Extracted from:

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

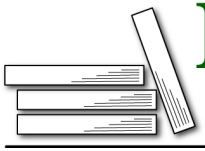
edited by Jacquelyn Carter

Programming Phoenix LiveView

Interactive Elixir Web Programming Without Writing Any JavaScript

Bruce A. Tate
Sophie DeBenedetto

The Pragmatic Bookshelf
Raleigh, North Carolina



Pragmatic Bookshelf

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

Add Filters to Make Charts Interactive

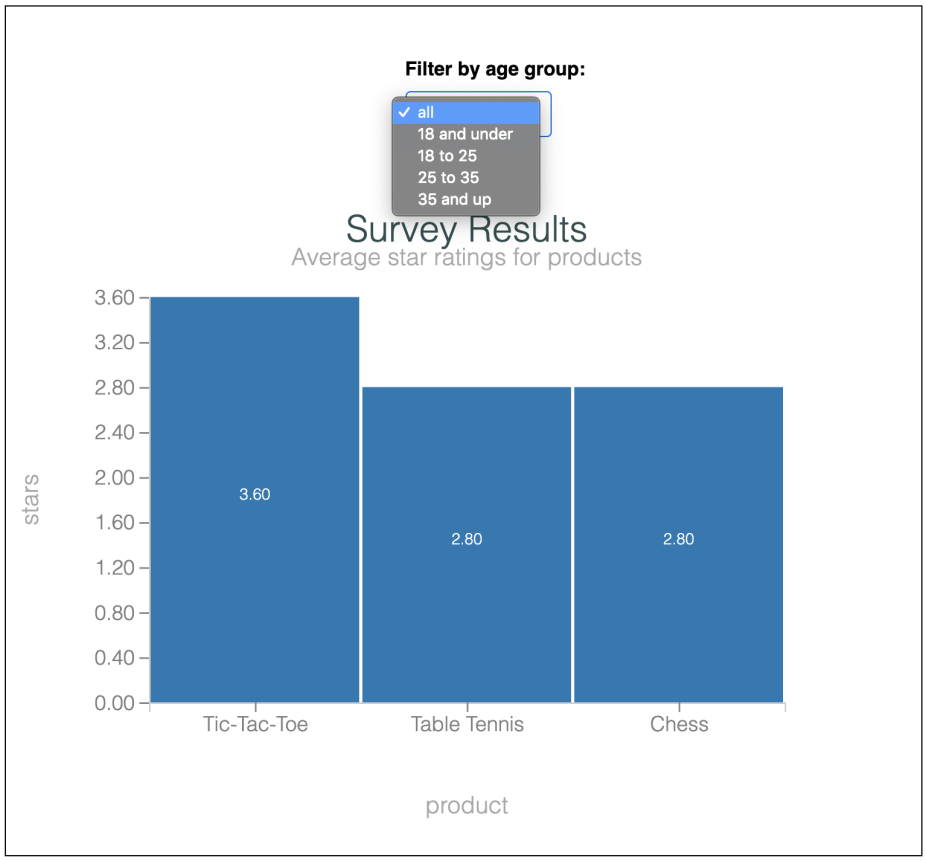
So far, we have a beautiful server-side rendered dashboard, but we haven't done anything yet that is LiveView specific. In this section, we change that. We'll give our users the ability to filter the survey results chart by demographic, and you'll see how we can re-use the reducers we wrote earlier to 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:

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
```

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

In IEx, run this 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
```

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.

