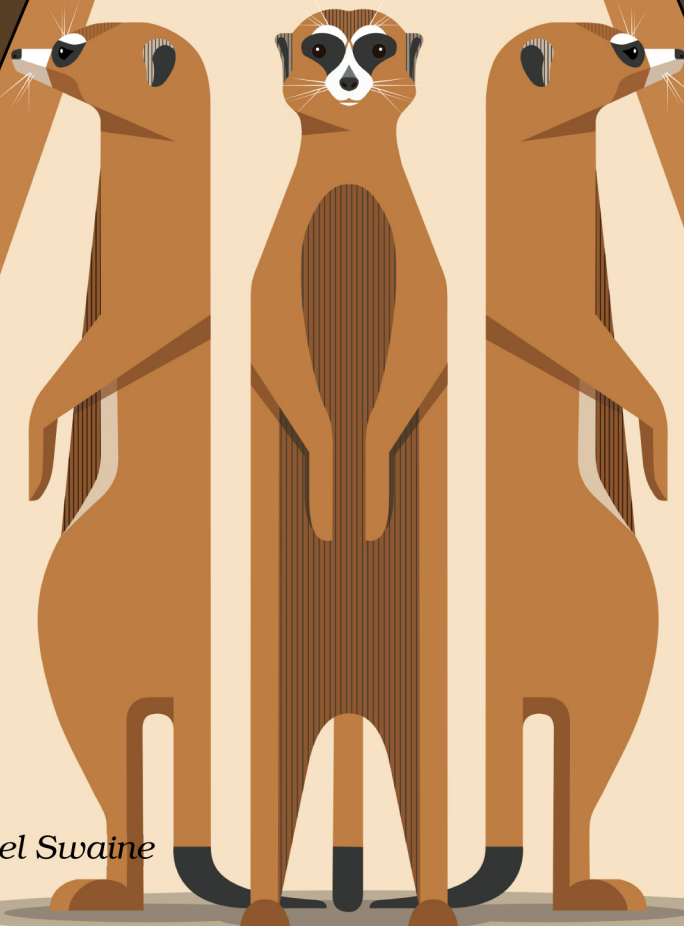


The
Pragmatic
Programmers

Building Table Views with Phoenix LiveView

*Advanced Table UIs
for Accessible Data*



Peter Ullrich
Edited by Michael Swaine

Early Praise for *Building Table Views with Phoenix LiveView*

Peter Ullrich's *Building Table Views with Phoenix LiveView* is a great, practical book. It will give the reader useful recipes on LiveView, filtering, pagination—all of it described in a clear and easy-to-follow format. A book that will get you excited about tables :D

► **Joel Carlbark**

Staff Engineer, Remote

This book covers a lot of ground in very few pages. Recommended for all levels of expertise with web development.

► **Vasilios Andrikopoulos**

Associate Professor, University of Groningen

Getting things done the right way with a new language and framework can be a daunting and time-consuming experience. Peter will be your guide in this book that will shorten the path that leads to an awesome user experience by building advanced table UIs with Elixir, Phoenix, and LiveView.

► **Pedro Gaspar**

Senior Software Engineer, Remote



We've left this page blank to make the page numbers the same in the electronic and paper books.

We tried just leaving it out, but then people wrote us to ask about the missing pages.

Anyway, Eddy the Gerbil wanted to say "hello."

Building Table Views with Phoenix LiveView

Advanced Table UIs for Accessible Data

Peter Ullrich

The Pragmatic Bookshelf

Raleigh, North Carolina



Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

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.

Copyright © 2023 The Pragmatic Programmers, LLC.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

ISBN-13: 978-1-68050-973-1

Encoded using the finest acid-free high-entropy binary digits.

Book version: P1.0—January 2023

Contents

| | | |
|----|---|-----|
| | Acknowledgments | vii |
| | Introduction | ix |
| 1. | Building a Simple Table UI | 1 |
| | Creating the Schema and Context | 1 |
| | Creating the LiveView | 2 |
| | Wrapping Up | 3 |
| 2. | Sorting the Table | 5 |
| | Sorting the Data in the Database | 5 |
| | Setting up LiveView | 6 |
| | Sorting with LiveComponent | 7 |
| | Parsing and Assigning the Sorting Parameters | 10 |
| | Putting the SortingForm to Use | 13 |
| | Wrapping up | 16 |
| 3. | Filtering the Table | 17 |
| | Filtering in the Database | 17 |
| | Creating the Filter LiveComponent | 20 |
| | Adding the Filter to the LiveView | 23 |
| | Wrapping Up | 26 |
| 4. | Paginating the Table | 29 |
| | Paginating in the Database | 29 |
| | Creating the Pagination LiveComponent | 34 |
| | Adding Pagination to the LiveView | 38 |
| | Wrapping Up | 41 |
| 5. | Paginating the Table Using Infinity Scrolling | 43 |
| | Setting up the Context | 44 |
| | Creating the LiveView | 45 |

Adding the LiveView Client Hook

49

Wrapping Up

57

Acknowledgments

I would like to thank the Elixir community for their endless support and inspiration. I thank my family and my reviewers for supporting and proofreading me. Lastly, I thank the band The Pineapple Thief for providing the album *Versions of the Truth*, the soundtrack to which I wrote this book.

I am sincerely thankful to my technical reviewers, who spent their precious free time and carried this book over the finish line. In particular, I would like to thank Svilen Gospodinov, Odhiambo Dormnic, Njoki Kiarie, Filipe Cabaço, Thiago Ramos, Osman Perviz, Sigu Magwa, Anthony Leiro, Pedro Gaspar, Joel Carlbark, Jacquie Kaunda, Vasilios Andrikopoulos, and Gregor Ihmor.

Peter Ullrich, 11 January 2023, Leiden, NL.

Introduction

There's a joke in web development that 50% of our work is building tables and the other 50% is building forms. This book is about the first 50%.

If you're an Elixir developer, chances are that you have to build tables. Lots and lots of tables. So, with table UIs making up such a significant part of our daily work, it's paramount to understand how to build them properly. This book is here to teach you exactly that.

In the following chapters we will see how to build a table UI with advanced features like sorting, filtering, pagination, and infinity scrolling. Using the marvelous Phoenix LiveView framework, we'll make them interactive. We'll work with LiveComponents and Ecto schemaless changesets to parse user input. We'll see how to organize our database operations using query composition and how to execute performance-heavy operations like sorting and filtering right in the database.

This book assumes that you know the basics of Elixir and how to set up a Phoenix LiveView application. If you're unfamiliar with this, first check out the Elixir "Getting Started" guide, <https://elixir-lang.org/getting-started/introduction.html>, and the Phoenix "Installation" documentation, <https://hexdocs.pm/phoenix/installation.html>.

You can find the complete codebase on GitHub here: <https://github.com/PJUlrich/ragprog-book-tables>. Feel free to download the code and play around with the application before diving into this book.

We have a lot of ground to cover, so open up your favorite code editor, turn to the next page, and let's get started!

Building a Simple Table UI

In the upcoming chapters, we'll solve a common problem when displaying data in a web application: our dataset is too large to show it all at once. Showing everything at once would cause significant performance issues both during the transport and the presentation of the data. Our user would have to wait forever until they see the data. It also prevents our user from narrowing down the dataset to the subsets they are most interested in.

We'll explore how to paginate, sort, and filter the dataset to break it into smaller, more digestible chunks. Our goal is to give the user full power over this functionality so that they can explore our dataset all on their own. Eventually, we want our user to be able to share their findings. Therefore, we'll build our solution in such a way that the user can share their view of the data by simply copy-pasting the URL from their browser.

The foundation for our exploration will be a demo application called the Meerkat Observation Warden, or short Meow. Its functionality is limited: it displays fictitious meerkat data.

Currently, the Meow application has a problem: it displays the entire meerkat dataset in a basic table UI. Our goal is to solve this problem by adding advanced features like pagination, sorting, and filtering to it. But first, let's explore the existing Meow application to understand where and how we can add our features.

Creating the Schema and Context

We begin with the core of our application: the Meerkat database schema. Open up `lib/meow/meerkats/meerkat.ex`, and you'll notice that the Meerkat schema holds only one attribute: the meerkat's name. We'll use this attribute and the auto-generated id of the meerkat to sort and filter the meerkat data later on.

```
defmodule Meow.Meerkats.Meerkat do
  use Ecto.Schema

  schema "meerkats" do
    field :name, :string
  end
end
```

Now, our Meerkat schema won't bring us far without the ability to fetch the existing meerkat data from the database. We want our Meow application to adhere to Phoenix's Model-View-Controller pattern. Therefore, we use the Meerkats context to execute database operations on the meerkat data. Open up `lib/meow/meerkats.ex`, and have a look at the `list_meerkats/0` function.

```
defmodule Meow.Meerkats do
  import Ecto.Query, warn: false

  alias Meow.Repo
  alias Meow.Meerkats.Meerkat

  def list_meerkats() do
    Repo.all(Meerkat)
  end
end
```

You'll notice that the function currently returns all meerkat data from our database, and that's the problem we'll solve. We want to narrow down our meerkat data based on the user's wishes, and this function will be our place to do just that. It will also be the location where we add the pagination of our data. You'll see that the Meerkats context will have our main focus when we implement the solution.

Creating the LiveView

The purpose of our Meow application is to display meerkat data to our user using a table UI. We want that display to be interactive and therefore use a Phoenix LiveView to add the reactivity to our UI. Open up `lib/meow_web/live/meerkat_live.ex`, and you'll see our basic MeerkatLive LiveView for displaying the meerkat data.

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

  alias Meow.Meerkats

  def mount(_params, _session, socket), do: {:ok, socket}

  def handle_params(_params, _url, socket) do
    {:noreply, assign_meerkats(socket)}
  end
end
```

```

defp assign_meerkats(socket) do
  assign(socket, :meerkats, Meerkats.list_meerkats())
end
end

```

As you can see, MeerkatLive loads our data using the Meerkats context and assigns it to the LiveView Socket under the `:meerkats` key.

Notice that we don't load the data in `mount/3` but in the `handle_params/3` callback. The reason for this is that we'll use live navigation for changing the sorting, pagination, and filtering of our data. This means that LiveView will call `handle_params/3` with updated parameters whenever we change one of these aspects. Every time that happens, we want to reload our data, which is why we use the `handle_params/3` callback for fetching and assigning the data.

Now, we need to display the data to the user. Since Phoenix 1.6, the recommended way of creating the HTML code for our user is using a `.heex` template. Open the `lib/meow_web/live/meerkat_live.html.heex` template and have a look at how we present the data to the user.

