The Pragmatic Programmers

Programming Phoenix LiveView

Interactive Elixir Web Programming Without Writing Any JavaScript

Bruce A. Tate and Sophie DeBenedetto

edited by Jacquelyn Carter

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

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

Copyright © The Pragmatic Programmers, LLC.

CHAPTER 7

Live Components

In the previous chapter, we began building an interactive survey feature by creating the backend core and boundary functionality, along with a live view and function component to make up the beginnings of our UI. In this chapter, we'll build *live components* to manage demographics and ratings. Our live component will manage its own state in its own lifecycle. We'll compose an interactive survey with it layer by layer.

Along the way, you'll learn how live components work. You've seen this concept briefly within the forms we generated. The function components you built in the previous chapter render static markup such as HTML. Those components don't have their own state and users can't modify the state through events. Live components are different. They have their own state, can process their own events, and even have their own lifecycle. You'll use them to build fully interactive parts of a page, like counters, timers, data-backed forms, and the like. Our live components will be tiny forms that permit our users to rate products.

You'll explore for yourself how these communicate with their parent live view, you'll continue to 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.

Build the Live Demographic Form Component

Let's put together a plan before we get started. We'll begin by implementing a live component module to house our demographic form. We'll use the available LiveView and LiveComponent lifecycle callbacks to establish the state of our form component. Then, we'll render the form markup using the same `simple_form/1` function component you saw in earlier chapters. Finally,

we'll teach our form component to respond to user input and save demographic data for the user.

Define the Live Component

First up, create a new file lib/pento_web/live/demographic_live/form.ex and define the form component module like this:

```
defmodule PentoWeb.DemographicLive.Form do
  use PentoWeb, :live_component
  alias Pento.Survey
  alias Pento.Survey.Demographic
end
```

This is simple enough to begin with. We implement a module that uses the :live_component behavior in order to create a live component. Then, we add a few aliases that we'll take advantage of in a bit.

We'll use LiveView's simple_form/1 function to construct the demographic form. This function requires a form wrapping a changeset, so we'll need to store one in our component's state. Like full live views, live components first have one workflow to establish the initial page and then a change management workflow to modify and render the component state. As we did in our live views, we'll use CRC to think about how to organize our code. For the initial mount/render workflow, we don't really need a reducer. The flow looks like this:

```
inputs |> construct() |> convert()
```

In this diagram, the construct() refers to a function that establishes the initial state in the form of a socket, and convert() refers to a function that transforms all of that socket data to HTML. In the constructor, the component lifecycle comes into play. For our component, the update callback will act as our constructor, and the render callback will act as our converter. Why the update function and not the mount function? To answer that question, we need to understand the live component lifecycle.

When we render a live component, LiveView 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. You'll use this function to do one-time setup.

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. You'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.

Live components will always follow this three-step process when they are first mounted and rendered. Then, when the component updates in response to changes in the parent live view, only the update/2 and render/1 callbacks fire. Since these updates skip the mount/1 callback, the update/2 function is the safest place to establish the component's initial state.

We'll use the update/2 callback as our constructor to add a Demographic changeset to socket.assigns so we can render it in a form on the template. For a converter, we'll let the implicit render/1 function render a HEEx template that matches the name of our live component.

Let's get ready to set the initial state of our live component now. Our demographic *belongs to* a user and we'll need access to that user to construct a demographic changeset. Recall that we're planning to render our form live component from the SurveyLive template defined in lib/pento_web/live/survey_live.html.heex like this:

```
<%= if @demographic do %>
  <DemographicLive.Show.details demographic={@demographic} />
<% else %>
  <h2>Demographic Form coming soon!</h2>
<% end %>
```

The SurveyLive socket assigns already contains a @current_user assignment, so we'll make sure to pass it into our live component. The DemographicLive.Form.update/2 function can then safely rely on a current user.

With that assumption in mind, we can implement an update/2 function to build a Demographic struct and a form struct, like this:

stateful_components/pento/lib/pento_web/live/demographic_live/form.ex

```
def update(assigns, socket) do
  {
    :ok,
    socket
    |> assign(assigns)
    |> assign_demographic()
```

• Click HERE to purchase this book now. discuss

```
    |> clear_form()
  }
end
```

This code uses the same technique we used in our SurveyLive.mount/3 function. We build a couple of single-purpose reducers to add the demographic and empty form 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:

stateful_components/pento/lib/pento_web/live/demographic_live/form.ex

```
defp assign_demographic(
  %{assigns: %{current_user: current_user}} = socket) do
  assign(socket, :demographic, %Demographic{user_id: current_user.id})
end
```

It simply adds an empty demographic struct.