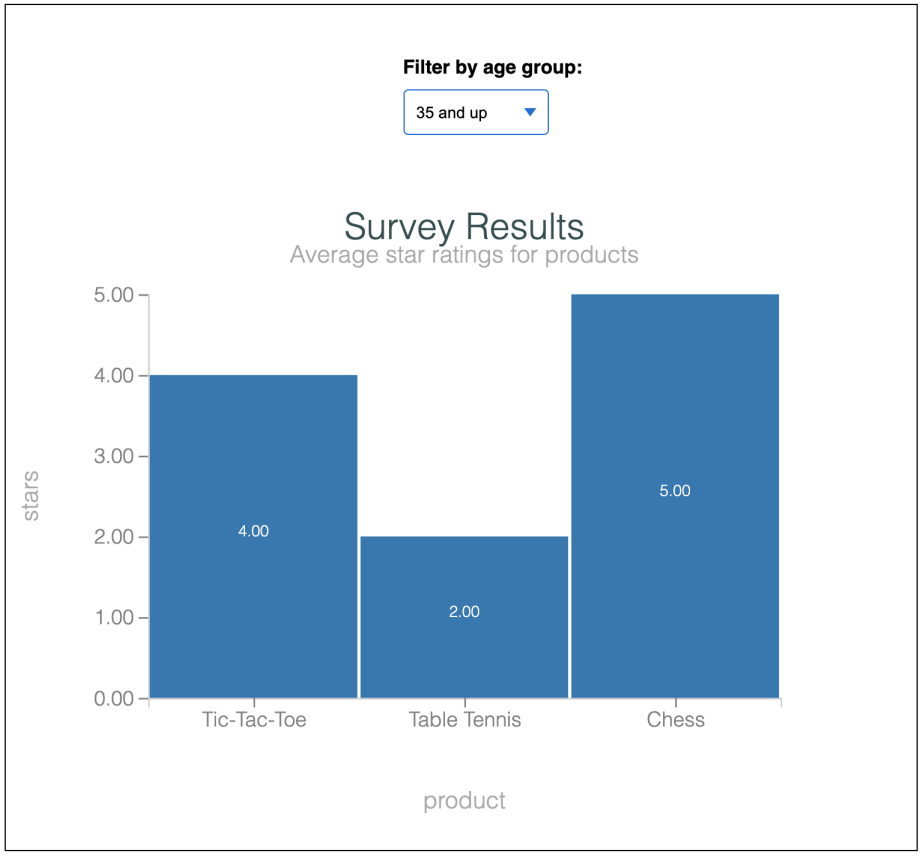
First, we update socket assigns :age_group_filter with the new age group filter from the event. We do this by implementing a new version of our assign_age_group_filter/2 function.

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:



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
(contex 0.3.0) lib/chart/mapping.ex:180: Contex.Mapping.missing_columns/2
```

```
...
(contex 0.3.0) lib/chart/mapping.ex:139: Contex.Mapping.validate_mappings/3
(contex 0.3.0) lib/chart/mapping.ex:57: Contex.Mapping.new/3
(contex 0.3.0) lib/chart/barchart.ex:73: Contex.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
```

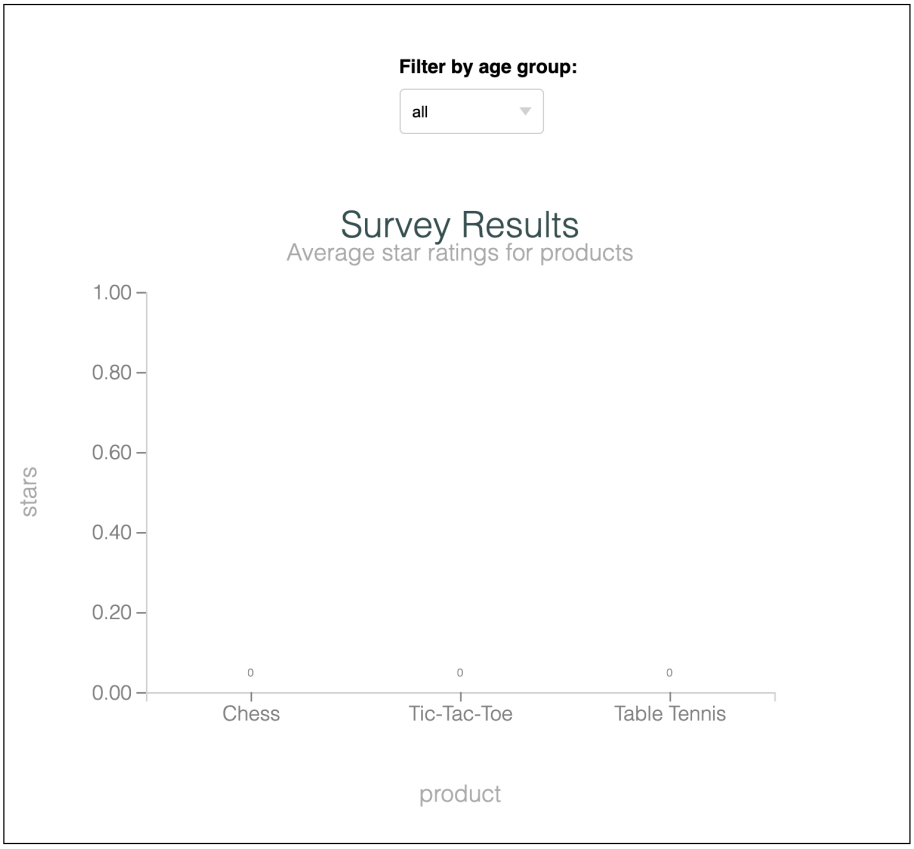
We can update our LiveView to implement the necessary logic:

```
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.