```

<table>
  <tbody>
    <%= for meerkat <- @meerkats do %>
      <tr>
        <td><%= meerkat.id %></td>
        <td><%= meerkat.name %></td>
      </tr>
    <% end %>
  </tbody>
</table>

```

We see that the template builds a simple HTML table in which every meerkat has its own row displaying its id and name attributes.

Wrapping Up

The problem we'll solve in the upcoming chapters is that we load and display *all* meerkat data in this table. As the meerkat data grows and grows, so too will our table. This will cause performance issues where the user will have to wait a long time before their browser displays the table. Also, it's difficult to filter the data for specific subsets. The sorting of the data is hardcoded as well. This prevents our users from changing the hierarchy of the data where, for example, larger values rank higher than lower values.

Next, we'll begin to improve the table UI by allowing the user to sort the data by ID or name, both ascending and descending. Let's go.

Sorting the Table

The first feature we're going to add to our table UI is sorting the data by each column, both ascending and descending. We'll leverage Phoenix LiveComponents and query composition for this task. Let's get started.

Sorting the Data in the Database

Before we add the sorting functionality to our LiveView, let's first explore how we can sort the data. Loading all meerkat data into memory and sorting it there is slow and will potentially exhaust our memory resources. Luckily, our Postgres database is heavily optimized for this use-case and allows us to add an ORDER BY SQL statement to our query. On the Elixir side, Ecto.Query provides the `order_by/2` function for adding this instruction. Let's see how we can make use of that function.

Open the Meerkats context in `lib/meow/meerkats.ex` and have a look at the `list_meerkats/0` function. Let's add the `order_by/2` function here. We want to make our sorting function reusable for other meerkat queries as well. Therefore, we'll use query composition to separate the database instructions into small and reusable functions. Here's the final code. We'll discuss it below:

```
def list_meerkats(opts) do
  from(m in Meerkat)
  |> sort(opts)
  |> Repo.all()
end

defp sort(query, %{sort_by: sort_by, sort_dir: sort_dir})
  when sort_by in [:id, :name] and
       sort_dir in [:asc, :desc] do
  order_by(query, {^sort_dir, ^sort_by})
end

defp sort(query, _opts), do: query
```

Have a look at the `list_meerkats/1` function. First, we create our query with `from(m in Meerkat)`. This allows us to pipe the query through our `sort/2` function and add more functions to the pipe later on.

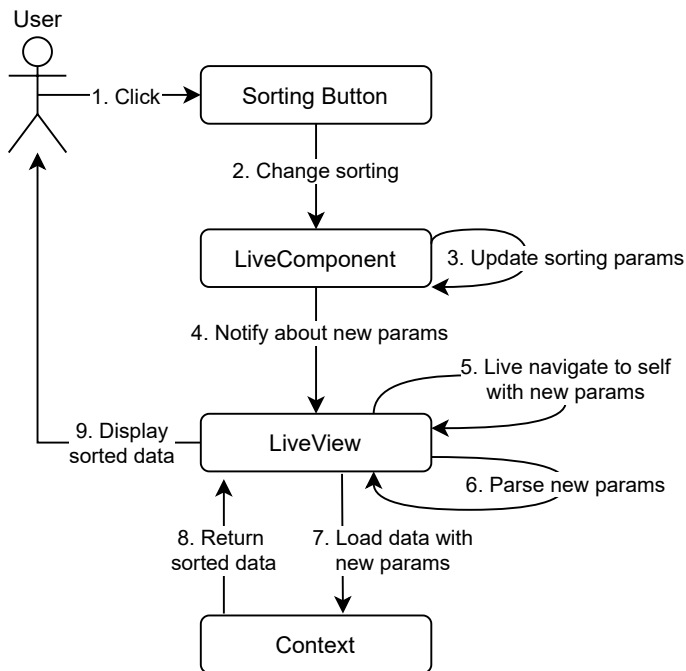
Now, have a look at the first `sort/2` function. You'll see that it receives the query and our sorting parameters. It pattern matches against the `sort_by` and `sort_dir` keys in the parameters. We verify with the `when` guard that we only pass valid sorting keys and sorting directions to the database. Eventually, we add the `order_by` statement to our query by providing a tuple with the sorting parameters.

In the case where our `opts` map contains no or invalid `sort_by` or `sort_dir` values, we fall back to the second `sort/2` method, which simply returns the query without adding any sorting to it. If you wanted to add a default sorting, which applies when no or invalid sorting parameters are given, you could add it here.

Now that we can sort the data when fetching it from the database, let's explore how we can give the user control over this new functionality by adding it to our `LiveView`.

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
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>
```

```
| <.live_component         module={MeowWeb.MeerkatLive.SortingComponent}         id={"sorting-id"}         key={:id}         sorting={@sorting} /> | <.live_component         module={MeowWeb.MeerkatLive.SortingComponent}         id={"sorting-name"}         key={:name}         sorting={@sorting} /> |
| --- | --- |

<!-- 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`.

Building a Schemaless Changeset

We want our `SortingForm` to take the received parameters and convert them into a valid map of values we can use for sorting the meerkat data. Open up `lib/meow_web/forms/sorting_form.ex` and type in the following. We'll discuss it later on.

```
defmodule MeowWeb.Forms.SortingForm do
```

```

import Ecto.Changeset

alias Meow.EctoHelper

@fields %{
  sort_by: EctoHelper.enum([:id, :name]),
  sort_dir: EctoHelper.enum([:asc, :desc])
}

@default_values %{
  sort_by: :id,
  sort_dir: :asc
}

def parse(params) do
  {@default_values, @fields}
  |> cast(params, Map.keys(@fields))
  |> apply_action(:insert)
end

def default_values(), do: @default_values
end

```

Since we use Ecto.Changeset, most of the parsing and validation happens under the (Ecto) hood. However, here's what happens in general terms:

The `parse/1` function receives all of our parameters, takes the `sort_by` and `sort_dir` parameters and converts their binary representation (for example, `%"sort_by" => "name"`) into an atom key-value pair (for example, `%{sort_by: :name}`). It checks whether the converted value exists in a list of valid values we define in the `@fields` map. Note that we used our `EctoHelper.enum/1` function here to define two Ecto.Enum typed fields.

If a value is received that is not in the list, `parse/1` returns `{:error, %Ecto.Changeset{}}`. Otherwise it returns `{:ok, %{sort_by: parsed_value, sort_dir: parsed_value}}`.

If one or both of the `sort_by` and `sort_dir` parameters are missing, it fetches a replacement value from the `@default_values` map and returns that one instead. This way, we always have a valid value for `sort_by` and `sort_dir` for sorting our data.

Testing the SortingForm

This might sound a bit complex, but it will become clearer once we've played around with it. So let's kick the tires and see how the `SortingForm` behaves when we throw parameters at it. Open an IEx session and follow along with these tests.

First, let's see what happens if we provide both `sort_by` and `sort_dir` parameters:

```
iex(1)> alias MeowWeb.Forms.SortingForm
```

```
MeowWeb.Forms.SortingForm
```

```
iex(2)> SortingForm.parse(%{"sort_by" => "name", "sort_dir" => "desc"})
{:ok, %{sort_by: :name, sort_dir: :desc}}
```

You can see that the parse/1-function took the binary values of the sort_by and sort_dir parameters and converted them into valid atom values. Great!

Now let's see what happens if one of the parameters is missing:

```
iex(3)> SortingForm.parse(%{"sort_by" => "name"})
{:ok, %{sort_by: :name, sort_dir: :asc}}
```

Aha! As expected, the parse/1 function filled in the value for sort_dir with the default value :asc. Also great!

Now, let's break something. Let's see what happens when we throw invalid values at it:

```
iex(4)> SortingForm.parse(%{"sort_by" => "foo", "sort_dir" => "bar"})
{:error,
 #Ecto.Changeset<
  action: :insert,
  changes: %{},
  errors: [
    sort_by: {"is invalid", [ ... ]},
    sort_dir: {"is invalid", [ ... ]}
  ],
  data: %{sort_by: :id, sort_dir: :asc},
  valid?: false
 >}
```

When the SortingForm cannot parse a parameter into a value from the acceptable values list, it marks that field as invalid and returns an error. Just as we expected!

It seems that our SortingForm works as expected, which is great! Now, we can use it in our LiveView to validate the sorting parameters coming in through the handle_params/3 callback.

Putting the SortingForm to Use

Now that we've created the functionality to parse and validate the input parameters, let's put it to use. Navigate back to the MeerkatLive module and make these changes as described here:

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

  alias Meow.Meerkats

  # Add this alias:
```

```

alias MeowWeb.Forms.SortingForm

def mount(_params, _session, socket), do: {:ok, socket}

# Update handle_params/3 like this:
def handle_params(params, _url, socket) do
  socket =
    socket
    |> parse_params(params)
    |> assign_meerkats()

  {:noreply, socket}
end

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

# Add this function:
defp parse_params(socket, params) do
  with {:ok, sorting_opts} <- SortingForm.parse(params) do
    assign_sorting(socket, sorting_opts)
  else
    _error ->
      assign_sorting(socket)
  end
end

# Add this function:
defp assign_sorting(socket, overrides \\ %{}) do
  opts = Map.merge(SortingForm.default_values(), overrides)
  assign(socket, :sorting, opts)
end

# Update assign_meerkats/1 like this:
defp assign_meerkats(socket) do
  %{sorting: sorting} = socket.assigns

  assign(socket, :meerkats, Meerkats.list_meerkats(sorting))
end
end

```

Okay, let's unpack this code step by step. First, we updated our `handle_params/3` callback to parse the parameters with `parse_params/2` and then fetch the meerkat data with `assign_meerkats/1`.

We use the `SortingForm` in our `parse_params/2` function to validate the sorting parameters and assign them to the socket if they are valid. If the `SortingForm` can't parse the parameters, we, for now, simply ignore the error and assign a default sorting using the `assign_sorting/1` function.

We updated our `assign_meerkats/1` function to fetch the current sorting parameters from the socket and pass them on to our Meerkats context. The context uses these parameters for sorting the data before it returns it to our LiveView, where we assign the data to our socket to update the UI.

With all these functions in place, we can now use the sorting elements in our table UI to update the sorting of the meerkat data.

