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

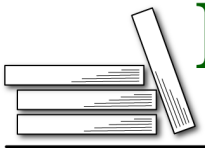
Peter Ullrich
Edited by Michael Swaine

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

Creating the LiveView

Let's create a new LiveView for implementing the infinity scrolling logic. Create a new file at lib/meow_web/live/infinity_live.ex and add the following code to it:

```
defmodule MeowWeb.InfinityLive do
  use MeowWeb, :live_view

  alias Meow.Meerkats

  def render(assigns) do
    ~H"""
    <table>
      <tbody id="meerkats"
             phx-update="append"
             phx-hook="InfinityScroll">
        <%= for meerkat <- @meerkats do %>
          <tr id={"meerkat-#{meerkat.id}"}>
            <td><%= meerkat.id %></td>
            <td><%= meerkat.name %></td>
          </tr>
        <% end %>
      </tbody>
    </table>
    """
  end
end
```

Let's have a look at the render/1 function. It generates a HEEx template, which displays a simple table that renders our meerkat data row by row. This isn't very different from our previous table UI, so let's have a closer look at the more interesting phx-update and phx-hook tags.

Instructing LiveView How to Handle New Data

The phx-update tag instructs Phoenix LiveView how to behave when our LiveView updates the data in the @meerkats assign. If you don't set this tag, it defaults to *replace*, which means that it will replace the current content of the table with the new data from the @meerkats assign.

However, this isn't quite what we want. When we load the next page of data, we want to keep the old content in the table and add the new content to the end of the table. We can instruct LiveView to do exactly that by setting the phx-update tag to *append*. Now, LiveView will add new rows to the end of the table and keep the existing ones. This works well for infinity scrolling because we can append any new content to the table before the user sees it. This way, we ensure that the user feels like the content never ends. Hence the *infinity*

in infinity scrolling. For completeness, let's have a look at the other possible values for the phx-update tag.

We could instruct LiveView to disregard any updates by setting this tag to *ignore*. Now, LiveView will render the table only once and ignore any updates to the @meerkats assign afterward. This can come in handy when your website needs to interoperate with JavaScript frameworks for, for example, showing alerts. Such frameworks typically create and manage the state of their own HTML elements. If LiveView were to replace these elements every time an update comes in, it could break the functionality that the framework provides. Typically, you would set the phx-update tag to *ignore* on such elements and handle any updates using LiveView Client Hooks. We'll talk about those later.

If you wanted to add new rows to the top instead of the end of the table, you could set the phx-update tag to *prepend*. This is useful if you want to show the latest updates always at the top of the table and let them supersede any previous messages. In that case, LiveView would keep the existing rows but push them down the table by adding the new rows to the top of the table.

When we want to append or prepend new rows to the table, we need to give each row a unique identifier. This way, LiveView can track which data entries already exist in the table and which it needs to add. We accomplish this by adding the id={"meerkat-#{meerkat.id}"} tag to every table row.

In our case, we can assume that the id of each meerkat is unique because we use their unique database identifier. If you want to use a different field as a row identifier, you need to make sure that it is unique, as well, and that you never show the same row twice. LiveView won't crash in that case, but the user's browser will log an error and LiveView's updating behavior will become unpredictable.

Finally, you might have spotted the phx-hook="InfinityScroll" tag already. With this, we define which LiveView Client Hook should be mounted to the table element. We'll dive into client hooks in the next section. For now, all you need to know is that this hook sends a *"load-more"* event to our LiveView whenever the user gets close to the bottom of the page. This causes our LiveView to fetch more meerkat data and append it to the table before the user reaches the end of it.

Next, let's define the mount/3 function for our LiveView. Add the following two functions underneath the existing render/1 function:

```
def mount(_params, _session, socket) do
  count = Meerkats.meerkat_count()
```

```
  socket =
    socket
    |> assign(offset: 0, limit: 25, count: count)
    |> load_meerkats()

  {:ok, socket, temporary_assigns: [meerkats: []]}
end

defp load_meerkats(socket) do
  %{offset: offset, limit: limit} = socket.assigns
  meerkats = Meerkats.list_meerkats_with_pagination(offset, limit)
  assign(socket, :meerkats, meerkats)
end
```

For our new LiveView, the mount/3 callback is rather simple: it first fetches the count of all meerkat data from the database, assigns default values for the offset and limit parameters, and fetches the first page of meerkat data using the load_meerkats/1 function. However, one interesting detail is the temporary_assigns: [meerkats: []] option, so let's have a closer look at it.

Optimizing Memory Consumption with Temporary Assigns

Let's imagine that a user opens our website and scrolls all the way to the bottom of the page. We would fetch and store all meerkat data in-memory until the user's session ends. Now, imagine that thousands of users access and scroll to the bottom of our website, all at the same time. Elixir is not a memory-heavy language, but even that won't save our server from running out of memory and crashing.

