CRUD

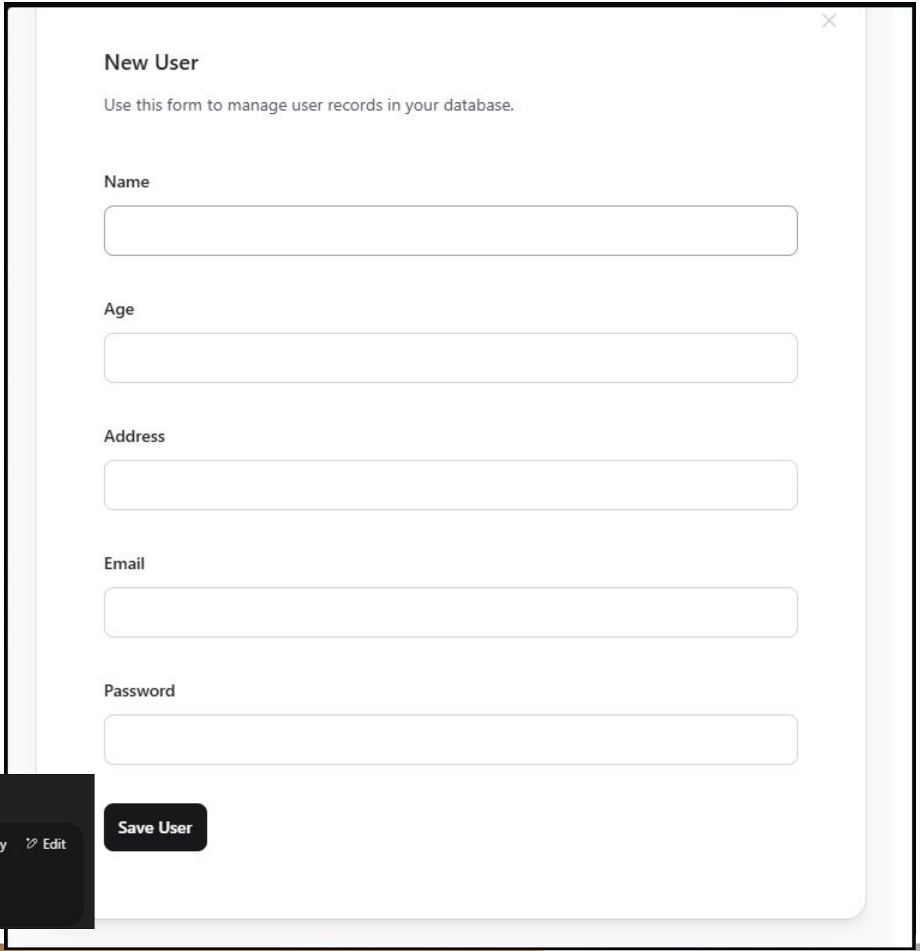
CREATE.READ.UPDATE.DELETE

Create

Frontend is what the user see-

You have a LiveView page with a **form that includes**:

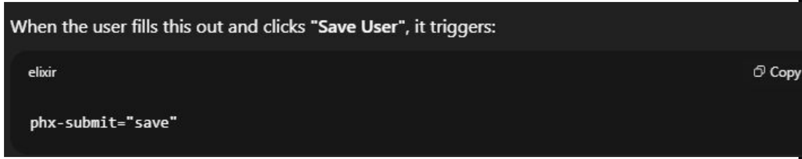
- Name
- Age
- Address
- Email
- Password



New User

Use this form to manage user records in your database.

Name
Age
Address
Email
Password

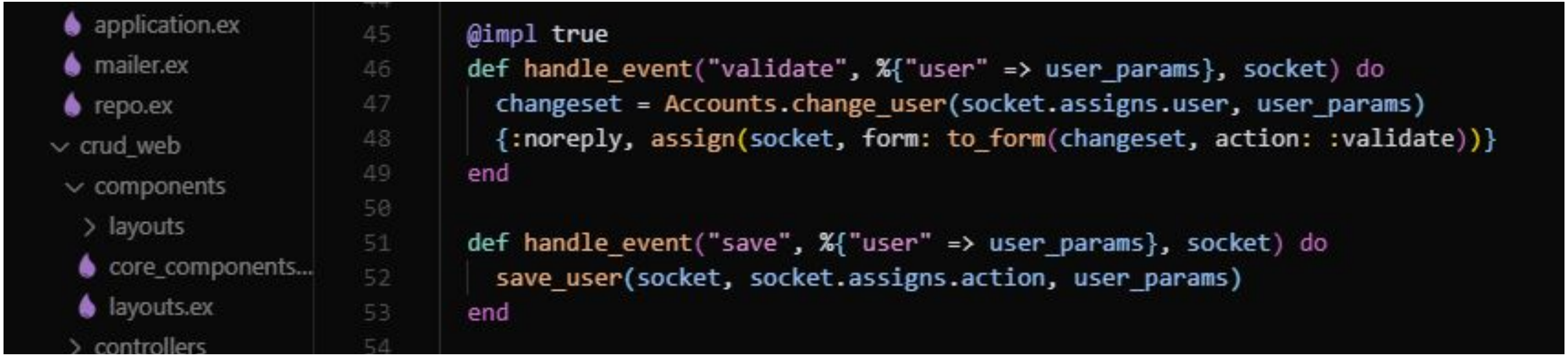


When the user fills this out and clicks **"Save User"**, it triggers:

```elixir
phx-submit="save"
```

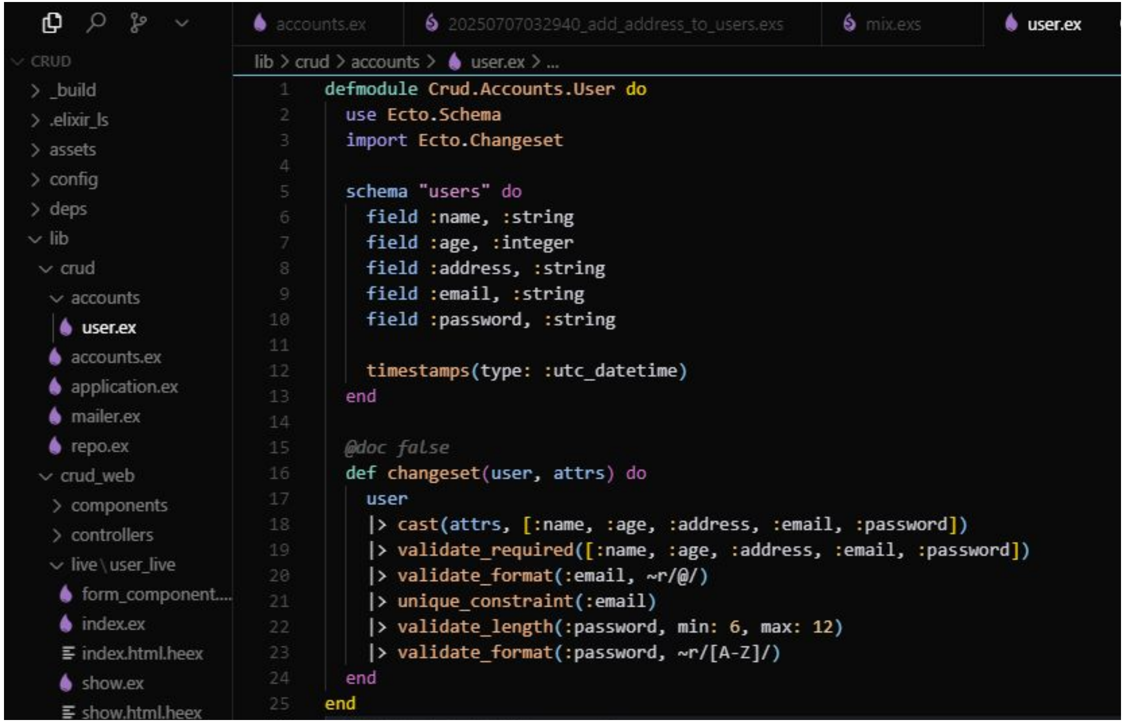
Handle_event (“validate”)- as user types, it use the changeset for live validation feedback

Handle_event (“save”)- After the user save, it create or update the user information into a Database



```elixir
@impl true
def handle_event("validate", %{"user" => user_params}, socket) do
  changeset = Accounts.change_user(socket.assigns.user, user_params)
  {:noreply, assign(socket, form: to_form(changeset, action: :validate))}
end

def handle_event("save", %{"user" => user_params}, socket) do
  save_user(socket, socket.assigns.action, user_params)
end
```

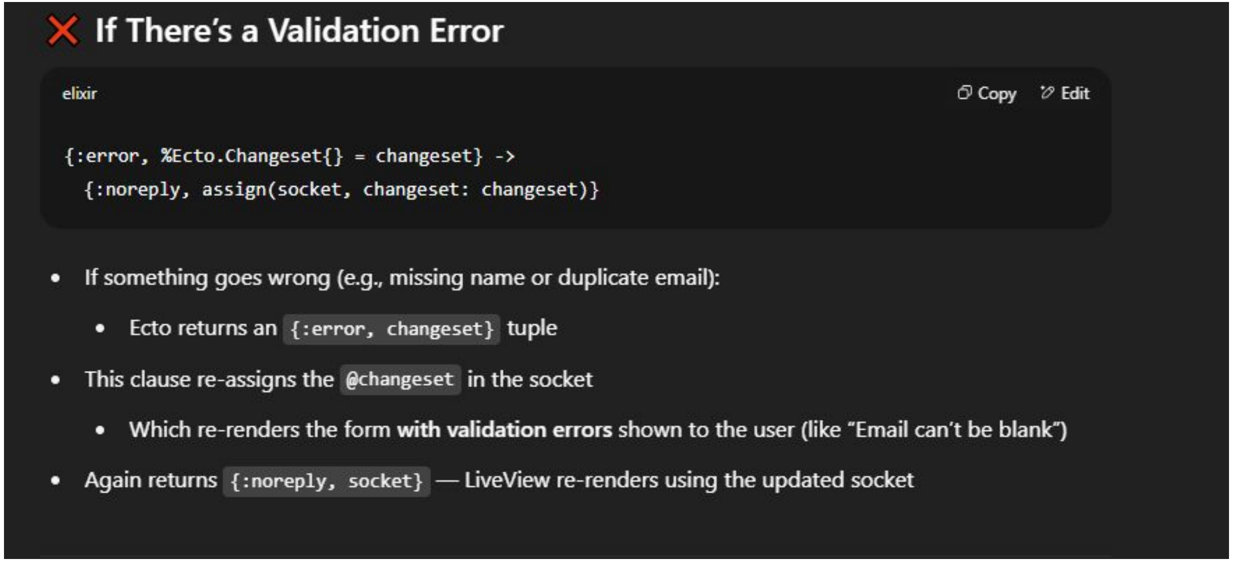
This is your changeset



```elixir
defmodule Crud.Accounts.User do
  use Ecto.Schema
  import Ecto.Changeset

  schema "users" do
    field :name, :string
    field :age, :integer
    field :address, :string
    field :email, :string
    field :password, :string

    timestamps(type: :utc_datetime)
  end

  @doc false
  def changeset(user, attrs) do
    user
    |> cast(attrs, [:name, :age, :address, :email, :password])
    |> validate_required([:name, :age, :address, :email, :password])
    |> validate_format(:email, ~r/@/)
    |> unique_constraint(:email)
    |> validate_length(:password, min: 6, max: 12)
    |> validate_format(:password, ~r/[A-Z]/)
  end
end
```

If there’s a validation error.