Whenever we interact with the UI element of the `SortingComponent`, it updates the sorting parameters, notifies the LiveView through the `handle_info/2` callback, which then live navigates to a URL containing the updated sorting parameters. In our `parse_params/2` function, we use the `SortingForm` to validate these parameters and assign them to the socket. We then re-fetch the meerkat data with `assign_meerkats/1`, which fetches and passes on the sorting parameters to the `list_meerkats/1` function. Our Meerkats context applies the new sorting parameters when fetching the meerkat data from the database and returns the properly sorted meerkat data to the LiveView. Our LiveView assigns the new meerkat data to our socket, which causes a re-render of our table UI, which eventually displays the newly sorted data to the user.

Start the server with `mix phx.server` and play around with the new sorting elements. You should see something like this:



| id | name ↓ |
|----|---------------|
| 41 | Sweet Buttons |
| 75 | Sweet Bubbles |
| 62 | Sweet Baloo |

Note how we store the sorting parameters as URL parameters. Try refreshing the page or copy-pasting the URL into a new browser window. You should see that we always end up with the same view again.

We can now update the sorting of our meerkat data by interacting with the sorting UI elements. We can even copy and paste the URL from one browser to another and our LiveView will apply the same sorting. That's it, we're done!

Wrapping up

Phew! That was a lot! Take a second and pat yourself on the shoulder for following along until now. You managed to get through the most complex chapter of this book! This chapter introduced all the bits and pieces you'll need to add advanced features to your table UI. The following chapters will be a lot lighter since they build upon and extend these components. So, if you've understood the concepts so far, the rest should be smooth sailing!

Moving forward, you probably already spotted the next problem with our table UI: we still load all meerkat data from our database and present it in our table. Next, let's make the data filterable so that the user can focus on the data they really need.

Filtering the Table

Building a table UI is great for presenting your data to the user, but it can also be overwhelming if too much data is presented at once. Users are typically only interested in a subset of the entire dataset. Our job as developers is to provide them with the tools they need to get their work done. Filtering a table is one of these tools.

In the following chapter, we'll explore how to add filtering to our meerkat data and how to make the user's filter shareable. We'll use the building blocks we introduced in the previous chapter for this. So, most of what you'll see will be familiar to you, just applied in a slightly different way. Let's go!

Filtering in the Database

Let's start with adding filters to our meerkat data by extending our Meerkat context. We want to add filters for the `id` and the `name` fields of a meerkat. The `id` filter will be a distinct filter that matches only the meerkat with the exact `id`. The `name` filter will be a fuzzy filter that matches any meerkat whose name contains a certain string.

So, when we filter by an ID like 123, only the meerkat with the ID 123 will match our query. However, when we filter by name with, for example, the search term `big`, we want to find meerkats that have names like `Mr. Big`, `Bigly Ticky`, or `Overbig Snoutly`. As you can see, we want to match names that contain the string `big` regardless of its position in the name or its case. `Ecto.Query` offers the `ilike/2` method for exactly this use case. It searches for a string in a case insensitive fashion. So, both `Bigly` and `Overbig` match the query, although the `b` is sometimes uppercase and sometimes lowercase.

Now that we know what we want and how to achieve it, open up Meerkats context in `lib/meow/meerkats.ex`. This is our starting point. Let's change it in the following way:

```
# Change this function like so:
def list_meerkats(opts) do
  from(m in Meerkat)
  |> filter(opts)
  |> sort(opts)
  |> Repo.all()
end

# Add the following functions:
defp filter(query, opts) do
  query
  |> filter_by_id(opts)
  |> filter_by_name(opts)
end

defp filter_by_id(query, %{id: id}) when is_integer(id) do
  where(query, id: ^id)
end

defp filter_by_id(query, _opts), do: query

defp filter_by_name(query, %{name: name})
  when is_binary(name) and name != "" do
  query_string = "%#{name}%"
  where(query, [m], ilike(m.name, ^query_string))
end

defp filter_by_name(query, _opts), do: query
```

Let's walk through the code step by step. First, we add a `filter/2` function that extends our query with filters for ID and name. We move each filter into its own function so that we can reuse them and to make our code cleaner. The `filter_by_id/2` function is straightforward, as it only adds a `where/2` clause filtering by the id of a meerkat.

The `filter_by_name/2` function is a bit more complex. First, we create a `query_string` that surrounds our search term with percent characters. This means that we match any name that contains the search term *at any position*. It gives us the most results but might be less efficient since the database has to scan every name in its entirety to check for matches (unless you use fancy indexing, but that's for another time).

A more typical use-case is to match names that *start* with a given search term. In this case, we add a percent character after the search term like this: `"#{name}%"`. This will match names that begin with a search term like Bigly

Tickly but not names that have the search term anywhere like Overbig Snouty or Mr. Big.

After creating our query string, we add another `where/3` statement to our query which uses the `ilike/2` function to search for our search term case insensitively. And that's it! With only a few functions, we added filtering to our meerkat query. Marvelous!

A Note on Query Composition

As you can see, we use query composition again to add the `filter_by_id/2` and `filter_by_name/2` functions. We use this method because it keeps our code open for extension but reduces the potential need for modification. This means that we can easily add more filters and reuse existing ones but that we don't have to modify existing filters when we make changes like adding fields to the Meerkat schema or rename the database table.

Note also that we don't make any assumptions about the schema for which we query in the `filter_by_id/2` and `filter_by_name/2` functions. Technically, we can use these filters to query for any schema, as long as it has an `id` or a `name`.

So, when you write your next Ecto context, consider splitting your queries into smaller functions and use query composition to combine them into larger queries. This way, you'll avoid duplication of your code and make your queries a bit more readable.

However, in software engineering, very few upsides come without a downside; this approach also has a potential downside. If you split your queries into many smaller functions and use them to compose larger queries, you increase the dependencies between your larger queries since they use the same building blocks. As an example, if you have two large queries that both use the `filter_by_name/2` function, you can't change it without affecting both large queries. If one large query wants to query case insensitively but the other one doesn't, then you have a conflict.

To solve this, you could add a Boolean flag to your `filter_by_name/2` method and change the case sensitivity based on it. You could also create two functions that filter by name, one case sensitively and the other one case insensitively. However you solve this conflict, your code always becomes more complex and harder to maintain.

You could avoid the conflict altogether by not using query composition. Instead, you could have dedicated functions that are large and complex but self-contained and independent of other functions. They would serve a very

specific purpose and might be large and complex, but they won't have to change if other functions change.

Sometimes this is the right way to go, especially when you have many large queries that all use the same building blocks. So, to sum it up: be mindful when using query composition and don't use it blindly.

Alright! Now that we're able to filter by ID and name when fetching the meerkat data, let's expose this functionality to our users by adding it to our LiveView.

Creating the Filter LiveComponent

As with our sorting functionality, we'll encapsulate the filter logic into its own LiveComponent called FilterComponent. It will contain a simple form for receiving the user input and use another schemaless changeset for parsing said input. Create a file at `lib/meow_web/live/filter_component.ex` and key in the following code:

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

  alias MeowWeb.Forms.FilterForm

  def render(assigns) do
    ~H"""
    <div>
      <.form let={f} for={@changeset} as="filter"
        phx-submit="search" phx-target={@myself} >
        <div>
          <div>
            <%= label f, :id %>
            <%= number_input f, :id %>
            <%= error_tag f, :id %>
          </div>
          <div>
            <%= label f, :name %>
            <%= text_input f, :name %>
            <%= error_tag f, :name %>
          </div>
          <%= submit "Search" %>
        </div>
      </.form>
    </div>
    """
  end

  def update(%{filter: filter}, socket) do
    {:ok, assign(socket, :changeset, FilterForm.change_values(filter))}
  end

  def handle_event("search", %{"filter" => filter}, socket) do
    case FilterForm.parse(filter) do

```

```

{:ok, opts} ->
  send(self(), {:update, opts})
  {:noreply, socket}

{:error, changeset} ->
  {:noreply, assign(socket, :changeset, changeset)}
end
end
end

```

Let's go through this code top to bottom. First, we create a HEEEx template which holds a simple HTML form. The form uses an Ecto.Changeset for populating and submitting the form data. When the user submits the form, we send a "search" event back to the LiveComponent.

We use the Phoenix.HTML.Form helper functions to create input fields for the id and name search terms. For the id field, we use a number input. This allows us to use HTML5 client-side validation, which checks the input data right there in the browser even before the user sends it to the LiveComponent. The name field is a simple text field that accepts any text input. But enough about the HTML form. Let's move on to the `update/2` and `handle_event/3` functions.

Every LiveComponent has a default `update/2` function, which merges all assigns into the socket. This is fine for most use-cases, but we want to convert the filter parameters into an Ecto.Changeset before rendering the template. This allows us to easily show error messages for each field using the `error_tag/2` element.

For example, we want to disallow negative values for the id field since our database IDs are always greater or equal to 0. If the user enters a negative value for the id field, we want our Ecto.Changeset to flag it as invalid and provide a simple error message which the `error_tag/2` then presents to the user.

That's why we override the default `update/2` function of the LiveComponent with our own. This way, we can fetch the filter assign and convert it into an Ecto.Changeset using the `FilterForm`, which we will get into in a moment. Eventually, we assign the Ecto.Changeset to the `@changeset` assign and use it in the HTML form as described before.

Next, let's look at the `handle_event/3` function. When the user submits the form, our LiveComponent receives the "search" event with the form data stored in the "filter" parameter. We fetch the data and parse it using the `FilterForm`. If the data can be parsed correctly, we instruct the LiveView to update the parameters with the `send(self(), {:update, opts})` call, just as we did in the `SortingComponent`.

If our `FilterForm` cannot parse the form data correctly, we assign the returned `Ecto.Changeset` to our socket and let the HTML form render the error messages which the `changeset` contains.

So, to sum it up: a user can enter filter parameters, and we parse them using the `FilterForm` and instruct the `LiveView` to update the URL parameters accordingly. This causes the `LiveView` to refetch the meerkat data and apply the new filter parameters while doing so. That's the gist of what the `FilterComponent` does.

Next, let's have a look at the `FilterForm` we mentioned before. Open up `lib/meow_web/forms/filter_form.ex` and enter the following code:

```
defmodule MeowWeb.Forms.FilterForm do
  import Ecto.Changeset

  @fields %{
    id: :integer,
    name: :string
  }

  @default_values %{
    id: nil,
    name: nil
  }

  def default_values(overrides \\ %{}) do
    Map.merge(@default_values, overrides)
  end

  def parse(params) do
    {@default_values, @fields}
    |> cast(params, Map.keys(@fields))
    |> validate_number(:id, greater_than_or_equal_to: 0)
    |> apply_action(:insert)
  end

  def change_values(values \\ @default_values) do
    {values, @fields}
    |> cast(%{}, Map.keys(@fields))
  end
end
```

As you can see, the `FilterForm` is not too different from the `SortingForm` which we created before. First, we specify a set of fields and their type. Then we define their default values, which in this case are just `nil` since we don't want to apply any filtering by default. We expose the default values with the `default_values/1` function. Our `LiveView` will use this function to set the default filter parameters when it is first created.

Next, we define the `parse/1` and `change_values/1` functions. The `parse/1` function casts the given parameters into a schemaless `changeset` and verifies that the

id is greater than or equal to 0. By applying the `:insert` action, we instruct `Ecto.Changeset` to verify that the casting and the validations were successful. It will either return a map with the parsed parameters or an `Ecto.Changeset` that contains error messages that explain why we couldn't parse or validate the parameters.

Remember that we want to convert the filter parameters to an `Ecto.Changeset` before rendering the template. That is what the `change_values/1` function is for. It takes a map of values and converts them into an `Ecto.Changeset`. Unlike in the `parse/1` function, where we override the default values with values from the `params` map, we don't want to make any changes to the values map in `change_values/1`. That's why we use an empty map in the `cast/3` function as a second argument. This way, we convert the values into an `Ecto.Changeset` without overriding any of them.

Alright, that's all there is to know about the `FilterComponent` and the `FilterForm`! Now, let's get to the juicy bits: using the filter parameters in the LiveView.

Adding the Filter to the LiveView

To add the `FilterComponent` to our LiveView, open up the `HEEx` template at `lib/meow_web/live/meerkat_live.html.heex` and add the following code:

```
<div>
  <!-- Add this .live_component element -->
  <.live_component
    module={MeowWeb.MeerkatLive.FilterComponent}
    id="filter"
    filter={@filter} />

  <!-- Table element stays here -->
</div>
```

Now, we render the `FilterComponent` above our table. However, if you try to start the server, you'll see an error message since we still have to add the `@filter` assign to the socket. Let's fix that by assigning the default filter parameters. Open the LiveView at `lib/meow_web/live/meerkat_live.ex` and add the following code:

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

  # Other aliases omitted

  # Add this alias
  alias MeowWeb.Forms.FilterForm

  # mount/3 and handle_params/3 omitted

  # Update parse_params/2 like this:
  defp parse_params(socket, params) do
```



```

with {:ok, sorting_opts} <- SortingForm.parse(params),
     {:ok, filter_opts} <- FilterForm.parse(params) do
  socket
  |> assign_filter(filter_opts)
  |> assign_sorting(sorting_opts)
else
  _error ->
    socket
    |> assign_sorting()
    |> assign_filter()
end
end

# assign_sorting/2 omitted

# Add the following function
defp assign_filter(socket, overrides \\ %{}) do
  assign(socket, :filter, FilterForm.default_values(overrides))
end
end

```

Let's first have a look at the updated `parse_params/2` function. We added `FilterForm.parse/1` here to parse any filter-related URL parameters provided by the `params` map. Our `FilterForm` will cast and validate any given `id` and `name` parameters and set default values (in this case `nil`) for those parameters that weren't provided.

If the `FilterForm` can parse the parameters successfully, we assign the result to our `socket` with the `assign_filter/2` helper function. If the parsing fails, for example, because a negative `id` was entered, we catch (and ignore) the error in the `else` clause and simply assign the default filter values as returned from `FilterForm.default_values/1`. This way, we always assign valid filter parameters, even when invalid URL parameters were entered.

This approach has the advantage that we can assume that we always receive valid `@filter` values in our `FilterComponent`. But it also creates a worse UX since we don't inform the user that the entered parameters are invalid and that we assigned default values instead. You can solve this problem easily by handling the error in `parse_params/2` and, for example, assigning an `:error` flash message to the `socket`. Proper error handling is a topic on its own, and it's up to you to find a solution that suits you best. However you approach this problem, though, the `parse_params/2` function would be a good place to start.

Now, start the server again with `mix phx.server`. You should see the new HTML form above the table now. Try entering a filter parameter and press *Search*. If you followed the instructions closely, you should see that we filter the data

correctly. Hurray! Try entering a negative id and you should see an error message. Double hurray!

But now, try sorting the table, for example, descending by id and add a filter. Sadly, this removes the sorting and only applies the filter to the data. Also, if you enter a single filter like, for instance, for an id, you'll see that the URL also receives a name parameter that is empty. There are two problems here: first, we don't merge all entered parameters from the sorting and the filter component before updating the URL parameters. Second, we don't remove empty parameters. Thankfully, these are easy problems to fix, so open up the LiveView again and update its code as follows:

```
# Update handle_info/2 like this:
def handle_info({:update, opts}, socket) do
  params = merge_and_sanitize_params(socket, opts)
  path = Routes.live_path(socket, __MODULE__, params)
  {:noreply, push_patch(socket, to: path, replace: true)}
end

# Update assign_meerkats/1 like this:
defp assign_meerkats(socket) do
  params = merge_and_sanitize_params(socket)

  assign(socket, :meerkats, Meerkats.list_meerkats(params))
end

# Add this function:
defp merge_and_sanitize_params(socket, overrides \\ %{}) do
  %{sorting: sorting, filter: filter} = socket.assigns

  %{
    |> Map.merge(sorting)
    |> Map.merge(filter)
    |> Map.merge(overrides)
    |> Enum.reject(fn {_key, value} -> is_nil(value) end)
    |> Map.new()
  }
end
```

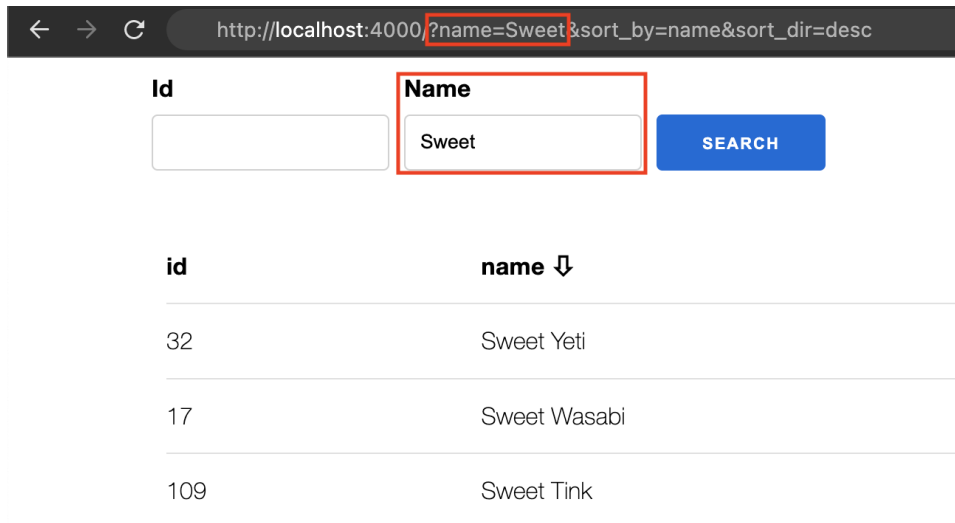
As you can see, the major change here is that we add a new function called `merge_and_sanitize_params/2` to both `handle_info/2` and `assign_meerkats/1`. As the name suggests, `merge_and_sanitize_params/2` merges and sanitizes our parameters. It retrieves all active sorting and filtering parameters, merges them, and removes any empty parameters from the mix.

We use `merge_and_sanitize_params/2` in `handle_info/2` to assign *all* parameters to the URL whenever *any* parameter changes. This solves our first problem, which was that we removed the sorting parameters when we applied new filter parameters.

We also use `merge_and_sanitize_params/2` when fetching the meerkat data from the database in `assign_meerkats/1`. If we don't add this call here, we still set all parameters in the URL but only apply some of them when actually fetching the data.

Since we remove any empty parameters after merging them, we solved our second problem, which was that we assigned empty parameters to the URL. You can try this out if you restart the server, set a filter, and remove the filter again. You should see that we first add the filter to the URL parameters but then remove it again when you leave its input field empty. Triple hurray!

Play around with the filter inputs a bit. You should see something like the following picture. Note how we store the sorting and filtering parameters as URL parameters. This way, the user can copy, share, and bookmark the URL and end up with exactly the same view again.



Wrapping Up

Again, great job for following along so far! We successfully added filtering to our table UI! Even more so, we extended the building blocks from the previous chapter and combined the sorting and filtering functionality in our LiveView with only a few lines of code. We encapsulated the new filtering logic into its own LiveComponent so that we can reuse or remove it easily in the future. We merged and updated all parameters in the URL and applied them when fetching the data. Great job!

One last problem we still need to fix, however, is that we fetch and present *all* meerkat data in our database. A common solution to this problem is to

paginate the data, and this is exactly what we'll look at in the next chapter, so stay tuned!

Paginating the Table

One of the last advanced functionalities that we'll add to our table UI is the pagination of the meerkat data. Instead of showing all data at once, we'll only show parts of the data, one at a time. This improves the loading performance and overall usability of our table.

Pagination means that we chunk our data into *pages* based on a *page size*. As an example, if we have 100 entries of meerkat data and a page size of 20, we chunk the data into five pages of 20 entries each. If the number of entries isn't divisible by the page size, the last page will contain the remainder of entries. For example, with 90 entries and a page size of 20, we have four pages with 20 entries each and a fifth page with only 10 entries.

The user can step through the pages using a *page stepper* and change the number of entries we show by adjusting the *page size*. As before, we'll store the page size and current page as URL parameters. This allows our users to share their current page with others and to bookmark them. So, without further ado, let's start paginating our meerkat data!

Paginating in the Database

As usual, we start with adding the new functionality to our Meerkats context. Open up `lib/meow/meerkats.ex` and add the following code:

```
def list_meerkats_with_total_count(opts) do
  query = from(m in Meerkat) |> filter(opts)

  total_count = Repo.aggregate(query, :count)

  result =
    query
    |> sort(opts)
    |> paginate(opts)
    |> Repo.all()
```

```

    %{meerkats: result, total_count: total_count}
  end

  defp paginate(query, %{page: page, page_size: page_size})
    when is_integer(page) and is_integer(page_size) do
      offset = max(page - 1, 0) * page_size

      query
      |> limit(^page_size)
      |> offset(^offset)
    end

  defp paginate(query, _opts), do: query

```

As you can see, we create a new `list_meerkats_with_total_count/1` function instead of extending the existing `list_meerkats/1` function. We could also rewrite the existing `list_meerkats/1` function; however, since we change the return type of the function, we'll create a new function instead. We return the total count of all entries affected by the query since we need this information for our pagination to work. Let's take a step back and understand why this is necessary.