Additionally, we store all that meerkat data in-memory although we don't really need to. Once a data entry is rendered, we could just forget about it. Even if we needed a single data entry to handle subsequent user actions like, for example, updating the entry, we could always fetch it from the database, update it, and re-render its row in the table. So, storing all meerkat data in the LiveView's process would be wasteful.

Luckily, LiveView has our back. With the temporary_assigns option, we can instruct LiveView to discard any meerkat data and reset the @meerkats assign back to an empty list, once it has rendered the initial meerkat data. This way, we only store the subset of the meerkat data in-memory until the LiveView has rendered it and free up its memory allocation right after.

Now that we assign the meerkat data only temporarily, we can support significantly more users with the same amount of memory. If you still expect performance issues, because you initially load too much data or you load data too often when the user starts scrolling, you could tweak the limit parameter until you find a sweet spot.

If you set limit too low, you'll have to load data more often and its response time will matter more because the user might hit the bottom of your website before you render new content. If you set the limit too high, you'll have peaks in your memory consumption and might hit your memory threshold more often because you load larger chunks of data per user. You can find the sweet spot by load-testing your application using frameworks like wrt, k6, or Jmeter.

Now that we've defined the render/1 and mount/3 function of our LiveView, let's dive into how to load more meerkat data when the user starts scrolling.

As mentioned above, we'll use a LiveView client hook to notify the LiveView when it should load and render more meerkat data. The notification will be a *"load-more"* event that the client hook sends through the websocket connection to the LiveView. Let's create a handler for this event by adding the following function to our LiveView:

```
def handle_event("load-more", _params, socket) do
  %{offset: offset, limit: limit, count: count} = socket.assigns

  socket =
    if offset < count do
      socket
      |> assign(offset: offset + limit)
      |> load_meerkats()
    else
      socket
    end

  {:noreply, socket}
end
```

Let's dissect this function. First, we fetch the offset, limit, and count assigns from the socket. Then we add a small but important detail to our event handler: the offset < count check. This check allows us to stop fetching new data once we've exhausted all meerkat data in our database.

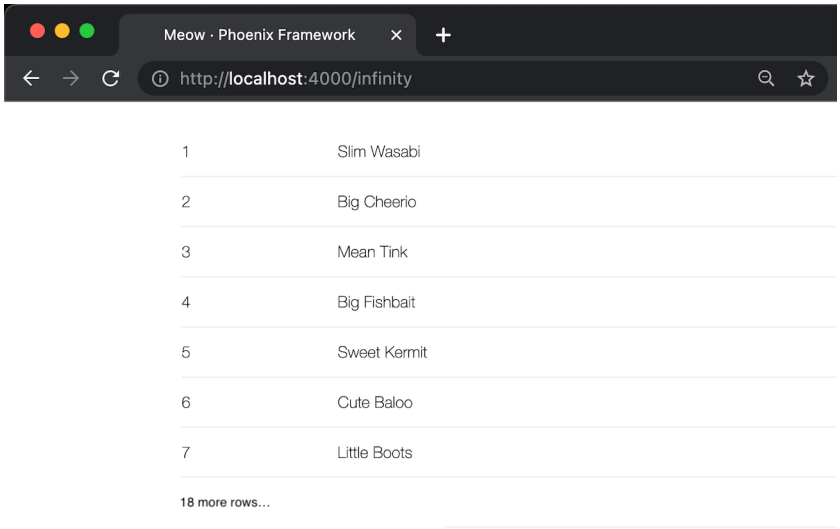
As you can see in the assign(offset: offset + limit) call, we increase our offset by our limit every time when we fetch new meerkat data. This means that at a certain point, our offset will be larger than the count of all meerkat data. When this happens, it would be pointless to keep on querying the database since no new data will be returned. That's why we add the offset < count check here. Once we've exhausted our meerkat data, our event handler will become a no-op and we don't query our database unnecessarily.

That's all the code needed on the LiveView side. Again, we've achieved so much with so little Elixir code! Now, to make our LiveView accessible, make sure to add it to lib/meow_web/router.ex, like this:

```
scope "/", MeowWeb do
  pipe_through(:browser)

  # Add the next line
  live("/infinity", InfinityLive)
  live("/", MeerkatLive)
end
```

Start the server with mix phx.server and navigate your browser to http://localhost:4000/infinity. You should see a table with 25 rows of meerkat data like this:



Try scrolling down the table. Unfortunately, no new meerkat data appears. This is because we haven't yet added the LiveView client hook that instructs the LiveView to load more data whenever the user gets close to the bottom of the table. Let's fix this.