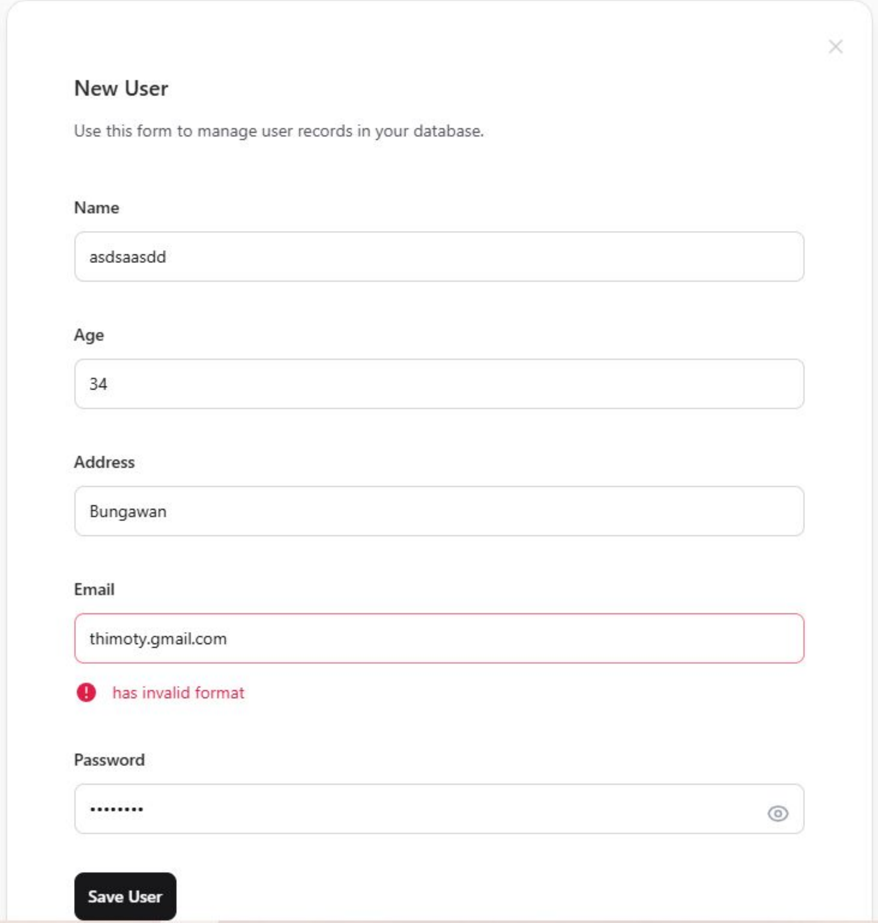
Ecto will returns a error like Email can’t be blank,ect..



❌ If There’s a Validation Error

```elixir
{:error, %Ecto.Changeset{} = changeset} ->
  {:noreply, assign(socket, changeset: changeset)}
```

- If something goes wrong (e.g., missing name or duplicate email):
  - Ecto returns an `{:error, changeset}` tuple
- This clause re-assigns the `@changeset` in the socket
  - Which re-renders the form **with validation errors** shown to the user (like “Email can’t be blank”)
- Again returns `{:noreply, socket}` — LiveView re-renders using the updated socket



New User

Use this form to manage user records in your database.

Name

asdsaasdd

Age

34

Address

Bungawan

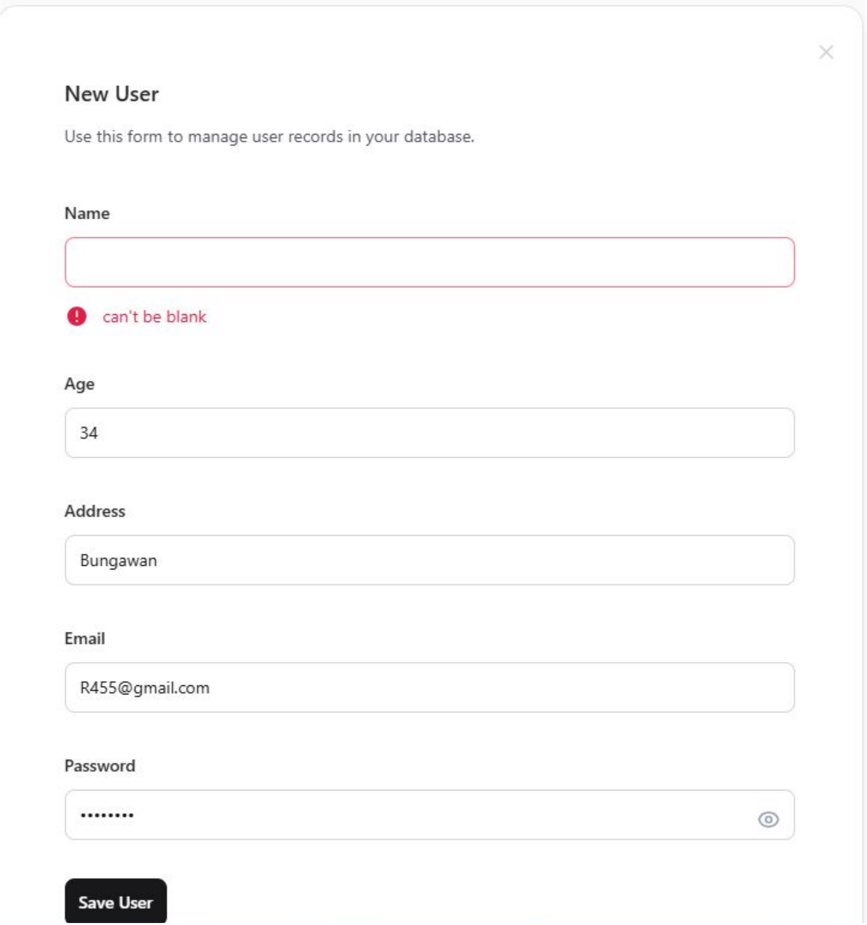
Email

thimoty.gmail.com

has invalid format

Password

••••••••



New User

Use this form to manage user records in your database.

