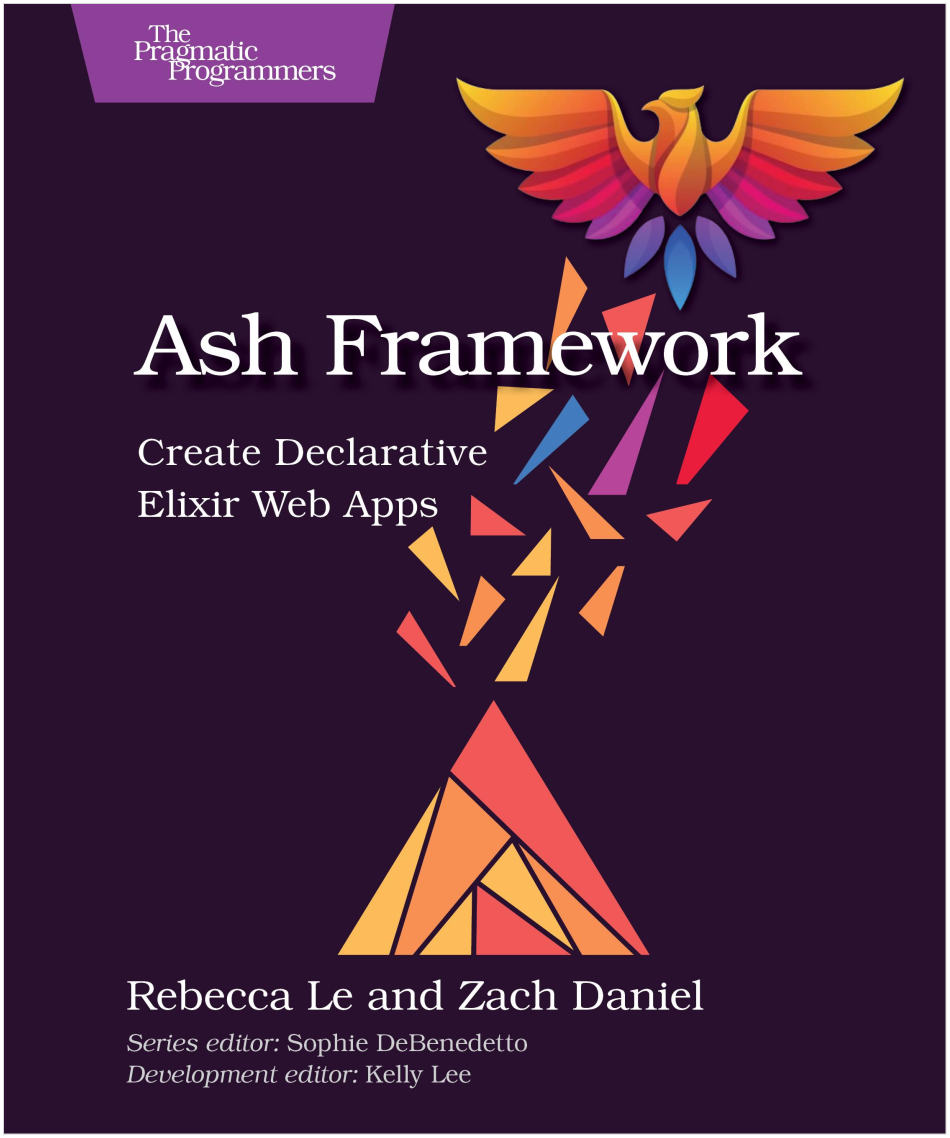

The Pragmatic Programmers
Ash Framework
Create Declarative Elixir Web Apps
Rebecca Le and Zach Daniel
Series editor: Sophie DeBenedetto
Development editor: Kelly Lee

