Extracted from:

Building Table Views with Phoenix LiveView

Advanced Table UIs for Accessible Data

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Building Table Views with Phoenix LiveView

Advanced Table UIs for Accessible Data

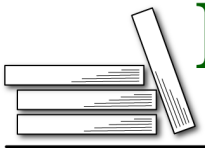
Peter Ullrich
Edited by Michael Swaine

Building Table Views with Phoenix LiveView

Advanced Table UIs for Accessible Data

Peter Ullrich

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*.

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Michael Swaine
Copy Editor: L. Sakhi MacMillan
Founders: Andy Hunt and Dave Thomas

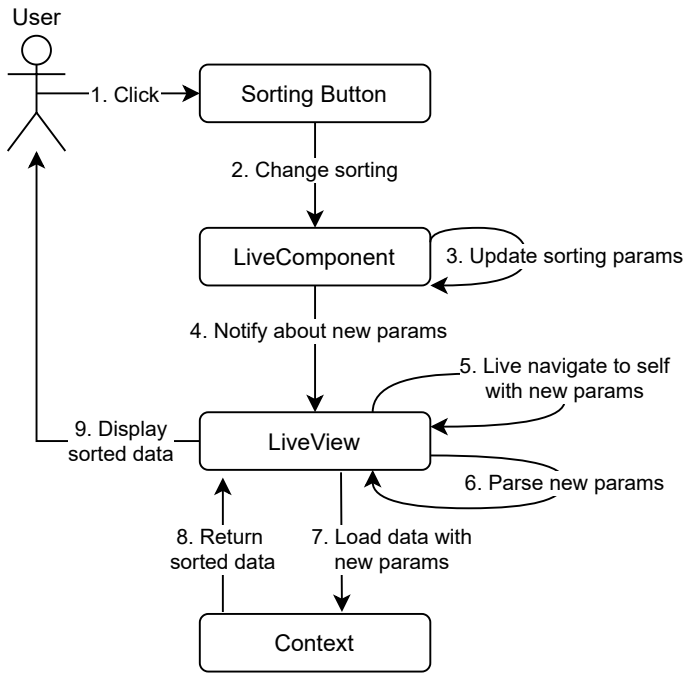
For sales, volume licensing, and support, please contact *support@pragprog.com*.

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

ISBN-13: 978-1-68050-973-1
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—January 2023

Setting up LiveView

We'll use a combination of a LiveComponent, a LiveView, and live navigation to handle and apply changes to the sorting of our meerkat data. In brief, we will use the LiveComponent to handle any user input to the sorting elements of our table UI. The LiveComponent updates the sorting parameters and notifies the LiveView about the changes. The LiveView navigates to itself with the updated sorting parameters added to the URL of our website. Upon completion of the live navigation, the LiveView parses and validates the updated parameters and passes them on to our context. The context returns the sorted data and our LiveView re-renders the table UI with it. The diagram below shows an overview of these steps.



The described approach has the advantages that we move the logic for updating the sorting parameters out of our LiveView and into a reusable LiveComponent. It also allows us to update the URL whenever the user changes their view onto the data. Keeping the URL in sync with our sorting parameters enables the user to share their view by simply copy-pasting the URL. It also prevents the loss of the users' view when they accidentally refresh the website. We can also use it to load specific views, like the latest meerkat data, to the user whenever they access the website.

The flow described above uses the sorting parameters as an example, but we'll also use it for filtering and paginating the data in the upcoming chapters.

Now that we have understood the flow of updating and applying our sorting parameters, let's start implementing them.

Sorting with LiveComponent

As mentioned above, we want our LiveComponent to handle the user interactions, update the sorting parameters, and notify the LiveView about the changes. The SortingComponent that follows does just that. Open lib/meow_web/live/sorting_component.ex, and have a look at the module shown here. We'll go through it step by step afterward.

```
defmodule MeowWeb.MeerkatLive.SortingComponent do
  use MeowWeb, :live_component

  def render(assigns) do
    ~H"""
    <div phx-click="sort" phx-target={@myself} >
      <%= @key %> <%= chevron(@sorting, @key) %>
    </div>
    """
  end

  def handle_event("sort", _params, socket) do
    %{sorting: %{sort_dir: sort_dir}, key: key} = socket.assigns

    sort_dir = if sort_dir == :asc, do: :desc, else: :asc
    opts = %{sort_by: key, sort_dir: sort_dir}

    send(self(), {:update, opts})
    {:noreply, assign(socket, :sorting, opts}
  end

  def chevron(%{sort_by: sort_by, sort_dir: sort_dir}, key)
      when sort_by == key do
    if sort_dir == :asc, do: "⇧", else: "⇩"
  end

  def chevron(_opts, _key), do: ""
end
```

We want to make the SortingComponent reusable. That's why we let it only render a single div element that shows the key of the field it sorts by and a chevron that indicates its current sorting direction. We can add this div wherever we want now—for example, as a header in our table UI.

Now, let's go through its functionality. Have a look at the handle_event/3 callback. You can see that if a user clicks the SortingComponent, we fetch the current sorting parameters and update the sorting direction from ascending to descending

or the other way around. We then notify the LiveView about the updated parameters by sending a message to self(). Eventually, we prevent any lag in the UI by assigning the updated sorting parameters back to the socket of our LiveComponent. This causes a re-render of our div element with the new sorting direction. This way, the user will see the updated sorting direction immediately, even when the LiveView has a delay in re-rendering the entire table UI.