Name

can't be blank

Age

34

Address

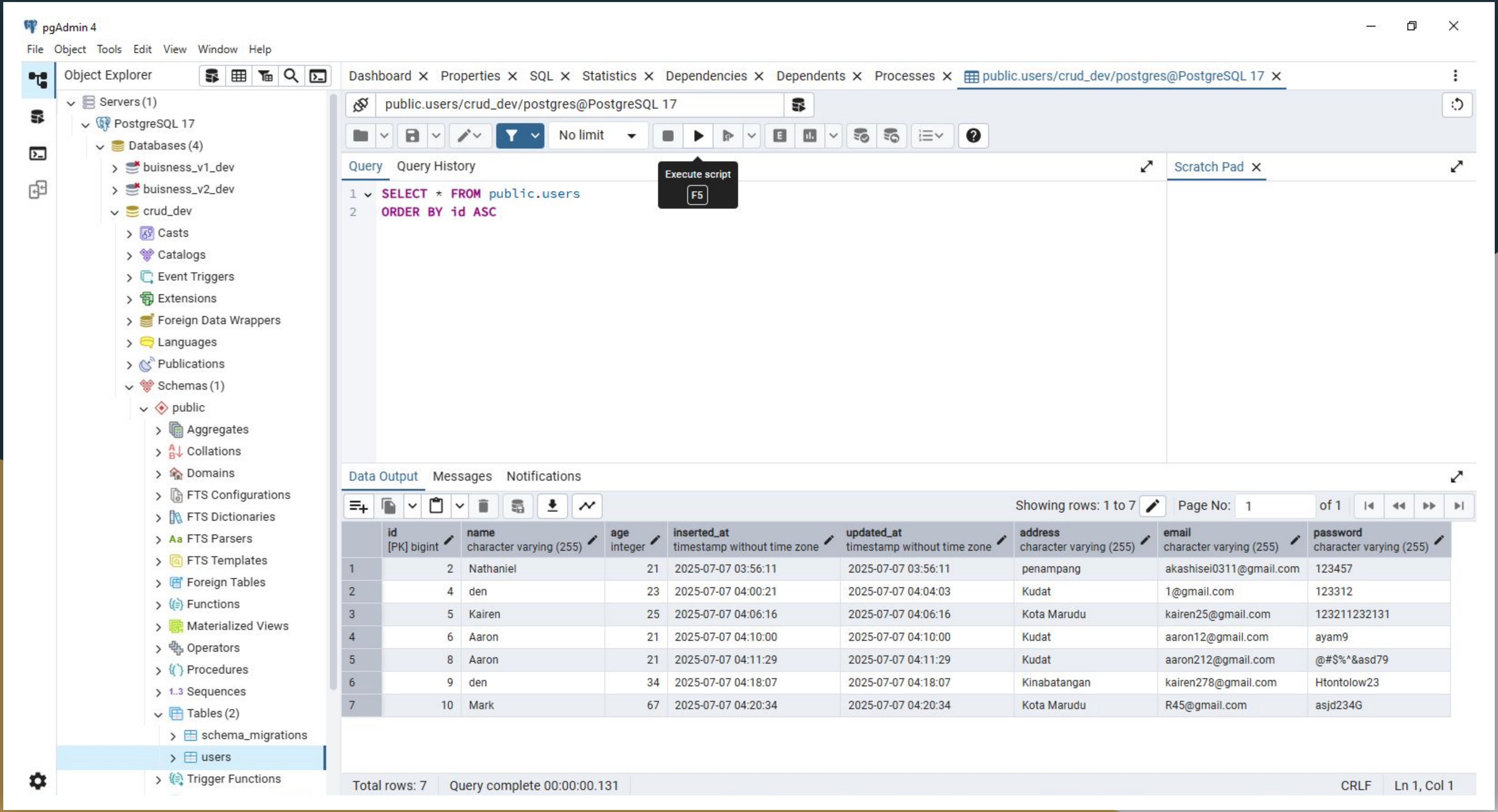
Bungawan

Email

R455@gmail.com

Password

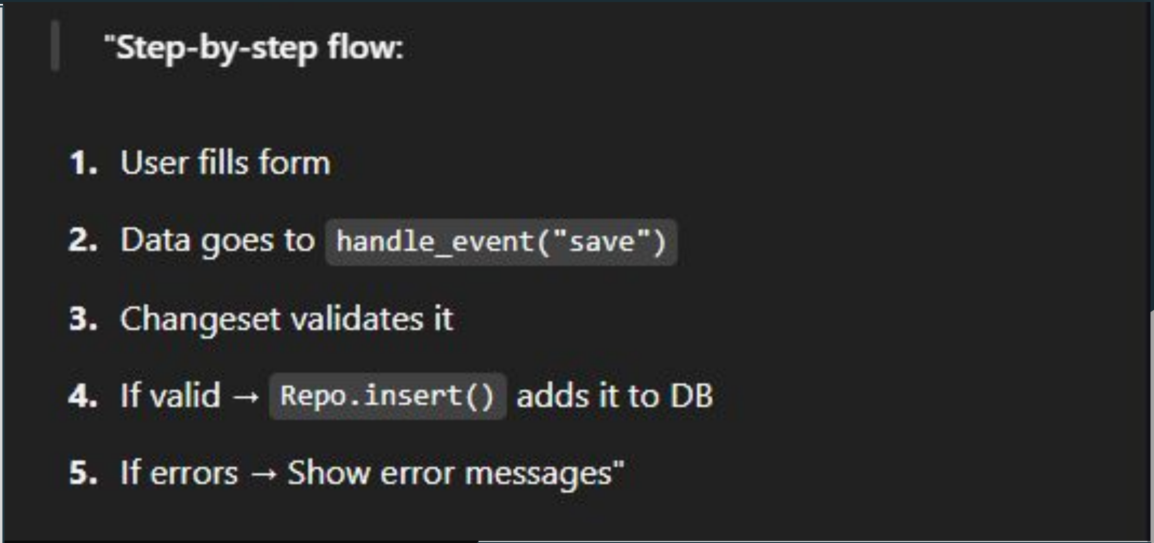
••••••••



```
SELECT * FROM public.users
ORDER BY id ASC
```

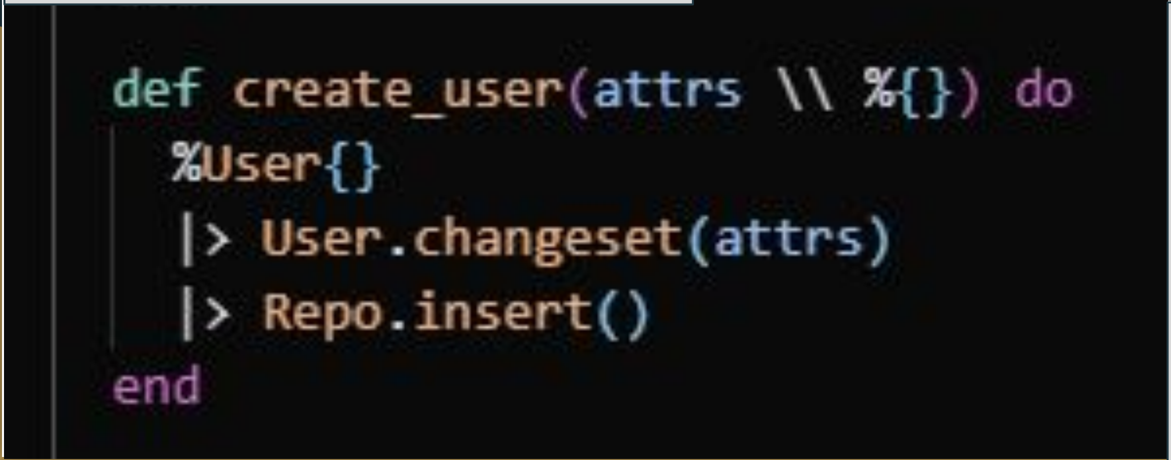
	id [PK] bigint	name character varying (255)	age integer	inserted_at timestamp without time zone	updated_at timestamp without time zone	address character varying (255)	email character varying (255)	password character varying (255)
1	2	Nathaniel	21	2025-07-07 03:56:11	2025-07-07 03:56:11	penampang	akashisei0311@gmail.com	123457
2	4	den	23	2025-07-07 04:00:21	2025-07-07 04:04:03	Kudat	1@gmail.com	123312
3	5	Kairen	25	2025-07-07 04:06:16	2025-07-07 04:06:16	Kota Marudu	kairen25@gmail.com	123211232131
4	6	Aaron	21	2025-07-07 04:10:00	2025-07-07 04:10:00	Kudat	aaron12@gmail.com	ayam9
5	8	Aaron	21	2025-07-07 04:11:29	2025-07-07 04:11:29	Kudat	aaron212@gmail.com	@#$%^&asd79
6	9	den	34	2025-07-07 04:18:07	2025-07-07 04:18:07	Kinabatangan	kairen278@gmail.com	Htontolow23
7	10	Mark	67	2025-07-07 04:20:34	2025-07-07 04:20:34	Kota Marudu	R45@gmail.com	asjd234G

Total rows: 7   Query complete 00:00:00.131   CRLF   Ln 1, Col 1



> "Step-by-step flow:

1. User fills form
2. Data goes to `handle_event("save")`
3. Changeset validates it
4. If valid → `Repo.insert()` adds it to DB
5. If errors → Show error messages"



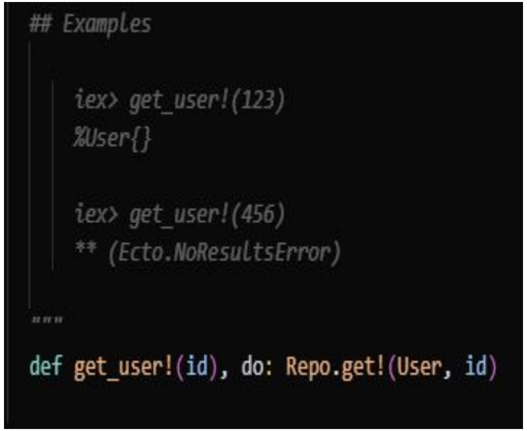
```
def create_user(attrs \\ %{}) do
  %User{}
  |> User.changeset(attrs)
  |> Repo.insert()
end
```

Read (get user)

➢ **Apa itu "Get User"?**

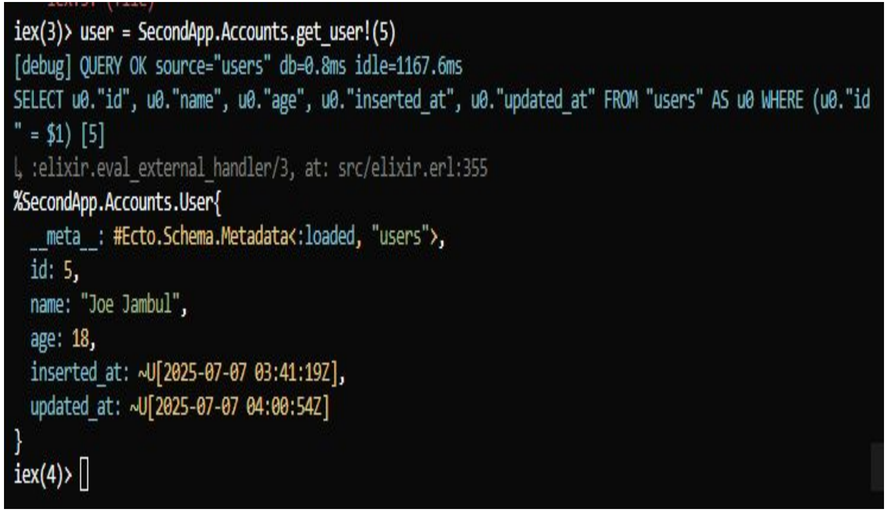
"Get user" bermaksud **mengambil data seorang pengguna** daripada pangkalan data berdasarkan **ID**. Ia merupakan sebahagian daripada operasi **Read** dalam CRUD.

➢ Contoh penggunaan dalam accounts.ex

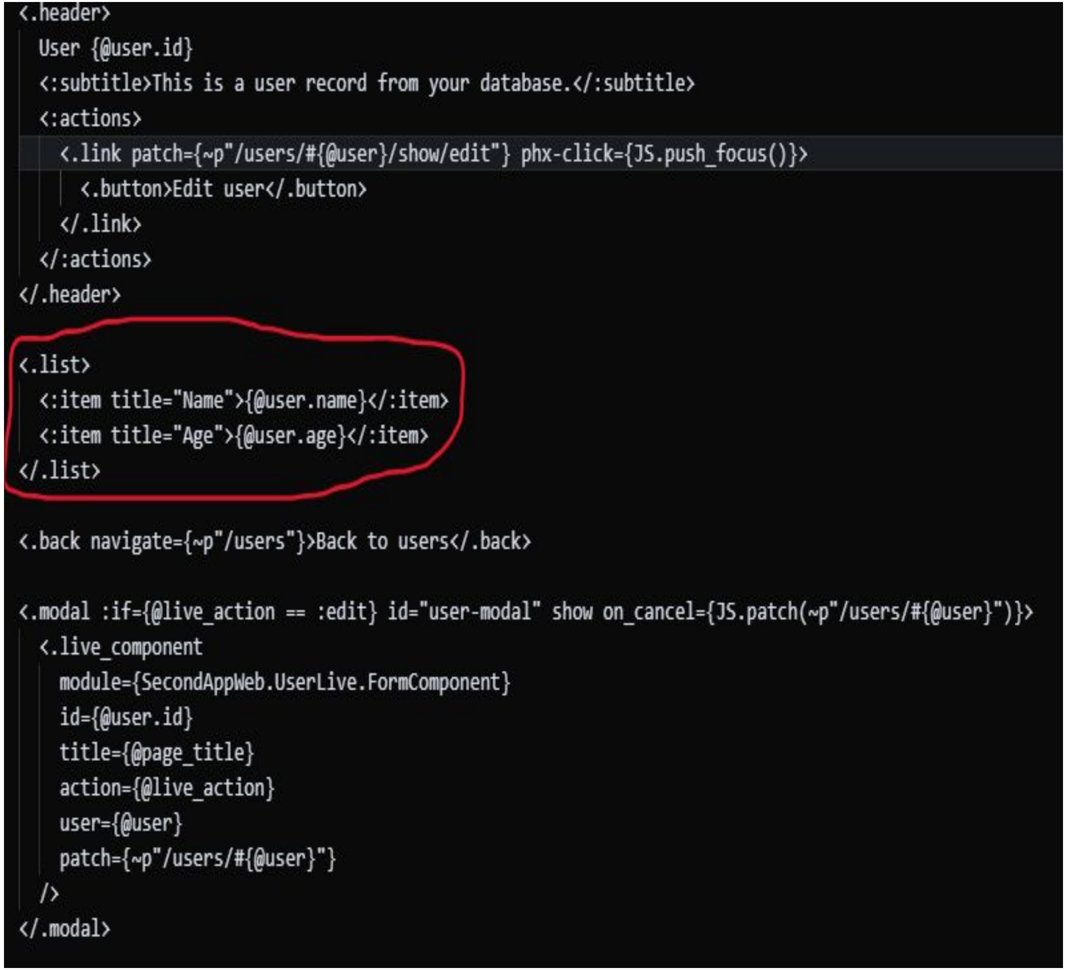


```
## Examples

    iex> get_user!(123)
    %User{}

    iex> get_user!(456)
    ** (Ecto.NoResultsError)

"""
def get_user!(id), do: Repo.get!(User, id)
```