In the UI, we want to show how many pages our data has so that the user can click through them. To do so, we need to calculate the total number of pages so that we can show one button per page. We can calculate the number of pages if we know the total count of the entries and the current page size. The total number of pages is the total count divided by the page size, rounded up. So, when we have 90 entries and a page size of 20, it's 4.5 rounded up, so 5 pages in total. Later on, we'll see how we can adjust the page size in the UI, but we need to calculate the total count in our Meerkats context whenever we fetch the meerkat data.

That's what our new `list_meerkats_with_total_count/1` function is for. It sorts, filters, and fetches our meerkat data just as before, but it also returns the total count of all entries that match our query. Additionally, it only returns the number of entries that we specify with the page size. So, instead of returning *all* meerkat data, it only returns, for example, 20 rows, which is exactly what we want to achieve with our pagination.

A Small Deep Dive into How Pagination Works

We implement the pagination by adding the `limit/2` and `offset/2` statements to our query. The `limit/2` call instructs the database to return only the number of entries we specify. The `offset/2` statement instructs the database from where it should count the number of entries to return. As an example, if we set the offset to 10 and the limit to 20, the database will return 20 entries counting from the eleventh row, effectively ignoring the first 10 entries.

This is how we achieve the pagination. We calculate the offset by multiplying the current page number by the page size. In the UI, we present the page numbers as starting from 1, but in our database, the first page is page 0. That's why we subtract 1 from the page number before calculating the offset. Let's look at an example:

Let's say we want to fetch the data for the first page with a page size of 10. We first subtract 1 from the page number, receiving a new page number of 0. Then we multiply the 0 with our page size of 10, and we get an offset of 0. This means that the database will return 10 entries beginning with the very first row, which is exactly what we want.

Now, let's say we want the data for the second page. We subtract 1 from the page number of 2, resulting in a new page number of 1. We multiply it with our page size of 10 and get an offset of 10. This means that the database will skip the first 10 entries and return entries from the eleventh row onward. So, for our second page, we return the second chunk of data, which is exactly what we want as well.

As a safety precaution, we use the $\max/2$ function to ensure that our page number is always equal to or greater than 0. In case we make a mistake somewhere and set the page number to 0 or even -1, this will fix the mistake by replacing it with 0.

A Short Comparison of Offset vs. Cursor Pagination

The pagination approach just described is called *offset pagination*. It's a popular approach to pagination because it's relatively simple and easy to understand. It lets the user jump to any page in the data. However, for very large datasets (we're talking multiple millions of rows here) and for real-time data presentation, offset pagination is unsuitable. The reason is that the database needs to count many rows to fetch, for example, the ten-millionth row.

The time complexity of offset pagination is $O(\text{offset} + \text{limit})$, which means that it has *linear complexity*. So, if you have a large dataset and you need to fetch the last page of the data frequently, offset pagination might not be efficient enough for your use-case since it will take potentially very long until your query returns the data.

Given this drawback, offset pagination is unsuitable for real-time data presentation where new data is continuously appended to the table and the most recent data is fetched often. Your database needs to count and ignore many

rows every time you fetch the latest data and the number of counted rows increases as new data is added.

Another problem with offset pagination is that you might return repeated rows to the user if you add data to a previous page. As an example, imagine that we sort the meerkat data by the meerkat's name and we chunk the data into pages of ten rows each. Furthermore, imagine that the first page has ten rows of names beginning with *A* and the second page has ten rows with names beginning with *B*. So, if the user request the first page, they will see ten names beginning with *A*.

Now, imagine we add another meerkat whose name begins with an *A* while the user views the first page in their browser. Now, if the user moves to the second page, suddenly they see a page containing one name beginning with *A* and nine names beginning with *B*! Worse even, they might have seen the name beginning with *A* on the previous page already! This happens because the second page returns the tenth to the twentieth entries, regardless of which data you presented previously. Since we now have eleven names with *A* and ten with *B*, it returns the eleventh name with *A* and nine names with *B*. Hence, the user see a repeated row with *A*, which might cause an inferior user experience.

So, if you have a very large dataset, need to fetch rows at the end of the table often, and add data anywhere in the dataset frequently, offset pagination might not be for you. In such cases, you might want to look into *cursor pagination* instead.

In short, cursor pagination uses *previous* and *next* pointers to indicate the upper and lower bounds of the current page. As an example, imagine you are on the third page of the meerkat data and use the *id* field to paginate the data. In this case, your previous, or lower bound, pointer would be 20 and the next, or upper bound, pointer would be 29 because we are 0-based here. So, how can you use these pointers to request the previous or the next page?

Fetching the next page is relatively simple. The query would look something like this:

```
from(m in Meerkat,
     where: m.id > ^29,
     order_by: [asc: :id],
     limit: 10
  )
```

So, using the next pointer, we can simply fetch the next page by filtering out any meerkat data with an *id* lower or equal to the next pointer, which is 29

in our case. This way, we fetch the thirtieth entry up to and including the thirty-ninth entry, which is equal to the fourth page of our data. So far, so good. However, fetching the previous page is slightly more complicated. Let's have a look at how a query could look:

```
from(m in Meerkat,
     where: m.id < ^20,
     order_by: [desc: :id],
     limit: 10
)
|> Repo.all()
|> Enum.sort_by(& &1.id, :asc)
```

First, we filter out any data with an id equal to or higher than our previous pointer, which is 20 in our case. We receive a dataset of twenty entries with IDs from 0 to 19. So far so good. However, we want to fetch the tenth until the nineteenth entry, which is why we sort the entries descending by their ID. This results in a dataset where the entry with the ID 19 comes first, followed by 18, 17, and so on. From that dataset, we then take ten entries as indicated by the `limit: 10` instruction. This returns the entries with an ID from 19 down to 10, which is the data we expect in the second page of our dataset. However, the data is sorted descending by ID, which is why we inverse the order using `Enum.sort_by/3` before we return the data from the function.

Phew! As you can see, implementing a cursor-based pagination is definitely more complex than the offset-based alternative. But the big advantage of cursor-based pagination is that it uses the `where` instruction instead of the `offset` instruction. The advantage of `where` over `offset` is that Postgres efficiently ignores any rows that don't match the `where` statement, whereas it counts and skips the rows before the given offset. If the offset is high, Postgres counts every single row until the offset, which can result in a full-table scan in the worst case. With `where`, it first filters out any unsuitable rows before starting to count. This results in a much more efficient query.

However, the efficiency gains of cursor pagination largely depend on properly configured indices for the variables that you want to use for pagination. Our primary key `id` already has an index, but if you wanted to paginate using the `name` variable, you would have to create the proper index yourself. Otherwise, Postgres still needs to check every row whether it matches the `where` statement or not, and your cursor pagination won't be better than a simple offset pagination.

Cursor pagination solves the problem with showing repeated entries, as well, since it doesn't rely on offsets but uses pointers to its upper and lower bound

entries instead. In our previous example, the upper bound of the first page would be the last name with *A*. So, when the user requests the second page, our query would only show names beginning with *B* since $B > A$. Even if we added another meerkat with a name beginning with *A* in the meantime, the second page would only show entries *after* the last name with *A*, so all names with *B*.

The big downside of cursor pagination, aside from its implementation complexity, is that users can't jump to specific pages in the dataset. Instead, they have to browse through every single page before the page that they want to access. This is because we don't have an *offset* anymore but only upper and lower bounds pointers. If this is a common use-case for your users, consider using offset pagination instead.

In summary, for small datasets whose last page is rarely accessed, the offset pagination is a simple and suitable solution. If your dataset is very large and you need to access its latest data frequently, a cursor-based pagination might be more suitable. In this book, we'll implement an offset-based pagination since it's easier to understand and implement. Also, once you have the offset pagination in place, it's a relatively small step to switch to cursor-based pagination.

Alright! Now that we discussed how the pagination works on the database side, let's see how we can expose the pagination functionality to the user. As usual, we'll create a LiveComponent to encapsulate the UI logic. Let's see what that looks like.

Creating the Pagination LiveComponent

Just as with sorting and filtering, we want to create a dedicated LiveComponent for the pagination functionality. Open up `lib/meow_web/live/pagination_component.ex`. Since the `PaginationComponent` is slightly larger than the previous LiveComponents, we'll add the code in two steps instead of one. Please key in the following code first:

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

  alias MeowWeb.Forms.PaginationForm

  def render(assigns) do
    ~H"""
    <div>
      <div>
        <%= for {page_number, current_page?} <- pages(@pagination) do %>
          <div phx-click="show_page"

```

```

        phx-value-page={page_number}
        phx-target={@myself}
        class={if current_page?, do: "active"} >
      <%= page_number %>
    </div>
  <% end %>
</div>
<div>
  <.form let={f}
    for={:page_size}
    phx-change="set_page_size"
    phx-target={@myself} >
    <%= select f, :page_size,
      [10, 20, 50, 100],
      selected: @pagination.page_size %>

  </.form>
</div>
</div>
"""
end
end

```

Let's have a look at what the `PaginationComponent` renders. First, we add a page stepper, which the user can use to step through the pages. We render one button per page. You might want to change this if you expect your data to have hundreds or more pages. In that case, you could only show the first five and the last five pages.

We generate a list of page numbers using the `pages/1` helper function. In short, it uses the `page`, `page_size`, and `total_count` for calculating how many pages we need to show and which one of them is the currently selected one. This will become clearer when we add the code in the next step.

When generating the page buttons, we add an `.active` class to the button that represents the currently selected page. This helps the user to keep track of which page they are on. When a user clicks a page button, we send a `show_page` event to the LiveComponent with the page number as a parameter. Don't worry about how we handle this event for now. We'll add its handler in the next step.

Below the page stepper, we show a dropdown element for selecting a page size. Theoretically, we could allow the user to specify any page size, but for UX reasons, we provide a list of predefined page sizes of 10, 20, 50, and 100 entries. When the user selects one of these options, we send a `set_page_size` event to the LiveComponent.

Alright, now that we've discussed the UI elements of the `PaginationComponent`, let's have a look at the event handlers and at the `pages/1` helper function. Please add the following code below the `render/1` function:

```
<!-- render/1 stays here -->

def pages(%{page_size: page_size, page: current_page,
  total_count: total_count}) do
  page_count = ceil(total_count / page_size)

  for page_number <- 1..page_count//1 do
    current_page? = page_number == current_page