And here are two functions to add forms to our socket, one for an empty, or "clear", form and one for a form with a validated changeset:

stateful_components/pento/lib/pento_web/live/demographic_live/form.ex

```
defp assign_form(socket, changeset) do
  assign(socket, :form, to_form(changeset))
end

defp clear_form(%{assigns: %{demographic: demographic}} = socket) do
  assign_form(socket, Survey.change_demographic(demographic))
end
```

For the assign_form/2 function, we convert an existing changeset to a form. For the clear_form/1 function, we take the empty demographic from the socket, wrap that in a changeset, and then add that to the socket with assign_form/2. Once the update/2 function finishes, the component renders the template. Let's define that template now to render the demographic form for our shiny new changeset.

Render The Demographic Form

You've seen what a LiveView form looks like. We won't bore you with the details. For now, add this to lib/pento_web/live/demographic_live/form.html.heex:

```
<div>
    <.simple_form
      for={@form}
      phx-change="validate"
      phx-submit="save"
      id={@id}>

      <.input
```

```
        field={@form[:gender]}
        type="select"
        label="Gender"
        options={["female", "male", "other", "prefer not to say"]} />
      <.input
        field={@form[:year_of_birth]}
        type="select"
        label="Year of Birth"
        options={Enum.reverse(1920..2023)} />

      <:actions>
        <.button phx-disable-with="Saving...">Save</.button>
      </:actions>
    </.simple_form>
</div>
```

Notice that our form is contained within a root <div> element. All live components require a single root element in their HTML templates. Also notice that we're also using the <.simple_form> component defined in the CoreComponents module. Let's dig briefly into our form rendering code.

Our update/2 function added the form struct to our socket assigns, and we access it with @form in our call to simple_form/1. The simple_form/1 function takes in the form struct, has an id, and applies the phx-submit LiveView binding for saving the form. Our form has labels, fields, and error tags for each field we want the user to populate. Finally, there's a submit tag with a phx-disable_with function—a little nicety that LiveView provides to handle multiple submits.

We're ready to put it all together by rendering the form component from the SurveyLive template in pento/lib/pento_web/live/survey_live.html.heex, like this:

Render the component from the template using the live_component/1[1] function component, like this:

```
<%= if @demographic do %>
  <DemographicLive.Show.details demographic={@demographic} />
<% else %>
  <.live_component module={DemographicLive.Form}
        id="demographic-form"
        current_user={@current_user} />
<% end %>
```

The live_component/1 function is a function component made available to us by the LiveView framework. It takes in an argument of some assigns and returns a HEEx template that renders the given component within the parent live view. When using live_component/1 to render a live component, you must specify

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

an assigns of `module`, pointing to the name of the live component module to mount and render, and an assigns of `id`, which LiveView will use to keep track of the component. Also note the `{}` interpolation syntax we're using—remember that this syntax is required when interpolating within HTML or HEEx tags.

Now log in as a user that does not have an associated demographic record, or simply temporarily tweak the `if` statement to render `if !@demographic`. Then, visit `/survey` to see our survey page with the demographic form, as shown here.

Survey

Gender

female

Year of Birth

2023

Save

If you try to submit the form, you'll find the live view crashes, but maybe not for the reason you thought. Look at the logs:

```
[error] GenServer #PID<0.1159.0> 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 parent `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.Form` should handle both the state for the survey's demographic section and the events to manage that state. To fix this, add the following `phx-target` attribute to your form in the `lib/pento_web/live/demographic_live/form.html.heex` template:

```
<.simple_form
      for={@form}
```

```
      phx-change="validate"
      phx-submit="save"
      id={@id}
      phx-target={@myself}  <!-- add this line -->
  <!-- ... -->