➢ Contoh guna dalam IEX

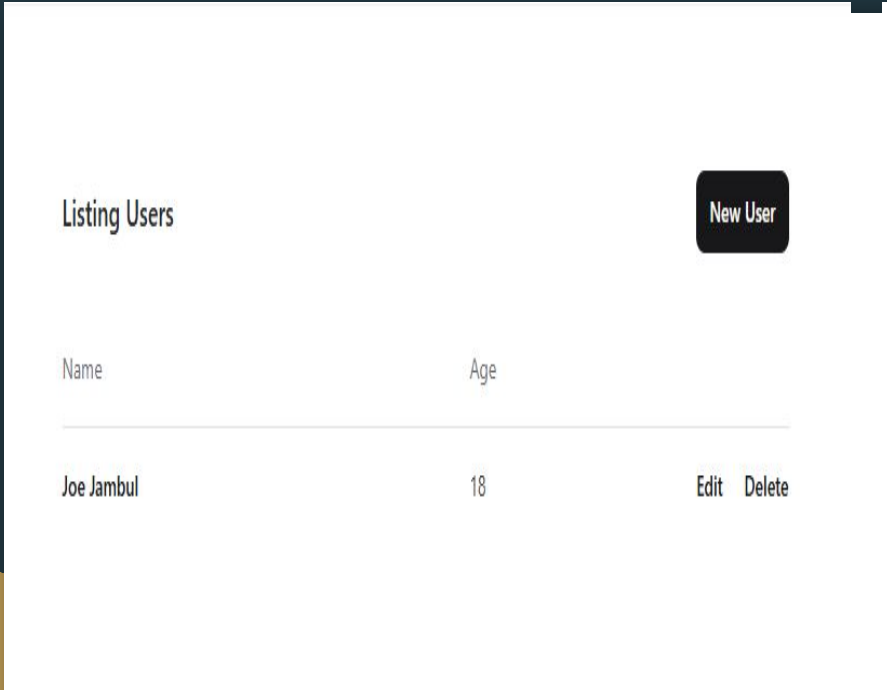


```
iex(3)> user = SecondApp.Accounts.get_user!(5)
[debug] QUERY OK source="users" db=0.8ms idle=1167.6ms
SELECT u0."id", u0."name", u0."age", u0."inserted_at", u0."updated_at" FROM "users" AS u0 WHERE (u0."id
" = $1) [5]
↳ :elixir.eval_external_handler/3, at: src/elixir.erl:355
%SecondApp.Accounts.User{
  __meta__: #Ecto.Schema.Metadata<:loaded, "users">,
  id: 5,
  name: "Joe Jambul",
  age: 18,
  inserted_at: ~U[2025-07-07 03:41:19Z],
  updated_at: ~U[2025-07-07 04:00:54Z]
}
iex(4)> 
```



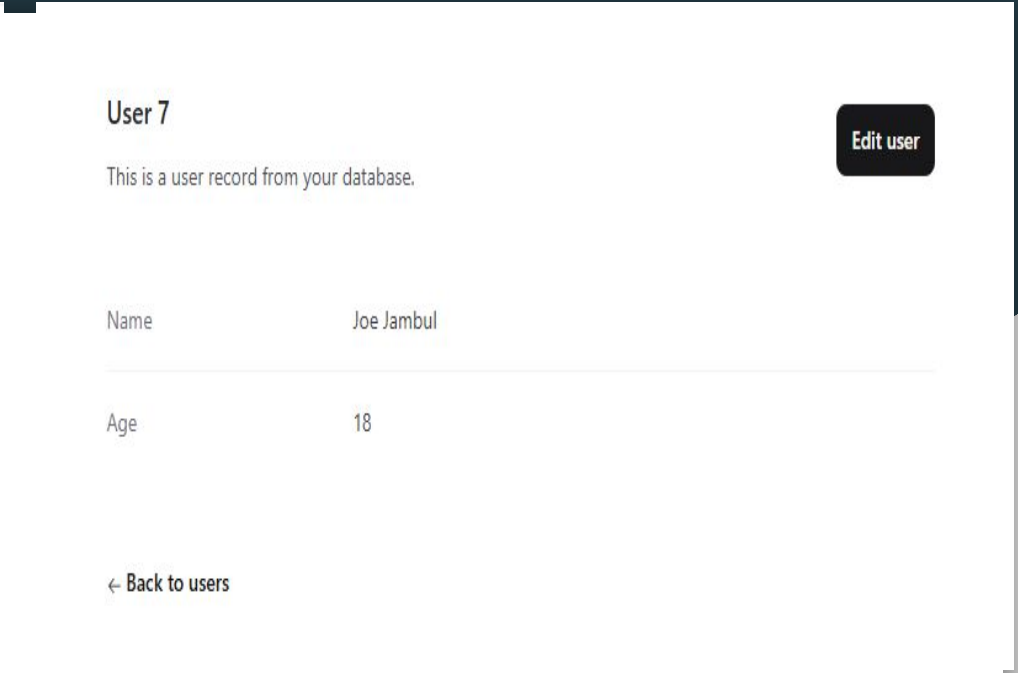
```
<.header>
  User {@user.id}
  <:subtitle>This is a user record from your database.</:subtitle>
  <:actions>
    <.link patch={~p"/users/#{@user}/show/edit"} phx-click={JS.push_focus()}>
      <.button>Edit user</.button>
    </.link>
  </:actions>
</.header>

<.list>
  <:item title="Name">{@user.name}</:item>
  <:item title="Age">{@user.age}</:item>
</.list>

<.back navigate={~p"/users"}>Back to users</.back>

<.modal :if={@live_action == :edit} id="user-modal" show on_cancel={JS.patch(~p"/users/#{@user}")}>
  <.live_component
    module={SecondAppWeb.UserLive.FormComponent}
    id={@user.id}
    title={@page_title}
    action={@live_action}
    user={@user}
    patch={~p"/users/#{@user}"}
  />
</.modal>
```

- ➢ Templat **HEEx** dalam projek **Phoenix LiveView**, digunakan untuk memaparkan maklumat seorang pengguna



Listing Users

Name	Age	
Joe Jambul	18	Edit Delete



User 7

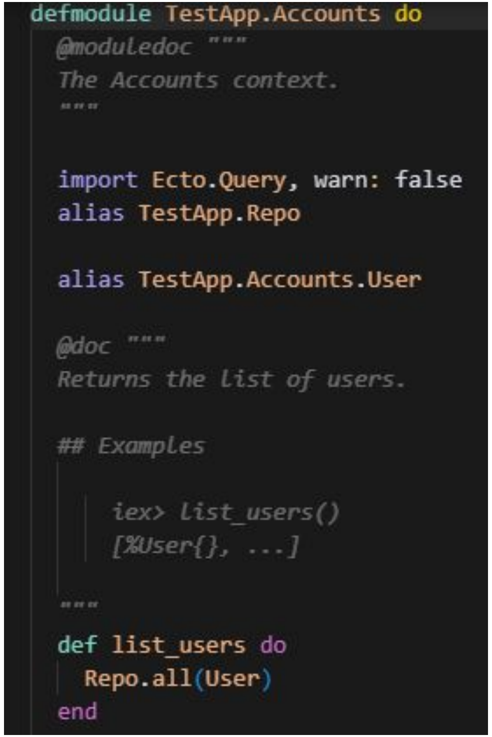
This is a user record from your database.

Name	Joe Jambul
Age	18

- **UI** yang dibina menggunakan **Phoenix LiveView**, khususnya halaman **paparan maklumat pengguna individu**.

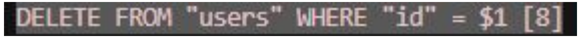
Read (List User)

Accounts.ex



```elixir
defmodule TestApp.Accounts do
  @moduledoc """
  The Accounts context.
  """

  import Ecto.Query, warn: false
  alias TestApp.Repo

  alias TestApp.Accounts.User

  @doc """
  Returns the list of users.

  ## Examples

      iex> list_users()
      [%User{}, ...]

  """
  def list_users do
    Repo.all(User)
  end
```

- Ecto.Query act as a bridge for repo to access the database.
- After access, it return to Elixir and structs (`%User{}`)



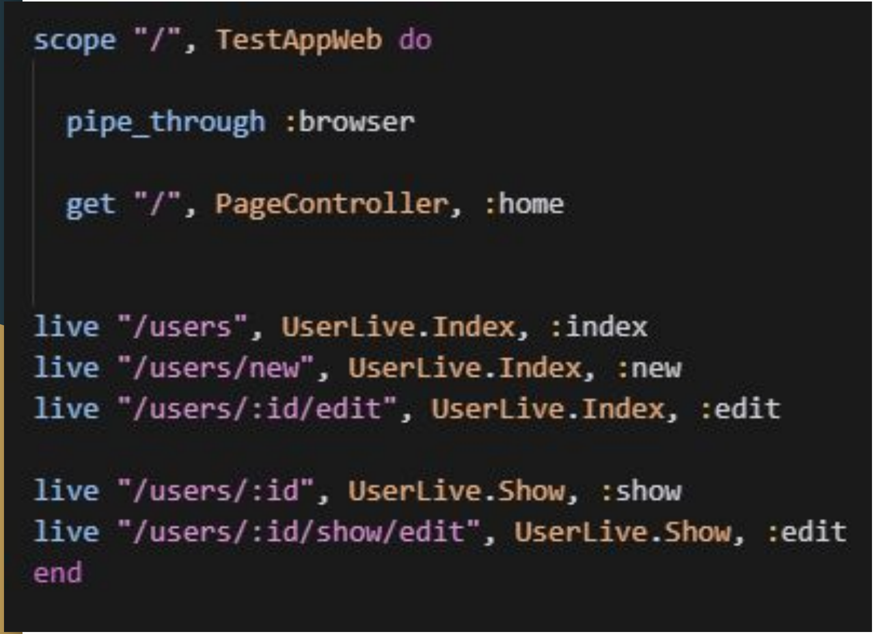
```
DELETE FROM "users" WHERE "id" = $1 [8]
```

