Add Filters to Make Charts Interactive

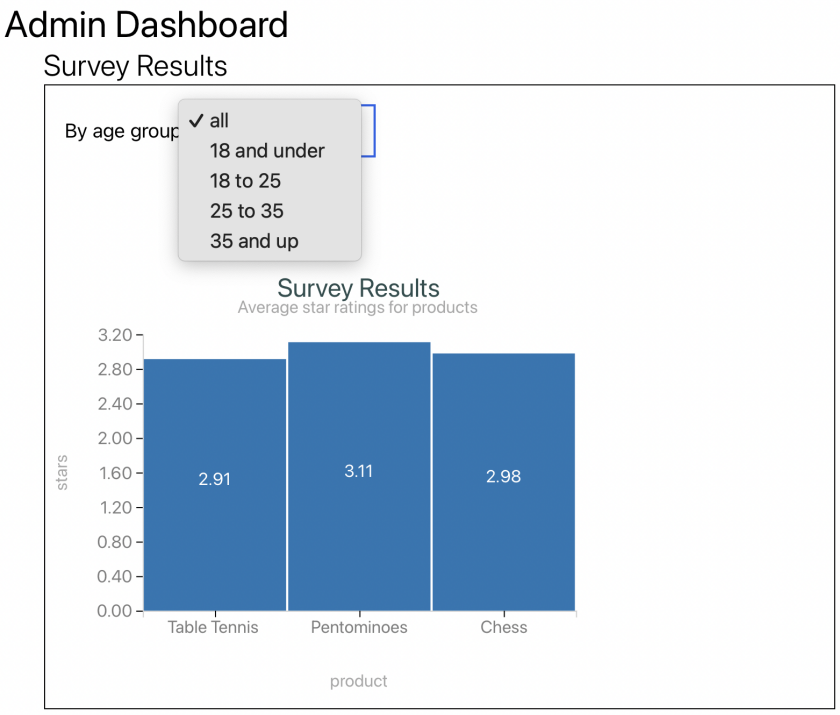
So far, we have a beautiful server-side rendered dashboard, but we haven't done anything yet that really leverages LiveView's interactive capabilities. In this section, we change that. We'll give our users the ability to filter the survey results chart by demographic, and you'll see how we can re-use 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 live 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 live 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 input. 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.

Establish Test Data

To have enough data to play with, we're going to build a simple seeds file that will generate users, random demographics for them, and random ratings. This data will let us put our interface through its paces much more easily than if we had to build out random ratings by hand. We're going to build a rating_seeds.exs script much like our seeds.exs script.

First, we need to leave a comment telling our users how to use the file and import or alias the modules our code will use.

interactive_dashboard/pento/priv/repo/rating_seeds.exs

```
#     mix run priv/repo/rating_seeds.exs

import Ecto.Query
alias Pento.Accounts.User
alias Pento.Catalog.Product
alias Pento.{Repo, Accounts, Survey}
```

Our code will use context functions to create database User, Demographic, and Rating records. It will also need to make use of the schema information for Product and User data. That's a good start. Let's create some users:

interactive_dashboard/pento/priv/repo/rating_seeds.exs

```
for i <- 1..43 do
  Accounts.register_user(%{
    email: "user#{i}@example.com",
    password: "userpassword#{i}"}) |> IO.inspect
end
```

We create 43 users. We really don't want an even number of them because we don't want the percentages to come out too clean. The exact number doesn't matter. To pass our changeset, we have to pass a valid email and a valid password. We'll point our email to example.com because we won't accidentally email known users that way. Our passwords are long enough to pass the validation. Now, we'll need a little setup data to use as we create demographics and ratings:

interactive_dashboard/pento/priv/repo/rating_seeds.exs

```
user_ids = Repo.all(from u in User, select: u.id)
product_ids = Repo.all(from p in Product, select: p.id)
genders = ["female", "male", "other", "prefer not to say"]
years = 1960..2017
stars = 1..5
```

We have lists we'll use in two ways. Our for comprehension will map over user and product ids. We'll pick random elements for the other lists. This data won't be truly representative of our real world data, but we really don't care at this point. We just want tangible changing values to use for our bar charts and filters. Let's create some demographics:

interactive_dashboard/pento/priv/repo/rating_seeds.exs

```
for uid <- user_ids do
  Survey.create_demographic(%{
    user_id: uid,
    gender: Enum.random(genders),
    year_of_birth: Enum.random(years)})
end
```