    {page_number, current_page?}
  end
end

def handle_event("show_page", params, socket) do
  parse_params(params, socket)
end

def handle_event("set_page_size", %{"page_size" => params}, socket) do
  parse_params(params, socket)
end

defp parse_params(params, socket) do
  %{pagination: pagination} = socket.assigns

  case PaginationForm.parse(params, pagination) do
    {:ok, opts} ->
      send(self(), {:update, opts})
      {:noreply, socket}
    {:error, _changeset} ->
      {:noreply, socket}
  end
end
```

Let's have a look at the `pages/1` helper function first. It generates a list of page numbers using the total count and the page size and provides an indicator whether a page is the current page or not.

First, it calculates the total number of pages by dividing the total count by the page size. We always want to create a last page that includes the remainder of the division, which is why we use the `ceil/1` function. For example, if we have 90 entries with a page size of 20, the `ceil/1` function makes sure that we show five pages (4.5 rounded up) instead of four. The fifth page then contains only the last 10 entries.

After calculating the page count, we generate a range of page numbers using the range stepper `page_count//1`. This prevents us from showing two pages when the page count equals 0. In that case we would generate a range from 1 to 0,

which means that Elixir creates a list of two descending numbers [1, 0]. However, since we use the positive range stepper `page_count//1`, we make sure to only generate ascending numbers. So, if the page count equals 0, we generate an empty list instead, which is exactly what we want.

Next, you see that we define two event handlers for the `show_page` and `set_page_size` events. They both call the `parse_params/2` function since they both update the parameters. The `parse_params/2` should look familiar to you since, just as in the other LiveComponents, we instruct a form to parse the parameters and send an `:update` message to the LiveView if the parsing is successful.

Now, let's continue with the `PaginationForm`. Open up `lib/meow_web/forms/pagination_form.ex` and key in the following code:

```
defmodule MeowWeb.Forms.PaginationForm do
  import Ecto.Changeset

  @fields %{
    page: :integer,
    page_size: :integer,
    total_count: :integer
  }

  @default_values %{
    page: 1,
    page_size: 20,
    total_count: 0
  }

  def parse(params, values \\ @default_values) do
    {values, @fields}
    |> cast(params, Map.keys(@fields))
    |> validate_number(:page, greater_than: 0)
    |> validate_number(:page_size, greater_than: 0)
    |> validate_number(:total_count, greater_than_or_equal_to: 0)
    |> apply_action(:insert)
  end

  def default_values(overrides \\ %{}) do
    Map.merge(@default_values, overrides)
  end
end
```

As with the previous forms, we define a list of fields, their types, and their default values first. You can see that we show the first page with a page size of 20 by default. Both the page and page_size must be greater than 0 since we always want to show at least one page if any meerkat data exists. If no data exists, we don't show the page stepper and don't validate the page or page size. So, valid page and page size values should always be greater than 0. However, our total count can be 0 since we might not have any meerkat data

in our database or we set a filter which matches no entries. That's why we validate that the total count is always greater than or equal to 0.

Now that we have the `PaginationComponent` and the `PaginationForm` ready, let's make them usable by adding them to the `LiveView`.

Adding Pagination to the LiveView

Before we can use the `PaginationComponent`, we need to add it to our `HEEx` template. Please open up `lib/meow_web/live/meerkat_live.html.heex` and add the following code:

```
<div>
  <!-- Filter form stays here -->

  <!-- Table stays here -->

  <!-- Add this .live_component -->
  <.live_component
    module={MeowWeb.MeerkatLive.PaginationComponent}
    id="pagination"
    pagination={@pagination} />
</div>
```

This will render our `PaginationComponent` beneath the table. Now, we need to assign our default `@pagination` values. Open up the `LiveView` at `lib/meow_web/live/meerkat_live.ex` and update the code as follows:

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

  # Other aliases omitted

  # Add this alias
  alias MeowWeb.Forms.PaginationForm

  # mount/3, handle_params/3, and handle_info/2 omitted

  # Update the parse_params/2 function like this:
  defp parse_params(socket, params) do
    with {:ok, sorting_opts} <- SortingForm.parse(params),
         {:ok, filter_opts} <- FilterForm.parse(params),
         {:ok, pagination_opts} <- PaginationForm.parse(params) do
      socket
      |> assign_sorting(sorting_opts)
      |> assign_filter(filter_opts)
      |> assign_pagination(pagination_opts)
    else
      _error ->
        socket
        |> assign_sorting()
        |> assign_filter()
        |> assign_pagination()
    end
  end
end
```

```

    end
  end

  # assign_sorting/2 and assign_filter/2 omitted

  # Add this function:
  defp assign_pagination(socket, overrides \\ %{}) do
    assign(socket, :pagination, PaginationForm.default_values(overrides))
  end

  # Other functions omitted
end

```

Now, we parse any pagination-related parameters using the `PaginationForm` and assign the default pagination values if no parameters were given. However, we still need to hand over the pagination parameters to our Meerkats context when we fetch the meerkat data. Let's first update the code and discuss it afterward. Please update the LiveView as follows:

```

# Update assign_meerkats/1 like this:
defp assign_meerkats(socket) do
  params = merge_and_sanitize_params(socket)

  %{meerkats: meerkats, total_count: total_count} =
    Meerkats.list_meerkats_with_total_count(params)

  socket
  |> assign(:meerkats, meerkats)
  |> assign_total_count(total_count)
end

# Update merge_and_sanitize_params/2 like this:
defp merge_and_sanitize_params(socket, overrides \\ %{}) do
  %{sorting: sorting, filter: filter, pagination: pagination} = socket.assigns
  overrides = maybe_reset_pagination(overrides)

  %{}
  |> Map.merge(sorting)
  |> Map.merge(filter)
  |> Map.merge(pagination)
  |> Map.merge(overrides)
  |> Map.drop([:total_count])
  |> Enum.reject(fn {_key, value} -> is_nil(value) end)
  |> Map.new()
end

# Add this function:
defp assign_total_count(socket, total_count) do
  update(socket, :pagination, fn pagination ->
    %{
      pagination
      | total_count: total_count
    }
  end)
end)

```

```

end
# And this one:
defp maybe_reset_pagination(overrides) do
  if FilterForm.contains_filter_values?(overrides) do
    Map.put(overrides, :page, 1)
  else
    overrides
  end
end
end

```

As a last step, add the following to the FilterForm-module:

```

def contains_filter_values?(opts) do
  @fields
  |> Map.keys()
  |> Enum.any?(fn key -> Map.get(opts, key) end)
end
end

```

There are a few things to unpack here. Let's start with `assign_meerkats/1`. We updated this function to use the new `list_meerkats_with_total_count/1` function from our Meerkats context. This means that we receive the meerkat data as well as its total count when we fetch our data. The meerkat data still behaves the same as before, but now we have to handle the new `total_count` value somehow.

We need the `total_count` in our `@pagination` assign to calculate how many page buttons to show in the page stepper. Since we assigned the `@pagination` values before we call the `assign_meerkats/1` function, we can use Phoenix LiveView's `update/3` function to fetch the current `@pagination` value and update its `total_count`. That's exactly what `assign_total_count/2` does. It fetches the current `@pagination` assign and updates its `total_count` with the `total_count` received from the database.

The last change we need to make is to update the `merge_and_sanitize_params/2` function to include the new pagination parameters when we merge all parameters. So, we add the new pagination parameters with an extra `Map.merge/2` call.

Also, we don't want to show the `total_count` parameter in the URL, since it's purely an internal variable which we use for calculating the number of pages. Hence, we remove it from the mix of parameters using `Map.drop/2`.

When we set a new filter, we need to reset the pagination. Otherwise, we might see an empty table, although our filter returned some meerkats, all because we're still on, for example, the second page and our new filter returned only a single page of results. That's why we reset the pagination in the `maybe_reset_pagination/1` function. It asks the FilterForm whether our new parameters contain any filter parameters. If yes, it resets our pagination to the first page.

And that's it! Start the Phoenix server with `mix phx.server` and have a go at the new pagination! You should see something like this:

The screenshot shows a web browser address bar with the URL: `http://localhost:4000/?name=Sweet&page=2&page_size=10&sort_by=name&sort_dir=desc`. Below the address bar is a search form with two input fields: one for 'Id' (empty) and one for 'Name' (containing 'Sweet'). A blue 'SEARCH' button is to the right. Below the search form is a table with two columns: 'id' and 'name'. The table contains five rows of data:

| id | name ↓ |
|----|---------------|
| 41 | Sweet Buttons |
| 75 | Sweet Bubbles |
| 62 | Sweet Baloo |
| 2 | Sweet Baloo |
| 49 | Sweet Bacon |

Below the table are pagination controls: two buttons labeled '1' and '2', where '2' is highlighted in black. Below these is a dropdown menu showing '10' with a downward arrow.

Note how we store all sorting, filtering, and pagination parameters as URL parameters. The user can copy and paste this URL, share it, or bookmark it, and will always get the same view when they access the URL again in the future.

Try out stepping through the pages and changing the page size. Try combining the pagination with the sorting and filtering functionality. Does it work as expected? How would you improve the combination of these features? Where would you implement these changes?

Wrapping Up

We covered how to manage large amounts of data by chunking them into pages and how to allow the user to navigate through these pages using a pagination stepper. A pagination stepper helps users to understand on which page they can find what data, and it lets them share their current page with others.

The downside of a pagination stepper is that it causes the user to lose their focus on the data when they navigate to another page. This is acceptable in systems that are created for analyzing data, like scientific applications or back-office software, since they care more about offering a predictable behavior and allowing the user to share their findings than they care about engaging the user.

However, in systems where user engagement is a core business value, like social media platforms, it's unacceptable if users lose their focus on the content, since it makes them more likely to leave the page or close the application. That's why such systems tend to use a different method for navigating through the pages, which will be the topic of the next chapter: infinity scrolling.

Paginating the Table Using Infinity Scrolling

The last advanced table UI feature that we'll explore in this book is how to paginate our data using infinity scrolling. It allows the user to navigate through the pages of data without requiring any user input like the pagination stepper did. This way, we can load our data in chunks, thus keeping our website performant and responsive while not impairing its user friendliness.

Let's think about our page navigation solution from the previous chapter: the pagination stepper. It's a good solution if the user requires a deterministic behavior from the UI and wants to share their findings with others. They can step through the pages and can expect to see a different subset of the data with every click. They can also copy and paste the URL and share their current view with others.