Warn: false

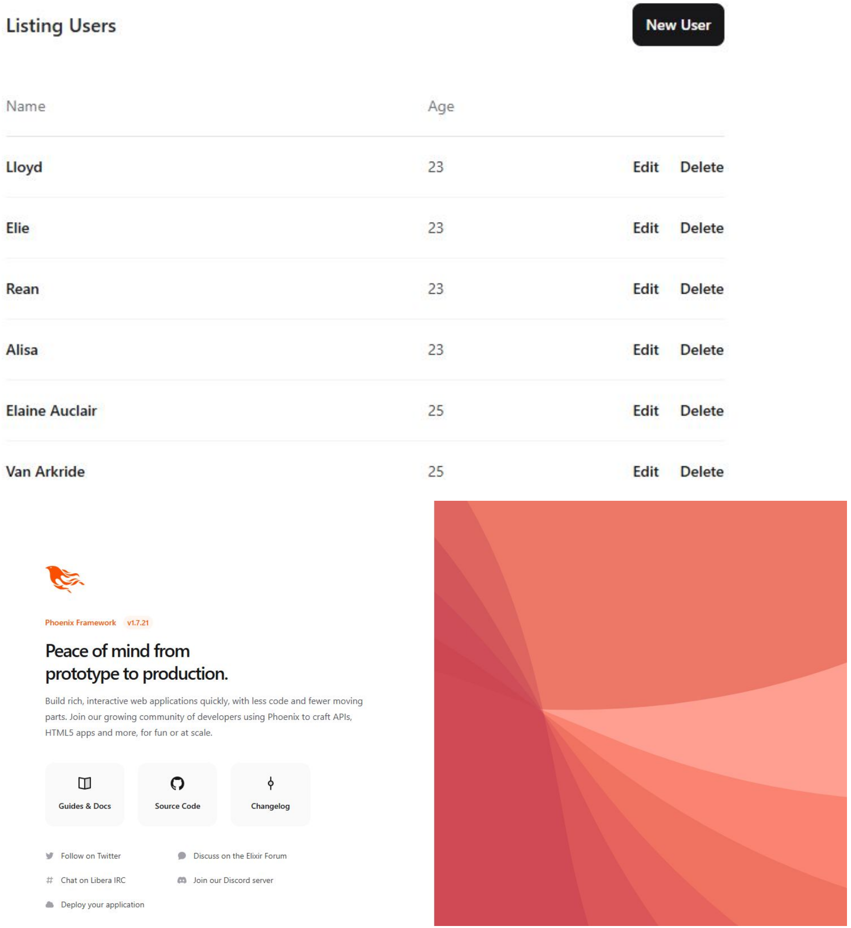
- Its function is to warn user if there’s an unused used code were detected
- If change to “true” it will highlight the unused code to alert the user

Read (List User)

Router.ex



```
scope "/", TestAppWeb do

  pipe_through :browser

  get "/", PageController, :home


live "/users", UserLive.Index, :index
live "/users/new", UserLive.Index, :new
live "/users/:id/edit", UserLive.Index, :edit

live "/users/:id", UserLive.Show, :show
live "/users/:id/show/edit", UserLive.Show, :edit
end
```



Listing Users

Name	Age		
Lloyd	23	Edit	Delete
Elie	23	Edit	Delete
Rean	23	Edit	Delete
Alisa	23	Edit	Delete
Elaine Auclair	25	Edit	Delete
Van Arkride	25	Edit	Delete

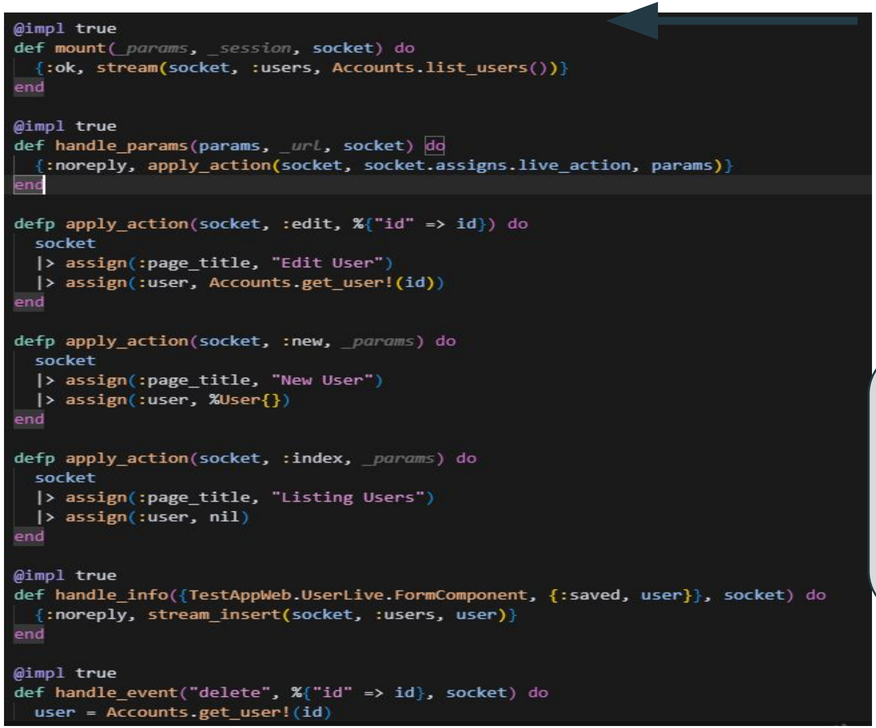
Phoenix Framework v1.7.21

Peace of mind from prototype to production.

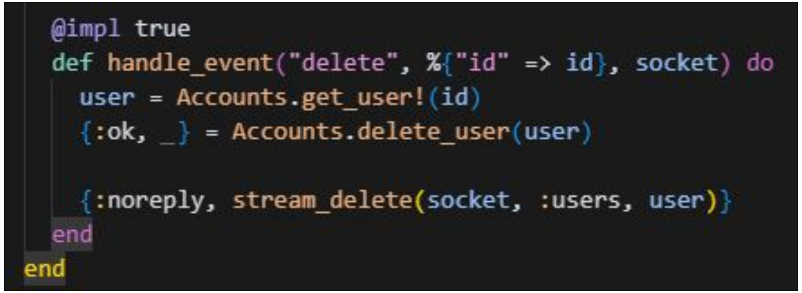
Build rich, interactive web applications quickly, with less code and fewer moving parts. Join our growing community of developers using Phoenix to craft APIs, HTML5 apps and more, for fun or at scale.

Read (List User)

index.ex



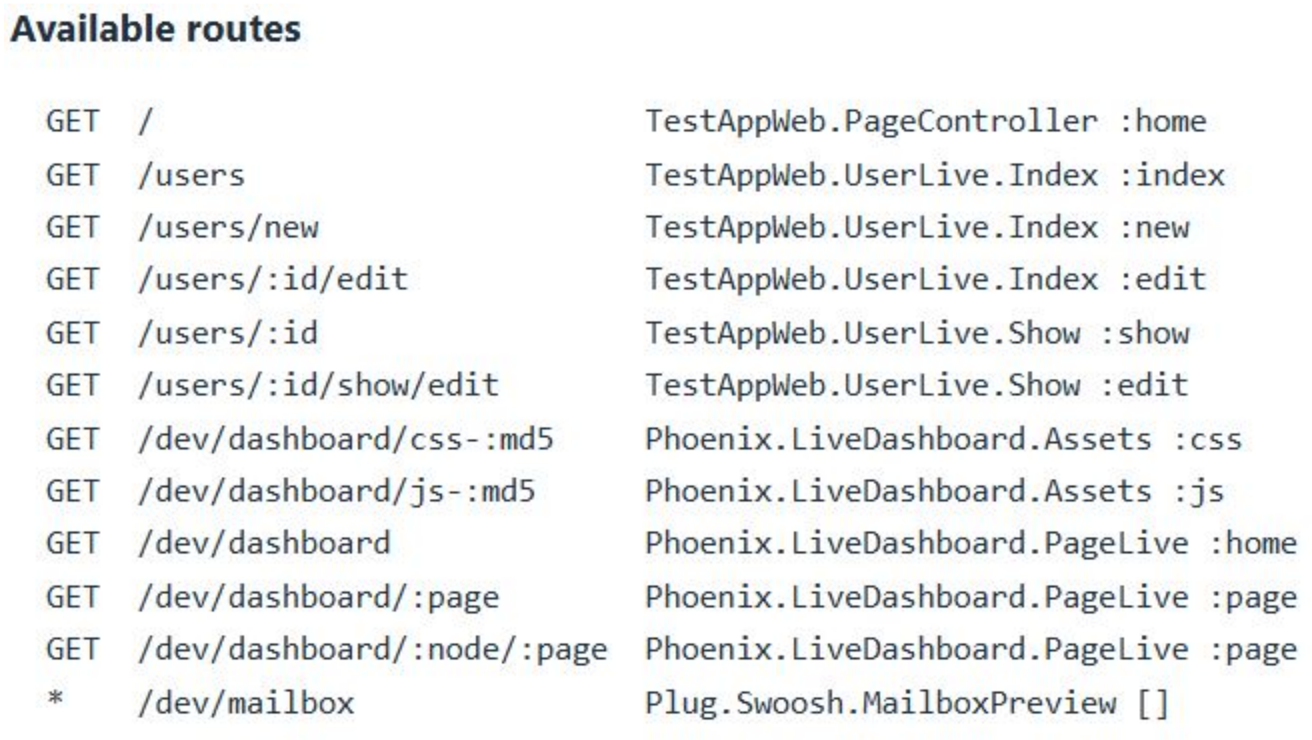
```elixir
@impl true
def mount(_params, _session, socket) do
  {:ok, stream(socket, :users, Accounts.list_users())}
end

@impl true
def handle_params(params, _url, socket) do
  {:noreply, apply_action(socket, socket.assigns.live_action, params)}
end

defp apply_action(socket, :edit, %{"id" => id}) do
  socket
  |> assign(:page_title, "Edit User")
  |> assign(:user, Accounts.get_user!(id))
end

defp apply_action(socket, :new, _params) do
  socket
  |> assign(:page_title, "New User")
  |> assign(:user, %User{})
end

defp apply_action(socket, :index, _params) do
  socket
  |> assign(:page_title, "Listing Users")
  |> assign(:user, nil)
end

@impl true
def handle_info({TestAppWeb.UserLive.FormComponent, {:saved, user}}, socket) do
  {:noreply, stream_insert(socket, :users, user)}
end

@impl true
def handle_event("delete", %{"id" => id}, socket) do
  user = Accounts.get_user!(id)
```



```elixir
  @impl true
  def handle_event("delete", %{"id" => id}, socket) do
    user = Accounts.get_user!(id)
    {:ok, _} = Accounts.delete_user(user)

    {:noreply, stream_delete(socket, :users, user)}
  end
end
```

1. **socket:** Contain data that travels between server and browser
2. **Socket.assigns.live_action:** Activate action the page is supposed to perform
3. **params:** Update info from the url

Read (List User)



Available routes

```
GET  /                              TestAppWeb.PageController :home
GET  /users                         TestAppWeb.UserLive.Index :index
GET  /users/new                     TestAppWeb.UserLive.Index :new
GET  /users/:id/edit                TestAppWeb.UserLive.Index :edit
GET  /users/:id                     TestAppWeb.UserLive.Show :show
GET  /users/:id/show/edit           TestAppWeb.UserLive.Show :edit
GET  /dev/dashboard/css-:md5        Phoenix.LiveDashboard.Assets :css
GET  /dev/dashboard/js-:md5         Phoenix.LiveDashboard.Assets :js
GET  /dev/dashboard                 Phoenix.LiveDashboard.PageLive :home
GET  /dev/dashboard/:page           Phoenix.LiveDashboard.PageLive :page
GET  /dev/dashboard/:node/:page     Phoenix.LiveDashboard.PageLive :page
*    /dev/mailbox                   Plug.Swoosh.MailboxPreview []
```

UPDATE

List of New Term Learned

Liveview- manages the page’s real time behavior

LiveComponent- is a reusable module, often used for forms like inserting or editing data,help structure UI.