Our comprehension covers all users in the database. We'll create demographics for each user, picking a random gender and year setting. Finally, we'll create some ratings. Add this at the end of your seed file:

```
# pento/priv/repo/rating_seeds.exs
for uid <- user_ids, pid <- product_ids do
  Survey.create_rating(%{
    user_id: uid,
    product_id: pid,
    stars: Enum.random(stars)
  })
end
```

This for comprehension will cover each possible combination with one user id and one product id. We create a rating with each of those values and a random number of stars. Run the script with mix run priv/repo/rating_seeds.exs and you'll see a bunch of SQL flying by, denoting new values inserted into our database. Load your /admin/dashboard page to satisfy yourself that there is more data, and then we can move on.

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, so we'll break it down. 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

Before you run the query in IEx, open up lib/pento_web/live/admin/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:

```
def 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 component. Before we can actually change data on the page, we'll need a filter in the socket when we call 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.Admin.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 pipeline 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 the assign_products_with_average_ratings/1 function in Admin.SurveyResultsLive to use the new age group filter:

```
def 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.

Now, we need to build the form input.

Send Age Group Filter Events

We're ready to add some event handlers to our component. We'll need a div to hold our form, like this:

```
<section class="ml-8">
  <h2 class="font-light text-2xl">Survey Results</h2>
  <div id="survey-results-component">
    <div class="container">
      ...filters will go here...
    </div>
  </div>
  <div id="survey-results-chart">
    <%= @chart_svg %>
```

```
  </div>
</section>
```

Inside this div, we'll add the select input for the age group filter and default the selected value to the @age_group_filter assignment. Go ahead and add this to your template now:

interactive_dashboard/pento/lib/pento_web/live/admin/survey_results_live.html.heex

```
<div>
  <.form
    for={%{}}
    as={:age_group_filter}
    phx-change="age_group_filter"
    phx-target={@myself}
    id="age-group-form"
  >
    <label>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} selected={@age_group_filter == age_group}>
          <%= age_group %>
        </option>
      <% end %>
    </select>
  </.form>
</div>
```

LiveView works best when we surround individual form helpers with a full form. We render a drop-down menu in a form using the form/1[4] function component. Our approach here is a little different than what you've seen in this book so far. We don't need to track changes to data with the help of a changeset, so we didn't create one for our age group filter. As a result, we didn't create a form struct and add it to socket assigns. That is likewise not needed here. Instead, we're going with a simpler approach. We're using a form for an empty map, and providing some additional instruction to teach the form how to behave. Let's take a closer look at how the form function component works with this empty struct.

Open up an IEx session with iex -S mix and key in the following:

```
iex> i %{}
Term
  %{}
...
Implemented protocols:
  Collectable, Enumerable, IEx.Info, ... Phoenix.HTML.FormData, ...
```

4. https://hexdocs.pm/phoenix_live_view/Phoenix.Component.html#form/1

```
iex> i Pento.Catalog.change_product(%Pento.Catalog.Product{}, %{id: 1})
Term
  #Ecto.Changeset<...>
Implemented protocols
  IEx.Info, Inspect, Jason.Encoder, ..., Phoenix.HTML.FormData, ...
```

Notice that *both* our product changeset *and* an empty map implement the Phoenix.HTML.FormData protocol. So, when we provide the empty map to the for attribute of our form function component, the to_form/2 function is called under the hood to convert the empty struct to a Phoenix.HTML.Form struct.

In addition to providing the empty map to our form, we've added a few other attributes as well. We want the form events to target the live component itself (rather than the parent live view), so we set the phx-target attribute to @myself. The form also has the phx-change event binding, since we want to respond to the event as soon as the user selects an age group, rather than forcing them to click a submit button.