However, every time the user navigates to another page, they lose focus on the currently displayed content. They need to find the button for the next page in the pagination stepper, click it, and move their focus back to the table where now everything has changed. They need to erase their mental model of the previous data and adjust to the new data. It's a hard transition between data subsets, which breaks the user focus and potentially their engagement with the site. Due to these issues with the pagination stepper, websites whose business model relies heavily on user engagement, like social media platforms, use infinity scrolling to paginate their content.

Infinity scrolling is simple to understand, and you've probably used it many times before. Whenever you scroll down a website and get close to the bottom of the content, the website loads and appends the next page of content outside your view and without your explicit request to do so. This way, you never run

out of content to view and all you have to do is to keep scrolling. It's a seamless experience and allows you to keep your focus on the content rather than having to request the next page of content manually.

Websites that use infinity scrolling rarely allow you to filter or sort the data. They rather rely on their own rules for sorting the data in order to choose the most suitable content for you. Therefore, in this chapter, we'll build a new table UI, one without sorting or filtering functionality, that teaches you how to implement infinity scrolling using Phoenix LiveView and a tiny bit of JavaScript. Let's get started!

Setting up the Context

Since our LiveView won't offer filtering or sorting of the data, it doesn't make sense to use the `list_meerkats_with_total_count/1` function that we used in the previous chapter. Rather, let's create a new function for fetching the next page of meerkat data for our new LiveView. Open up the Meerkats context in `lib/meow/meerkats.ex` and add the following two functions:

```
def meerkat_count(), do: Repo.aggregate(Meerkat, :count)

def list_meerkats_with_pagination(offset, limit) do
  from(m in Meerkat)
  |> limit(^limit)
  |> offset(^offset)
  |> Repo.all()
end
```

The first function, `meerkat_count/0`, simply returns the count of all meerkat entries in our database. We'll need this function in our new LiveView to determine when we have loaded all meerkat data and can stop fetching new data. However, you don't need this function if you think that you'll never run out of new content to show to the user. But more about this later.

The second function, `list_meerkats_with_pagination/2`, fetches a subset of meerkat data specified with an `offset` and a `limit`. If you think back to the previous chapter, you'll remember that we can specify how many entries the query should skip with the `offset` parameter and how many entries the query should return with the `limit` parameter. We'll use these parameters to load the next chunk of data whenever the user gets close to the end of the currently displayed content. Again, more about this later.

These two functions are all you need in order to implement infinity scrolling in your LiveView. If you believe that you will never run out of data to show, then you can even remove one of them! It's always astonishing to see how much you can do with a few lines of Elixir.

Now that we have these two functions in place, let's have a look at the new LiveView.

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.

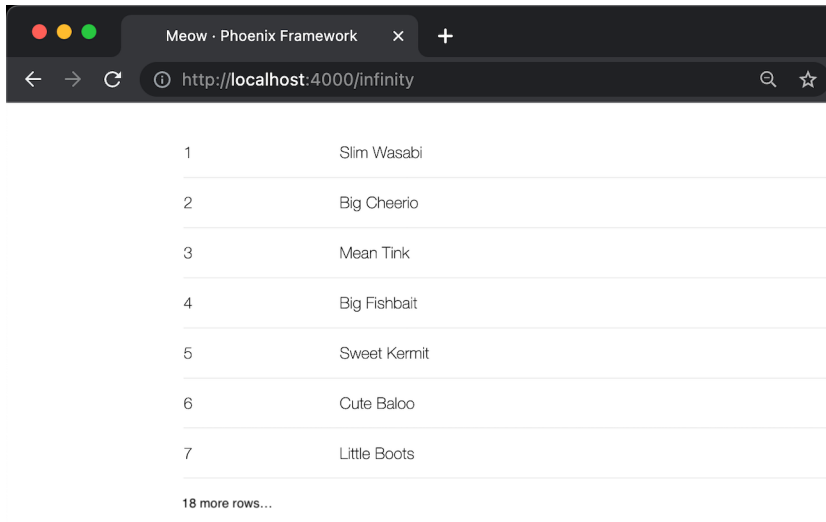
As you can see in the assign(offset: offset + limit) call, we increase our offset by our limit every time when we fetch new meerkat data. This means that at a certain point, our offset will be larger than the count of all meerkat data. When this happens, it would be pointless to keep on querying the 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:



| | |
|---------------------------------|--------------|
| 1 | Slim Wasabi |
| 2 | Big Cheerio |
| 3 | Mean Tink |
| 4 | Big Fishbait |
| 5 | Sweet Kermit |
| 6 | Cute Baloo |
| 7 | Little Boots |
| 18 more rows... | |

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.

Adding the LiveView Client Hook

If you've worked with Phoenix LiveView a bit, you know how simple it is to send updates from the server to the user's browser and let it display the changes. You simply assign the new data to your socket. LiveView then sends the updates through the websocket connection to the browser. Once the browser receives the updates, the JavaScript part of LiveView re-renders the relevant parts of the UI. So, server-to-client communication in Phoenix Live-

View is easy because the framework does all the heavily lifting for us. All it takes is a single function call and LiveView does the rest.

But this only covers the server-to-client communication. So, how does LiveView allow us to communicate in the other direction, from the client to the server? The answer is LiveView Client Hooks. We'll dive into them next.

A Brief Excursion into LiveView Client Hooks

The first versions of Phoenix LiveView only supported server-to-client communication. It was easy to update the UI whenever something on the back end changed. However, it was not trivial to customize the logic that handled the updates on the front end.

For example, what if you wanted to show an alert whenever an update comes in? You could build a solution using advanced CSS selectors. You could, for instance, let a new row flash in noticeable colors when you add it to the table. So, there were some work-arounds for handling updates in the UI, but some things remained almost impossible to implement. For example, you would struggle to add an event listener to the users' browser and send a custom event to the LiveView whenever the listener receives an event.

Think about how you would implement the popular game Snake using LiveView. It would be easy to render the current location and shape of the snake in the browser, but how would you handle the user input for moving the snake up, down, left, or right?

Before LiveView introduced key event handlers that made handling user input to the keyboard trivial, you had to implement your own event listener and call an API endpoint on your server to update the location of the snake. If you were very motivated, you could implement your own websocket connection with the server using Phoenix Channels, which reduced the overhead of making a separate HTTP request for every key event. As you can see, the client-to-server communication was complex and made LiveView unusable for certain use-cases.

The smart people behind the Phoenix LiveView framework quickly recognized the need to improve the interoperability of LiveView and client-side JavaScript. One of the biggest steps toward this goal is a feature called *client hooks*.

Client hooks, or in short just hooks, let you *hook into* the life cycle of any UI element. That means that with just a few lines of JavaScript, you can execute custom code when, for example, a new row is added to the table. They also

allow you to send custom events back to your LiveView with a single function call.

To use a hook, you need to do three things: define your hook, add it to your LiveSocket, and set a `phx-hook` tag on your UI element. Let's create a simple hook to see how it works.

Heads up: there's no need for you to add the following code to your Meow application, but if you want to, you can of course. However, it's enough to read and understand the following code. We'll implement an actual client hook later on.

First, we need to define our hook. Let's write an example hook that logs a message to the browser's console and sends an event back to our LiveView. The code could look as follows:

```
const PingPongHook = {
  addPongListener() {
    window.addEventListener("phx:pong", (event) => {
      console.log(event.type);
      console.log(event.detail.message)
    })
  },
  sendPing() {
    this.pushEvent("ping", { myVar: 1 });
  },
  mounted() {
    console.log("I'm alive!");
    this.addPongListener();
    this.sendPing();
  },
};
```

As you can see, our hook `PingPongHook` defines custom code that LiveView should execute when the UI element is mounted to the browser. In this case, we first log the message *"I'm alive"* to the browser's console. Then, we call a custom function that we defined on the hook called `sendPing()`. It sends a custom *"ping"* event back to the LiveView. We can even add a payload to the event with the second parameter of the `pushEvent` function.

We can extend the hook to run custom code at many different life-cycle steps. Have a look at the official documentation to see which steps are available by the time you're reading this book.

Now that we've defined our hook, we have to add it to the LiveSocket so that it becomes available in the browser. We can do that by adding the following code to our `app.js`:

```

let Hooks = {};
Hooks.PingPongHook = PingPongHook;

let liveSocket = new LiveSocket("/live", Socket, {
  hooks: Hooks,
  // params and other options
});

```

Now, we can add the hook to any HTML element. For example, we could create a simple *div* and add the hook like this:

```
<div id="myDiv" phx-hook="PingPongHook" />
```

Note that you have to assign an id to the HTML element to which you want to add the hook. LiveView needs this to uniquely identify the element of the hook and to call the correct hook when the element changes.

Let's create a simple event handler in our LiveView for the "ping" event. It could look something like this:

```

def handle_event("ping", params, socket) do
  IO.inspect("ping", label: "Event")
  IO.inspect(params, label: "Params")
  {:noreply, push_event(socket, "pong", %{message: "Hello there!"})
end

```

Now, if we would run our server with `mix phx.server` and open the browser's console, we would see the following: first, our client hook logs "I'm alive" to the browser's console. Then the hook sends the "ping" event to the LiveView, where we log the following output:

```

Event: "ping"
Params: %{"myVar" => 1}

```

As you can see, our LiveView received the "ping" event and our payload from our client hook through the websocket connection. So far so good!

Now, as a response, the LiveView sends a "pong" event back together with the event payload: `{:message: "Hello there!"}`. The "pong" event then triggers our event listener for the "phx:pong" event in our browser. If we peek into the browser's console once more, we see the following output:

```

phx:pong
Hello there!

```

It seems like our event listener successfully accessed the payload of the "pong" event and logged its name and payload to the console. Our ping-pong conversation between client and server worked!

And that's it. With just a few lines of Elixir and JavaScript, we added custom client-to-server and server-to-client communication to our website. Amazing!

Now, let's use what you've learned about client hooks to send a *"load-more"* event to your LiveView whenever the user scrolls and gets close to the bottom of our table.

First, we need to define the hook. It should check how much hidden content is left when the user scrolls and send a *"load-more"* event to the LiveView if the user is at more than 90% of the current content. So, whenever the user has less than 10% of the currently rendered content to scroll through, we want to load more content so that they don't hit the bottom of the page. The 90% is just an example here. You can set it to whatever threshold suits your application best.