Handle_params - perform call back

phoenix.submit,phx.change ,Handle_event - handle **client-side events** like clicks, form submits, key presses, save. how LiveView reacts to user actions

assign_new(:form, ...) - initializes the form with changesets.

<.simple_form> - how the form is built

Liveaction - an **assign (a variable in socket.assigns)** that acts as a **mode indicator**

Rander-producing what user sees

UPDATE

USER CLICKS "Edit"
↓
LiveView loads user and shows FormComponent
↓
Form shows data → User edits and submits
↓
LiveComponent calls update_user()
↓
✓ If OK → save to DB, redirect, flash
✗ If Error → show form errors

UPDATE

[1] You click "Edit" button on a user row

↓

[2] LiveView navigates to "/users/:id/edit" or shows a modal

↓

[3] LiveView loads the user and passes it to FormComponent

↓

[4] FormComponent renders form with current user data

↓

[5] You change the form and click "Save"

↓

[6] LiveComponent receives "save" event with new data

↓

[7] Calls Accounts.update_user(user, new_data)

↓

[8] If valid → user is updated in DB

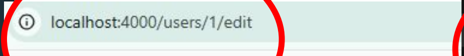
↓

[9] FormComponent sends {:saved, user} to parent

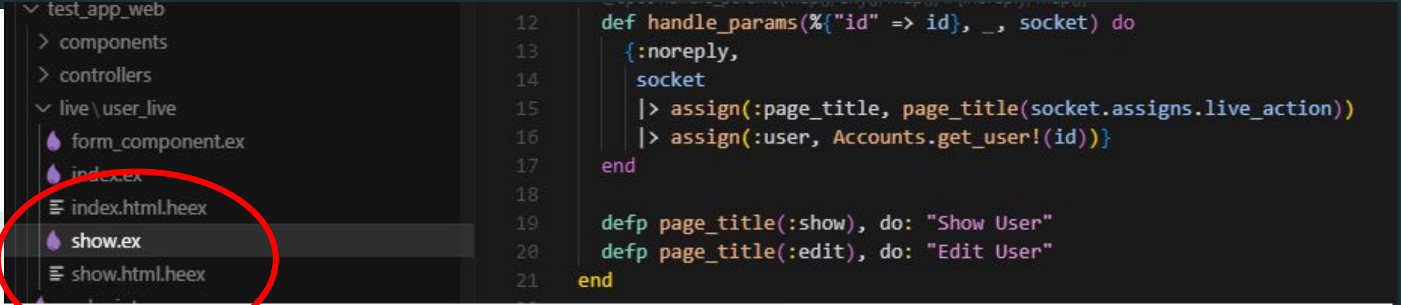
↓

[10] Parent LiveView updates list / shows flash / redirects

UPDATE

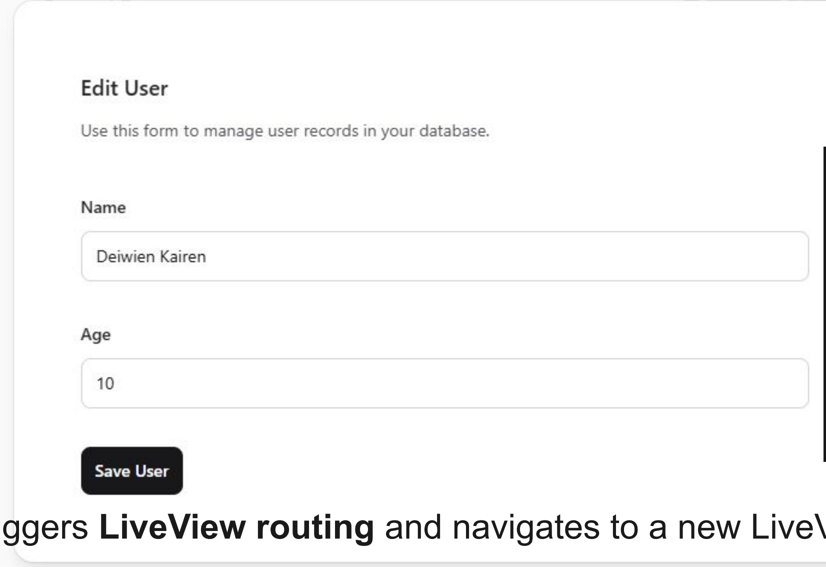


localhost:4000/users/1/edit



```elixir
  def handle_params(%{"id" => id}, _, socket) do
    {:noreply,
     socket
     |> assign(:page_title, page_title(socket.assigns.live_action))
     |> assign(:user, Accounts.get_user!(id))}
  end

  defp page_title(:show), do: "Show User"
  defp page_title(:edit), do: "Edit User"
end
```

handle_params/3 loads the user



Edit User

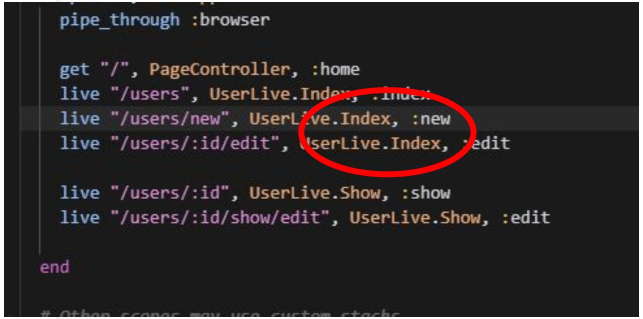
Use this form to manage user records in your database.

Name

Deiwien Kairen

Age

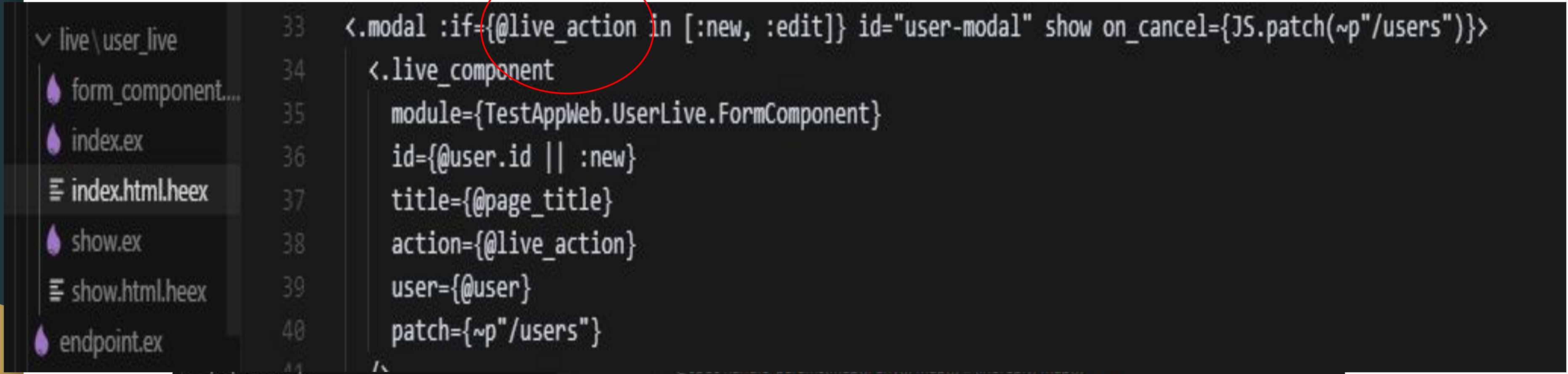
10



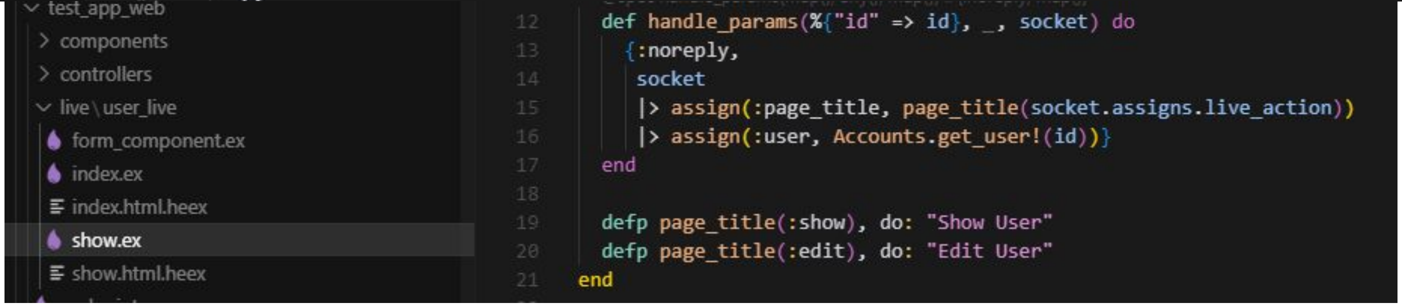
```elixir
    pipe_through :browser

    get "/", PageController, :home
    live "/users", UserLive.Index, :index
    live "/users/new", UserLive.Index, :new
    live "/users/:id/edit", UserLive.Index, :edit

    live "/users/:id", UserLive.Show, :show
    live "/users/:id/show/edit", UserLive.Show, :edit
  end

  # Other scopes may use custom stacks
```