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/admin/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 and construct the SVG chart with that updated list. 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:

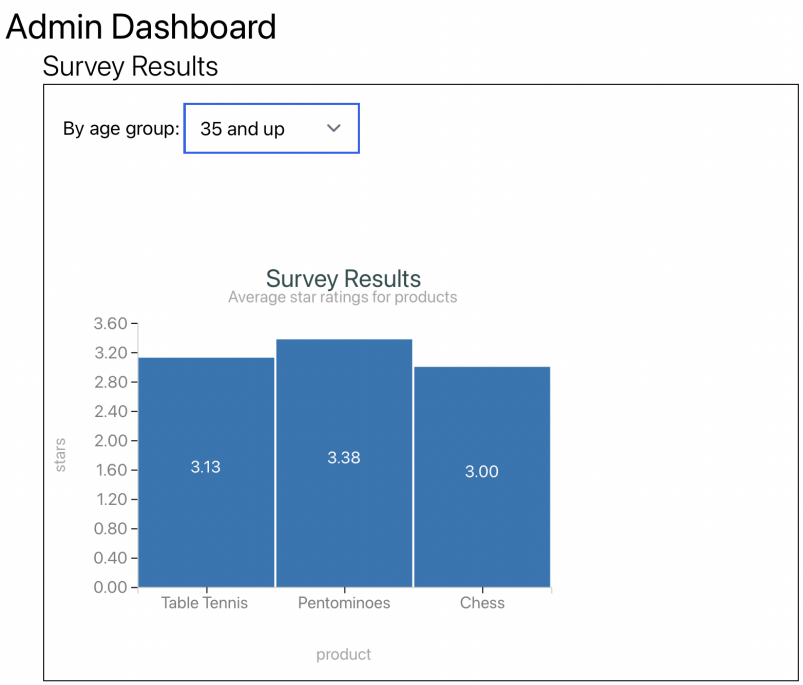
interactive_dashboard/pento/lib/pento_web/live/admin/survey_results_live.ex

```
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 for the given age group filter. 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 needs to account for an important edge case before we move on. There might not be any survey results returned from our database query! If you select an age group for which no product ratings exist, you'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:

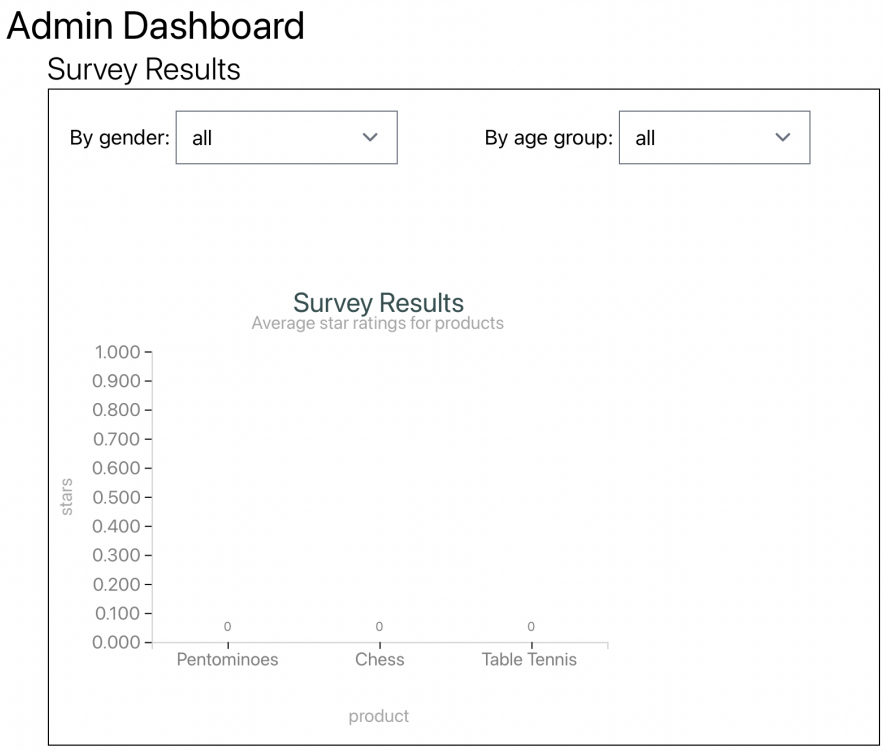
```
def assign_products_with_average_ratings(
      %{assigns: %{age_group_filter: age_group_filter}} =
      socket
    ) do
  assign(
```

```
    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.