</.simple_form>
```

The @myself assignment is made available in our component by LiveView, for free, and it always refers to the current component. This will ensure that any events sent by LiveView bindings on this element will go to the current component, rather than the parent live view.

Now we can send events to our demo form, so it's time to add some handlers.

Manage Component State

First, we'll briefly revisit the live component lifecycle that we'll take advantage of in order to manage component state. Then, we'll implement the event handlers we need to respond to our form events.

Consider the preload/1 Callback

Whenever live_component/1 is first invoked, the component will call mount/1, update/2 and render/1. Sometimes, these callbacks are not enough. You might need an additional callback called preload/1 to prevent a potentially serious N + 1 performance problem. For the mount/render workflow, LiveView calls preload/1, then mount/1, followed by update/2, and finally render/1. The change management workflow *drops* the mount/1, but maintains preload/1, then update/2, and finally render/1.

We won't take advantage of preload/1 in our component, but it's worth discussing what it can do for us. This function lets LiveView load all components of the same type at once, potentially saving many extra database queries. 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.

Because our component doesn't render any lists or child components, we can safely move forward without implementing preload/1. We're ready to teach our live component how to handle events.

Handle The Save Event

We're already sending events to our component when the form is saved. Now, we need to implement a handle_event/3 function for that "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 from 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.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 receiving the event. There's one problem though. Our form params don't include the "user_id", and the Survey.create_demographic function we plan to call in our reducer expects to receive a complete map of all of the demographic params needed to create a demographic. We can fix this with a simple helper function to get the ID of the current user socket assignment and add it to the params map:

stateful_components/pento/lib/pento_web/live/demographic_live/form.ex
```
def params_with_user_id(params, %{assigns: %{current_user: current_user}}) do
  params
  |> Map.put("user_id", current_user.id)
end
```

We'll call on this function in the event handler to construct the complete params to pass to our reducer, like this:

```
def handle_event("save", %{"demographic" => demographic_params}, socket) do
  params = params_with_user_id(demographic_params, socket)
  # ...
end
```

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!

    {:error, %Ecto.Changeset{} = changeset} ->
      assign_form(socket, 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 form 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.Form:

stateful_components/pento/lib/pento_web/live/demographic_live/form.ex
```
def handle_event("save", %{"demographic" => demographic_params}, socket) do
  params = params_with_user_id(demographic_params, socket)
  {:noreply, save_demographic(socket, 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 message. Our LiveView is a plain old GenServer[2] so it implements its own behaviour with its own callbacks. handle_info/2[3] is the GenServer callback function for receiving generic Elixir messages. Update save_demographic/2 to send a message to the parent on success:

stateful_components/pento/lib/pento_web/live/demographic_live/form.ex

```
defp 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_form(socket, 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

2. https://hexdocs.pm/elixir/1.14/GenServer.html
3. https://hexdocs.pm/elixir/1.14/GenServer.html#c:handle_info/2

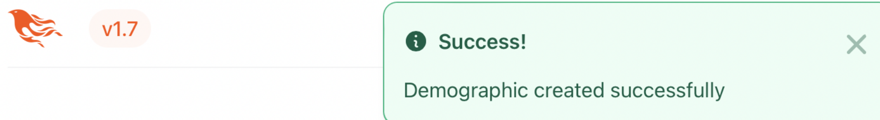
the socket. As usual, we want skinny handlers, so we call the handle_demographic_created/2 reducer to do the work. Now, we need to decide exactly what work to do in the handle_demographic_created/2 function.

We'll want to notify the user that the save was successful, and store the new demographic in the socket. Let's implement those new features in handle_demographic_create/2, like this:

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. The SurveyLive live view will re-render, this time with the :demographic key in socket assigns set to a valid demographic struct. Now, when the conditional logic in the SurveyLive template runs, the check for the @demographic assignment will evaluate as true. So, we will invoke the DemographicLive.Show.details function component to display the demographic details instead of displaying the form.

Let's see it in action. Log in as a user that does not yet have an associated demographic record. Then, point your browser at /survey and submit the demographic form. You should see the flash message, and you'll also see the form replaced with the demographic details, as in this image:



Survey
Demographics ✓
Gender: male
Year of birth: 1965

Great! With that, we've beautifully composed a set of layered components to support the beginnings of our survey UI. Each piece of the puzzle is simple and sweet—the SurveyLive live view conditionally renders either a child function component to display the demographic details or a child live component to display the interactive form. The SurveyLive view's state runs the show—the presence or absence of a demographic in socket assigns tells the child com-

ponents how to behave, and each child component has just one job to do. In this way, we can break down even complex view logic into simple components.

Our survey UI has a solid foundation. We're ready to build out the ratings flow.