1. This triggers **LiveView routing** and navigates to a new LiveView URL

UPDATE It sets up the form with the user's current data



```
<.modal :if={@live_action in [:new, :edit]} id="user-modal" show on_cancel={JS.patch(~p"/users")}>
  <.live_component
    module={TestAppWeb.UserLive.FormComponent}
    id={@user.id || :new}
    title={@page_title}
    action={@live_action}
    user={@user}
    patch={~p"/users"}
  />
```

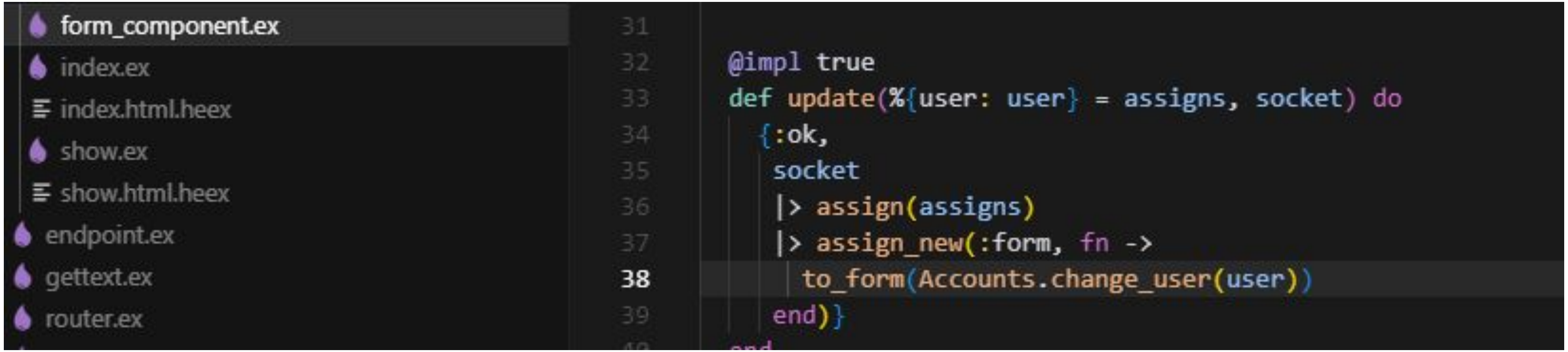


```
  def handle_params(%{"id" => id}, _, socket) do
    {:noreply,
     socket
     |> assign(:page_title, page_title(socket.assigns.live_action))
     |> assign(:user, Accounts.get_user!(id))}
  end

  defp page_title(:show), do: "Show User"
  defp page_title(:edit), do: "Edit User"
end
```

UPDATE

FormComponent renders the form with current user data

`FormComponent update/2`

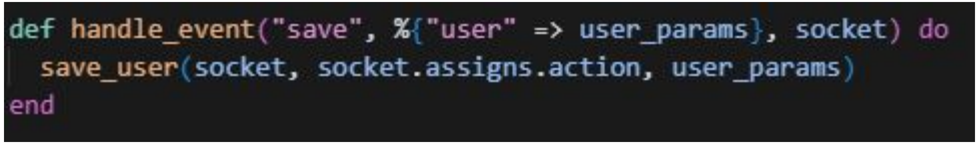


```elixir

  @impl true
  def update(%{user: user} = assigns, socket) do
    {:ok,
     socket
     |> assign(assigns)
     |> assign_new(:form, fn ->
       to_form(Accounts.change_user(user))
     end)}
  end
```

UPDATE

5. You edit the form and click “Save”

handle_event("save", ...) Gets Called



```
def handle_event("save", %{"user" => user_params}, socket) do
  save_user(socket, socket.assigns.action, user_params)
end
```

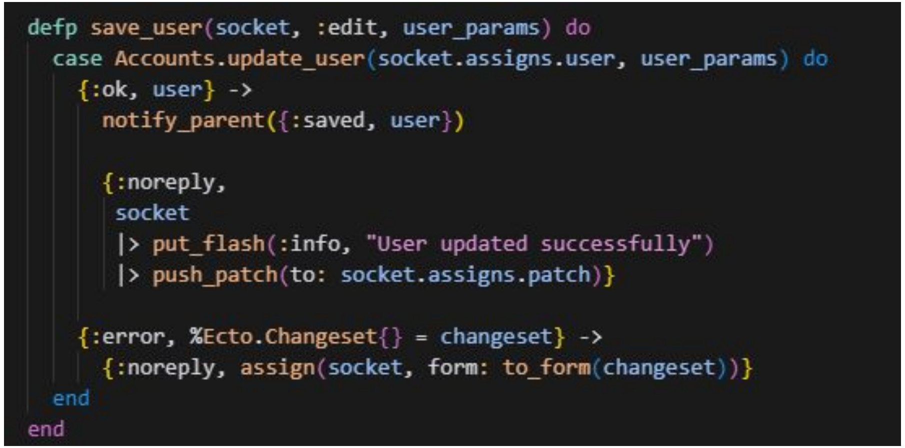
UPDATE

LiveComponent calls update function

`user_params` contains the new data from the form.

Calls `Accounts.update_user/2`, passing:

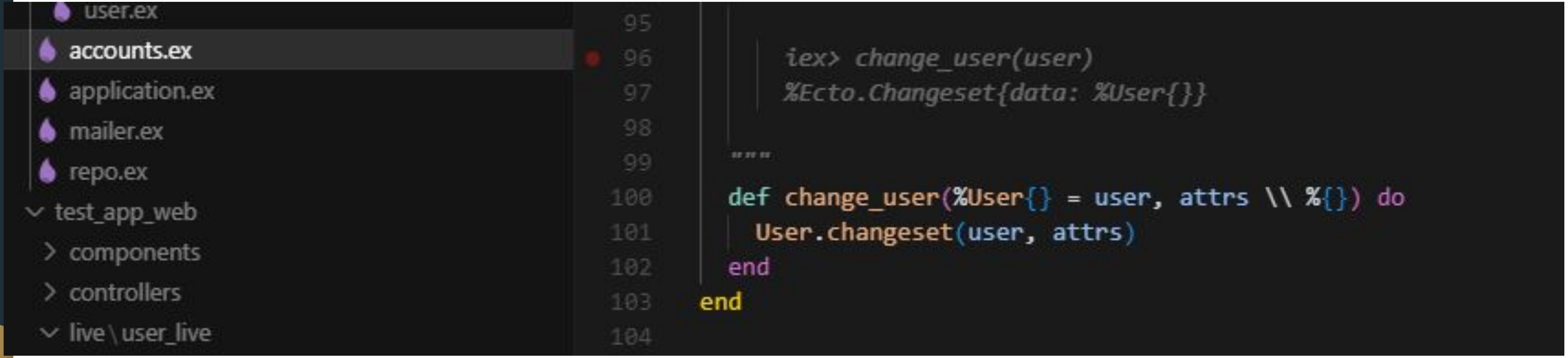
The original user
The new input data from the form



```
defp save_user(socket, :edit, user_params) do
  case Accounts.update_user(socket.assigns.user, user_params) do
    {:ok, user} ->
      notify_parent({:saved, user})

      {:noreply,
       socket
       |> put_flash(:info, "User updated successfully")
       |> push_patch(to: socket.assigns.patch)}

    {:error, %Ecto.Changeset{} = changeset} ->
      {:noreply, assign(socket, form: to_form(changeset))}
  end
end
```

UPDATE

Accounts.update_user/2 validates and saves



```
      iex> change_user(user)
      %Ecto.Changeset{data: %User{}}

  """
  def change_user(%User{} = user, attrs \\ %{}) do
    User.changeset(user, attrs)
  end
end
```

Builds a changeset from old + new data

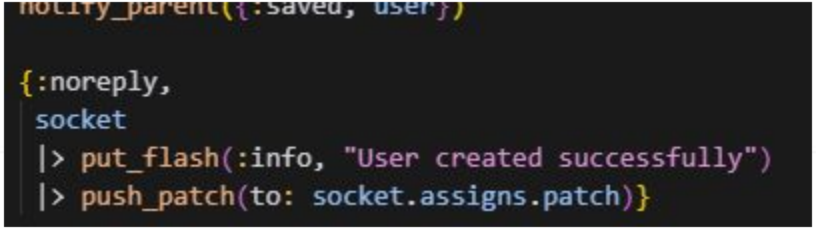
Runs validations

If valid → updates the DB `Repo.update()`

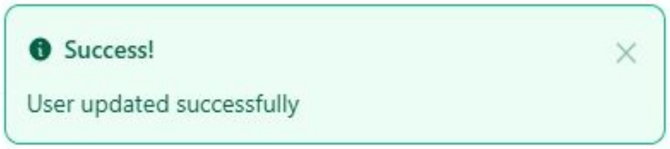
Ecto and Repo are used to interact with the database.

UPDATE

Success: notify parent and redirect

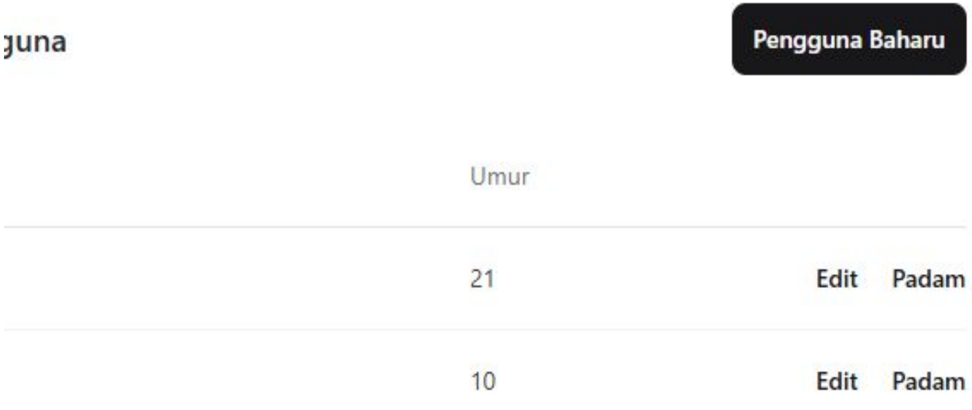


```
notify_parent({:saved, user})

{:noreply,
 socket
 |> put_flash(:info, "User created successfully")
 |> push_patch(to: socket.assigns.patch)}
```



Success!

User updated successfully



Umur		
21	Edit	Padam
10	Edit	Padam

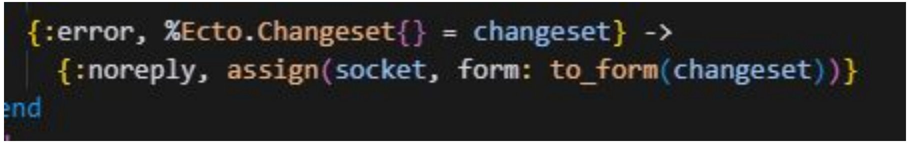
The LiveComponent sends a message to the parent.

Redirects the user back to the list page.