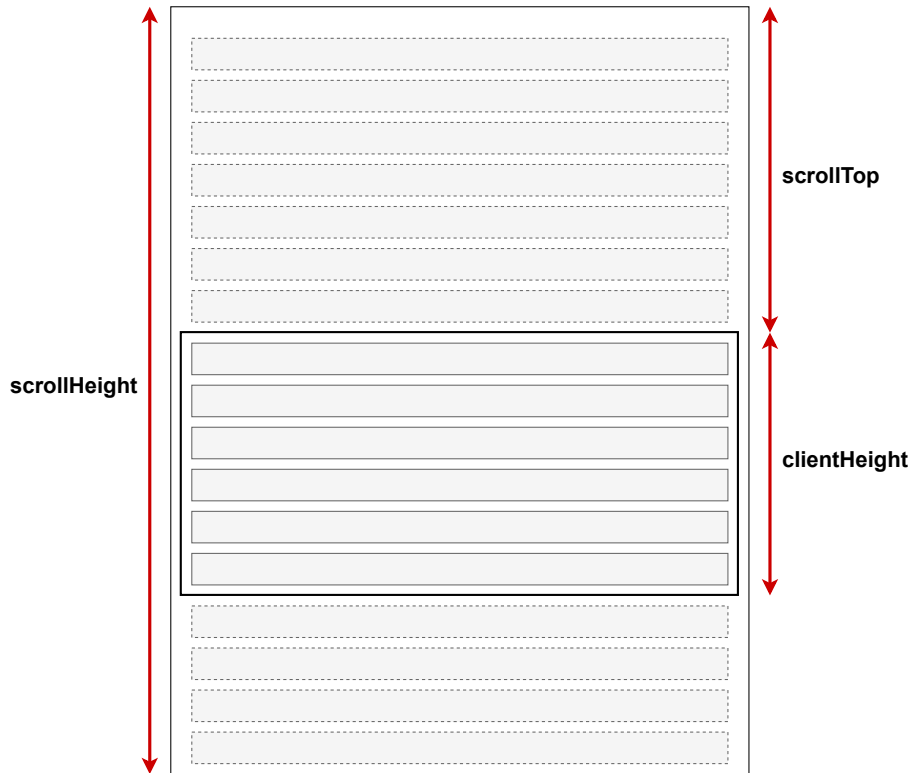
Let's write some code. Create a new file at `assets/js/infinity-scroll.js` and add the following:

```
export default {
  rootElement() {
    return (
      document.documentElement || document.body.parentNode || document.body
    );
  },
  scrollTop() {
    const { scrollTop, clientHeight, scrollHeight } = this.rootElement();
    return ((scrollTop + clientHeight) / scrollHeight) * 100;
  }
}
```

Let's have a look at the hook so far. We first added two helper functions: `rootElement()` and `scrollTop()`.

The `rootElement()` function tries to return the `html` element of your website. However, for browser compatibility, it sometimes falls back to the `body` element in case it doesn't have access to the `html` element. We won't dive into why this is necessary since it would be a whole other book about web standards and how all browsers implement them differently. For now, let's just assume that `rootElement()` returns the outermost element of our *HTML* template, which is usually the `html` element.

Now, let's have a look at the `scrollTop()` function. It first fetches the `scrollTop`, `clientHeight`, and `scrollHeight` parameters from our root element and then calculates how much content the user can still scroll through. To understand the calculation, we have to understand the three parameters first. Have a look at the following diagram:



The diagram shows an example table and the client's browser window that shows only a part of the entire table. The height of the browser window is captured by the `clientHeight` variable. It's important to take the browser height into our calculation since some users have very tall screens and they only have to scroll a little bit to reach the end of the table, whereas other users might have a small screen, like a laptop screen, and have to scroll much further to reach the end of the table. For the first group of users, we need to preload much more content so that they never run out of content to scroll through. For users with smaller screens, we don't need to preload as much.

When the user starts scrolling, the browser window moves down the table. The `scrollTop` variable captures how far the user has scrolled. It's the distance from the top of the table to the topmost visible content of the table. So, when we add up the `scrollTop` and `clientHeight` variables, we get a good measure for how far the user has scrolled down the table.

The next step is to calculate how much content is left for the user to scroll through. The `scrollHeight` variable captures the total height of the table. This height increases whenever we add content to the end of the table. Hence, we

can use this variable to calculate the percentage of content that is left for the user to scroll through by simply dividing the current scroll position by the total scroll height.

This is exactly what our `scrollPosition()` function does. It adds the `scrollTop` and `clientHeight` variables and divides the sum by the `scrollHeight`. This returns a value between 0 and 1. To get a better understandable percentage between 0 and 100, we multiply the result by 100. Now, we have the percentage of content that the user has seen.

We can use this percentage to load more content when the user is close to the end of the table. Let's add this functionality to the `mounted()` life-cycle step. Open up `assets/js/infinity-scroll.js` again and add the following function beneath `scrollPosition()`:

```
mounted() {
  this.threshold = 90;
  this.lastScrollPosition = 0;

  window.addEventListener("scroll", () => {
    const currentScrollPosition = this.scrollPosition();

    const isCloseToBottom =
      currentScrollPosition > this.threshold &&
      this.lastScrollPosition <= this.threshold;

    if (isCloseToBottom) this.pushEvent("load-more", {});

    this.lastScrollPosition = currentScrollPosition;
  });
}
```

There's a lot going on in this function, so let's dissect it. First, we define the threshold percentage at which we want to load more content. We set it to 90%, so whenever the user has seen at least 90% of the current content, we load more.

You can adjust this threshold to suit your needs, of course. If your content takes longer to load, consider decreasing this threshold to give your website more time to load the new content. If your content loads fast, you can increase this threshold and reduce the total amount of network requests that your server will receive. Feel free to play around with it until you hit the sweet spot.

Next, we define the `lastScrollPosition` variable. We'll use this variable to ensure that we request data only when the user passes the threshold from below 90% to 90% or above. If we don't add this variable, we would request data also when the user scrolls from 90% to 91%, from 91% to 92%, and so on,

which we don't want. We only want to load data once they pass the 90% threshold and then wait for the content to appear.

One edge-case that this check doesn't cover is when a user scrolls from 89% to 90%, back to 89%, and then again to 90% all before we add more content. In that case, we would request new data more than once. You could cover this edge-case by keeping track of the page or offset that you requested. So, when you load all the data up to and including page 10 and you just request page 11, you wouldn't request page 11 again.

However, this requires you to synchronize which pages you requested already, both in the LiveView and in the client hook, which is more complex than simply tracking the current scroll position. But if you're interested in a solution to this, have a look at the Phoenix LiveView documentation about JavaScript interoperability. The smart folks from the LiveView project present a page-based solution, which solves this edge-case but is harder to understand than the scroll-position approach.

Let's come back to the second part of the mounted() function, the event listener, part of our hook:

```
window.addEventListener("scroll", () => {
  const currentScrollPosition = this.scrollPosition();

  const isCloseToBottom =
    currentScrollPosition > this.threshold &&
    this.lastScrollPosition <= this.threshold;

  if (isCloseToBottom) this.pushEvent("load-more", {});

  this.lastScrollPosition = currentScrollPosition;
});
```

When LiveView mounts our table UI element, we add an event listener to its root that listens for "scroll" events. So, whenever the user scrolls, the browser calls the callback that we define here. Let's see what happens next:

First, we get the current scroll position, which we calculate with our scrollPosition() function. Then we check whether the user has crossed the 90% threshold with `currentScrollPosition > this.threshold`. We also check that the user hasn't crossed the threshold before with `this.lastScrollPosition <= this.threshold`.

If both of these checks are successful, we push a "load-more" event to the LiveView, which in turn loads more data and adds it to the table. When this happens, the current scroll position drops below 90% again since we increase the scrollHeight by adding more content. Now, the user is below the threshold

again and we're ready to trigger the *"load-more"* event again when they keep on scrolling.

And that's it. With only 30 lines of JavaScript and 50 lines of Elixir, we brought our data to the users' fingertips and even added the advanced feature of infinity scrolling to it!

This concludes our implementation of the advanced infinity scrolling feature. Before you leave, maybe play around with the new feature. Does your content always load in time? If not, adjust the threshold a bit. Also test this feature by simulating a slow internet connection. Most browsers can simulate this for you. Is the table still responsive and does it load the content in time? If not, consider lowering the threshold even more and reducing the limit variable in the LiveView.

Infinity scrolling can improve the UX of your website a lot, but it can also become a disadvantage if a slow internet connection causes it to become sluggish. So, test this feature heavily on different and sometimes choppy internet connections. If you put in the extra effort, you're more likely to see your user engagement soar, which is the whole point of infinity scrolling.

Wrapping Up

You've reached the end of this book. Well done! Hopefully, you feel comfortable with implementing these advanced table UI features in your next project.

In this book, you learned how to bring your data to your users' fingertips with an advanced table UI. We touched on many different topics such as query composition, memory optimization, JavaScript interoperability, live components, and more. You learned about all the features commonly used in table UIs, such as pagination, sorting, and filtering. You even learned how to optimize the table UI for user engagement with infinity scrolling. You learned a lot! Give yourself a pat on your shoulder, you deserve it.

Thank you!

We hope you enjoyed this book and that you're already thinking about what you want to learn next. To help make that decision easier, we're offering you this gift.

Head on over to <https://pragprog.com> right now, and use the coupon code BUYANOTHER2023 to save 30% on your next ebook. Offer is void where prohibited or restricted. This offer does not apply to any edition of the *The Pragmatic Programmer* ebook.

And if you'd like to share your own expertise with the world, why not propose a writing idea to us? After all, many of our best authors started off as our readers, just like you. With up to a 50% royalty, world-class editorial services, and a name you trust, there's nothing to lose. Visit <https://pragprog.com/become-an-author/> today to learn more and to get started.

We thank you for your continued support, and we hope to hear from you again soon!

The Pragmatic Bookshelf



The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by professional developers for professional developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

This Book's Home Page

<https://pragprog.com/book/puphoe>

Source code from this book, errata, and other resources. Come give us feedback, too!

Keep Up-to-Date

<https://pragprog.com>

Join our announcement mailing list (low volume) or follow us on Twitter @pragprog for new titles, sales, coupons, hot tips, and more.

New and Noteworthy

<https://pragprog.com/news>

Check out the latest Pragmatic developments, new titles, and other offerings.

Buy the Book

If you liked this ebook, perhaps you'd like to have a paper copy of the book. Paperbacks are available from your local independent bookstore and wherever fine books are sold.

Contact Us

Online Orders: <https://pragprog.com/catalog>

Customer Service: support@pragprog.com

International Rights: translations@pragprog.com

Academic Use: academic@pragprog.com

Write for Us: <http://write-for-us.pragprog.com>

Or Call: +1 800-699-7764