Adding the LiveComponent to the HEEx Template

Now that we've built the functionality of the SortingComponent, let's add the component to our table UI. Open lib/meow_web/live/meerkat_live.html.heex, and add the SortingComponent as a table header. It should look like the code that follows.

```
<table>
  <thead>
    <tr>
      <th>
        <.live_component
          module={MeowWeb.MeerkatLive.SortingComponent}
          id={"sorting-id"}
          key={:id}
          sorting={@sorting} />
      </th>
      <th>
        <.live_component
          module={MeowWeb.MeerkatLive.SortingComponent}
          id={"sorting-name"}
          key={:name}
          sorting={@sorting} />
      </th>
    </tr>
  </thead>
  <!-- Table body -->
</table>
```

As you can see, we added two table headers for the id and name fields of our meerkat data. Since we use Phoenix 1.6 with Phoenix LiveView 0.17.5, we can use the .live_component-function in our .heex file. If you use an older version, simply replace the .live-component element with the following:

```
<%= live_component
  MeowWeb.MeerkatLive.SortingComponent,
  id: "sorting-name",
  key: :name,
  sorting: @sorting %>
```

You might wonder about the @sorting assign we pass to our SortingComponent. It contains the current sorting key and sorting direction in a map like this: %{sort_by: :name, sort_dir: :desc}.

Now that we have a reusable SortingComponent that handles the user interactions, updates the sorting parameters accordingly, and notifies the LiveView about the changes, let's have a look at how the LiveView handles these changes.

Updating the URL with the New Sorting Parameters

Whenever the user changes the sorting of the table, the SortingComponent sends an {:update, new_sorting_params} message to the LiveView. However, our LiveView doesn't know how to handle that message yet. Open up the MeerkatLive module in lib/meow_web/live/meerkat_live.ex and write a handle_info/2 callback that handles the message. It should look like this:

```
def handle_info({:update, opts}, socket) do
  path = Routes.live_path(socket, __MODULE__, opts)
  {:noreply, push_patch(socket, to: path, replace: true)}
end
```

Our handle_info/2 callback doesn't do much. It generates a path with the new sorting parameters and uses push_patch/2 to live navigate to that path. This will trigger our handle_params/3 callback, with the new parameters. Let's see how we can parse these parameters and apply them when fetching the meerkat data.

Parsing and Assigning the Sorting Parameters

Our LiveView receives the sorting parameters in the handle_params/3 callback when the website is mounted or when the user changes the parameters through our SortingComponent.

As with all user input, we want to make sure that the received parameters are indeed valid. We don't want to build the validation ourselves though. Luckily Ecto.Changeset offers the functionality of parsing and validating the parameters for us. We'll use this functionality inside a schemaless changeset called SortingForm. Before we can create this form though, we have to take a small detour into the differences between a *schema* changeset and a *schemaless* changeset.

Using Ecto.Enum Inside a Schemaless Changeset

If you want to work with a database in your Elixir application, you'll likely use an Ecto.Schema for defining the fields and their types in your database schemas. Usually, a schema definition looks like this:

```
schema "my_models" do
  field :name, :string
  field :age, :integer
  field :status, Ecto.Enum, values: [:active, :inactive]
end
```

The preceding schema defines the fields and their type for a fictitious MyModel struct. Whenever you try to create such a struct, Ecto will check that your input can be converted into the specified type of the field. For example, the input %{"age" => "21"} is valid since "21" can be converted to an integer. However, the input %{"age" => "foo"} is invalid, since "foo" cannot be converted to an integer.

We want to use this type notation in our SortingForm as well. In particular, we want to define the valid values for our sort_by and sort_dir parameters as an Ecto.Enum. This way, we can check each input against a list of valid values for each parameter. However, our SortingForm doesn't correspond to a database schema. That is, we don't store its values in our database, but only use them in-memory. Therefore, we have to make it a schemaless changeset instead.

Schemaless changesets are Ecto.Changesets that don't use an Ecto.Schema to define the fields and the types of the data they validate. They validate the data against fields defined in regular Elixir structs or simple key-value maps. Whereas the purpose of regular changesets is usually to validate data before it's written to the database, schemaless changesets mostly validate user input coming from forms or URL parameters. Any data that doesn't correspond to a database model hence doesn't have an Ecto.Schema definition.

Unfortunately, the field :status, Ecto.Enum, values: [:active, :inactive] notation for Ecto.Enum typed fields cannot be used in schemaless changesets. Instead, we have to fall back to a general Ecto.ParameterizedType, which allows us to define any type of field also in a schemaless changeset. Its notation might look a bit wild, but the end result is the same. So, we wouldn't define an Ecto.Enum field like this:

```
field :sort_by, Ecto.Enum, values: [:id, :name]
```

Instead, we have to write this:

```
sort_by: {:parameterized, Ecto.Enum, Ecto.Enum.init(values: [:id, :name])}
```

This notation is a bit too complex and tedious to type out for every parameter we'll define. Let's create an EctoHelper module instead, which encapsulates this notation in a small helper function called enum/1. Open up the lib/meow/ecto_helper.ex file and type in the following:

```
defmodule Meow.EctoHelper do
  def enum(values) do
    {:parameterized, Ecto.Enum, Ecto.Enum.init(values: values)}
  end
end
```

Now, we can use the Meow.EctoHelper.enum/1 function to define Ecto.Enum fields also in a schemaless changeset. Let's use it to define the valid values for the sort_by and sort_dir parameters in our SortingForm.