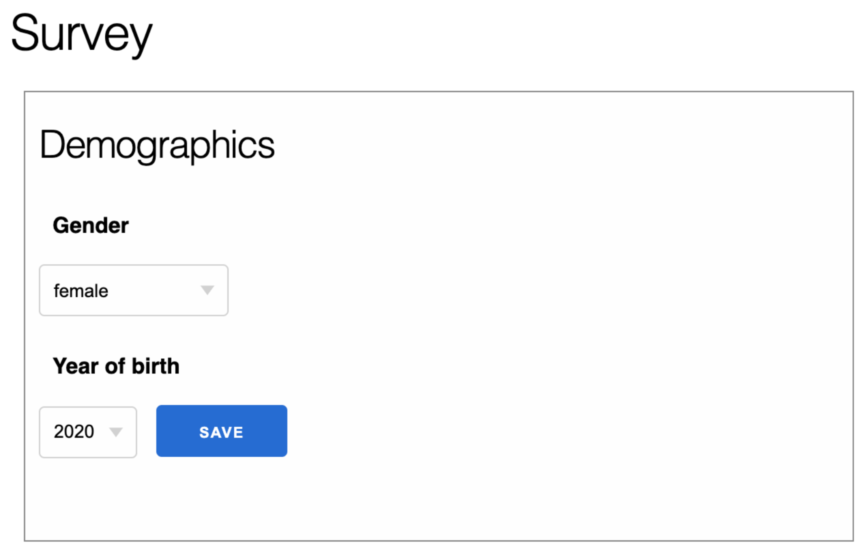
Now, we can render it from our template, below the header, with a call to live_component/3:

stateful_components/pento/lib/pento_web/live/survey_live.html.leex

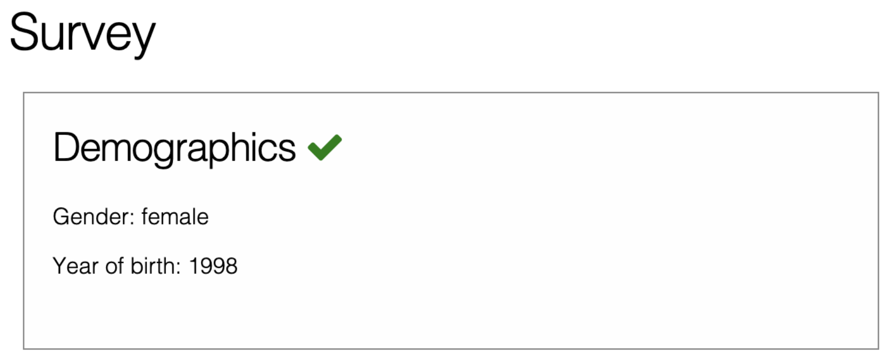
```
<%= if @demographic do %>
  <%= live_component @socket,
        PentoWeb.DemographicLive.ShowComponent,
        demographic: @demographic %>
```

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:



And if a user who *has* filled out the demographic form visits the page, they will see their demographic details rendered:



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.