Shows a flash message like “User updated successfully”.

UPDATE

If validation fails



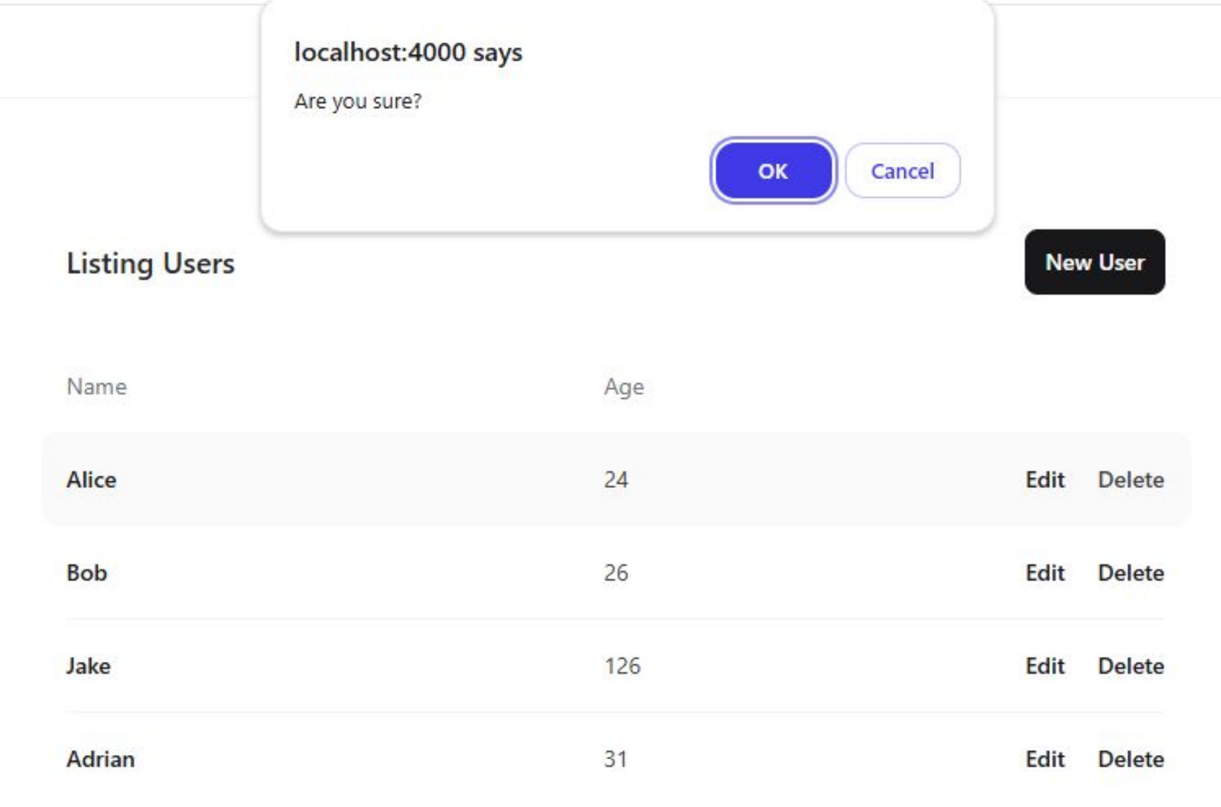
```
  {:error, %Ecto.Changeset{} = changeset} ->
    {:noreply, assign(socket, form: to_form(changeset))}
end
```

The form is re-rendered with error messages next to the fields (if your `input` component supports it).

You can fix the errors and try again.

Delete

Delete data from database

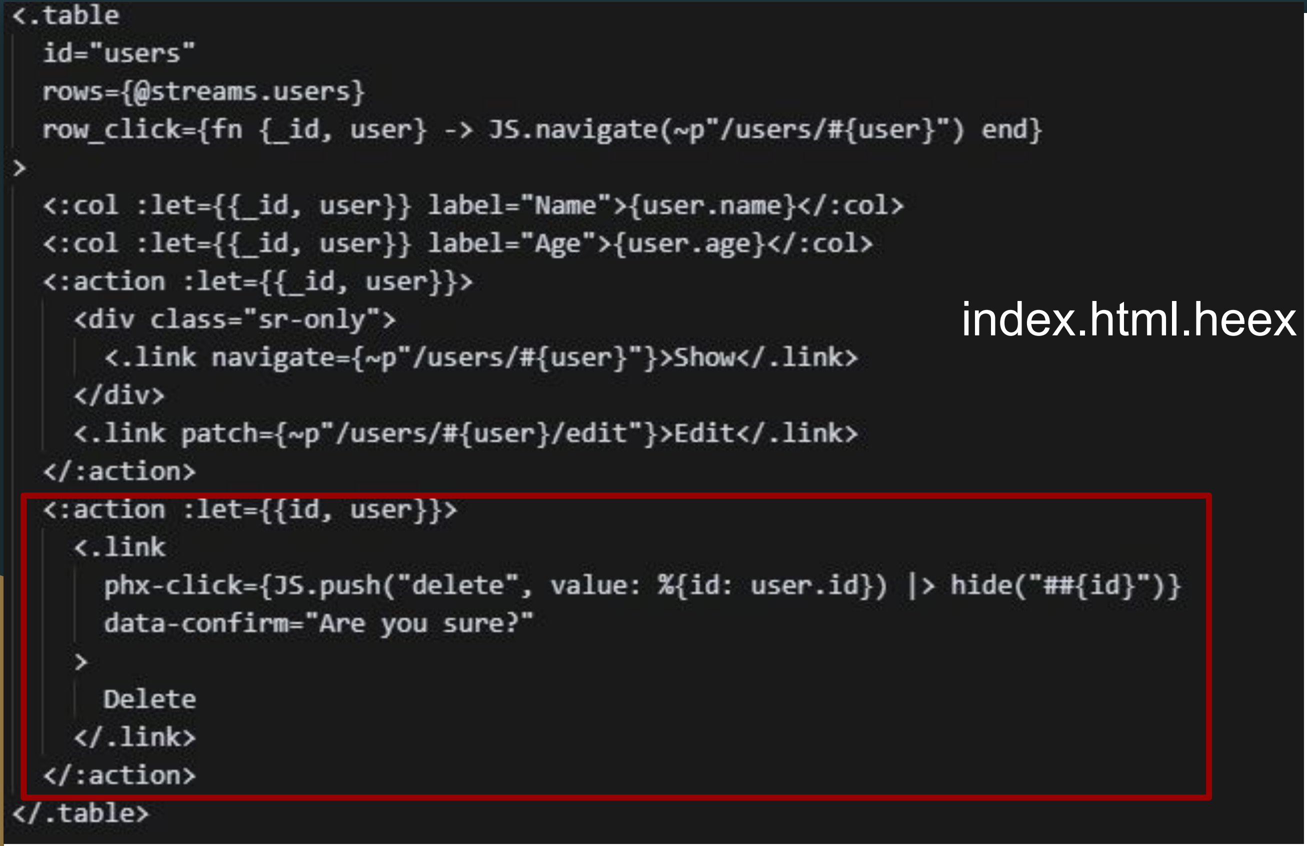


localhost:4000 says

Are you sure?

Listing Users

Name	Age		
Alice	24	Edit	Delete
Bob	26	Edit	Delete
Jake	126	Edit	Delete
Adrian	31	Edit	Delete



```
<.table
  id="users"
  rows={@streams.users}
  row_click={fn {_id, user} -> JS.navigate(~p"/users/#{user}") end}
>
  <:col :let={{_id, user}} label="Name">{user.name}</:col>
  <:col :let={{_id, user}} label="Age">{user.age}</:col>
  <:action :let={{_id, user}}>
    <div class="sr-only">
      <.link navigate={~p"/users/#{user}"}>Show</.link>
    </div>
    <.link patch={~p"/users/#{user}/edit"}>Edit</.link>
  </:action>
  <:action :let={{id, user}}>
    <.link
      phx-click={JS.push("delete", value: %{id: user.id}) |> hide("##{id}")}
      data-confirm="Are you sure?"
    >
      Delete
    </.link>
  </:action>
</.table>
```

index.html.heex

`phx-click`

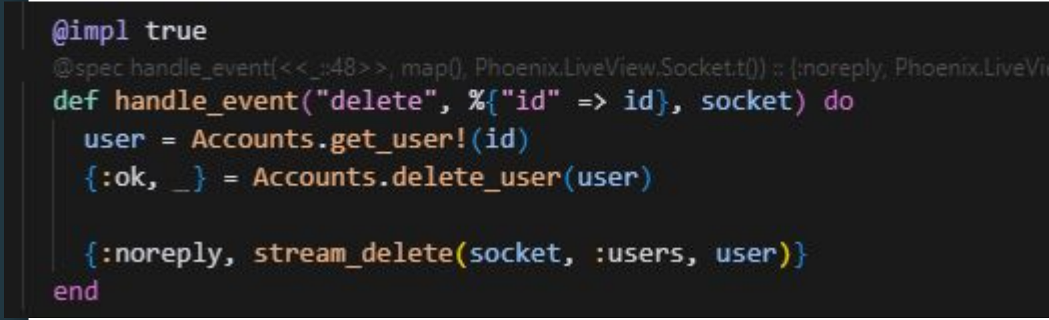
lets you respond to a click on an HTML element by sending an event to your Elixir server.

```
phx-click={JS.push("delete", value: %{id: user.id})
|> hide("##{id}")}
```

Sends a "delete" event to the LiveView with the user ID

```
|> hide("##{id}")
```

Hides the table row with the specified ID (provides immediate visual feedback)



```
@impl true
@spec handle_event(<<_::48>>, map(), Phoenix.LiveView.Socket.t()) :: {:noreply, Phoenix.LiveView.Socket.t()}
def handle_event("delete", %{"id" => id}, socket) do
  user = Accounts.get_user!(id)
  {:ok, _} = Accounts.delete_user(user)

  {:noreply, stream_delete(socket, :users, user)}
end
```

index.ex

```
def handle_event("delete",
%{"id" => id}, socket) do
```

This means: “When an event named `"delete"` is received from button click, the event data includes a map `{"id" => id}`

The `socket` represents the current LiveView state/connection.

```
user = Accounts.get_user!(id)
```

Get the user from the database by ID.

If the user is not found, it raises an error

```
{:ok, _} = Accounts.delete_user(user)
```

Deletes that user from the database.

The pattern match `{:ok, _}` ensures the deletion succeeded (and crashes if it didn't).

```
{:noreply, stream_delete
(socket, :user, user)
```

`stream_delete` updates the **stream** (a live list) and **removes the user** from the `:users` list.

`:noreply` tells Phoenix: “I updated the page, but don’t send a new reply to the browser manually

```
def delete_user(%User{} = user)
do
    Repo.delete(user)
end
```

account.ex