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

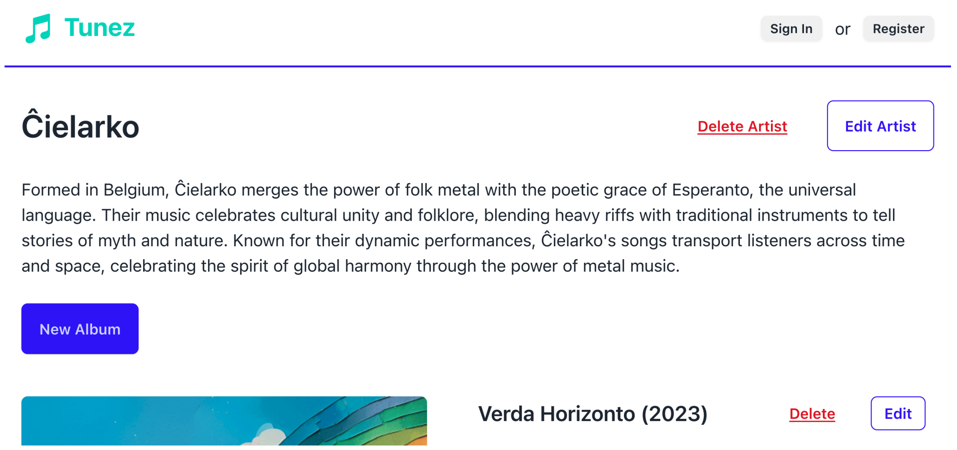
For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

Copyright © The Pragmatic Programmers, LLC.

Removing Forbidden Actions from the UI

At the moment, the Artist resource in Tunez is secure — actions that modify data can only be called if a) we pass in a user record as the actor and b) that actor is authorized to run that action. So far, so good.

The web UI doesn't reflect these changes, though. Even when not logged in to the app, we can still see buttons and forms inviting us to create, edit, or delete data.



Formed in Belgium, Ĉielarko merges the power of folk metal with the poetic grace of Esperanto, the universal language. Their music celebrates cultural unity and folklore, blending heavy riffs with traditional instruments to tell stories of myth and nature. Known for their dynamic performances, Ĉielarko's songs transport listeners across time and space, celebrating the spirit of global harmony through the power of metal music.

We can't actually *run* the actions, so clicking the buttons and submitting the forms will return an error, but it's not a good user experience to see them at all. And if we *are* logged in, so we *should* have access to manage data, we *still* get an error! Oops.

There are a few things we need to do, to make the UI behave correctly for any kind of user viewing it:

- Update all of our direct action calls to pass the current user as the actor
- Update our forms to ensure we only let the current user see them if they can submit them
- And lastly, update our templates to only show buttons if the current user is able to use them.

It sounds like a lot, but it's only a few changes to make, spread across a few different files. Let's dig in!

Identifying the actor when calling actions

For a more complex app, this would be the biggest change from a functionality perspective — allowing actions to be called by users who *are* authorized to do things. Tunez is a lot simpler, and most of the data management is done via forms, so this isn't a massive change for us. The only actions we call directly are read and destroy actions:

Tunez.Music.search_artists/2, in Tunez.Artists.IndexLive. We don't strictly *need* to pass the actor in here, as our current policies will allow the action for everyone even if they're not authenticated, but that could change in the future. If it does, we don't want to forget to set the actor, so we may as well do it now!

06/lib/tunez_web/live/artists/index_live.ex
```
def handle_params(params, _url, socket) do
  # ...

  page =
    Tunez.Music.search_artists!(query_text,
      page: page_params,
      query: [sort_input: sort_by],
➤     actor: socket.assigns.current_user
    )
```

Tunez.Music.get_artist_by_id/2, in Tunez.Artists.ShowLive. Again, we don't strictly need it because everyone can read artist information, but like the search_artists action, we can't guarantee that this will always be the case in the future. It does no harm to set the actor either, so we'll add it.

06/lib/tunez_web/live/artists/show_live.ex
```
def handle_params(%{"id" => artist_id}, _session, socket) do
  artist =
    Tunez.Music.get_artist_by_id!(artist_id,
      load: [:albums],
➤     actor: socket.assigns.current_user
    )
```

Tunez.Music.destroy_artist/2, in Tunez.Artists.ShowLive. This one we *do* need to pass the actor in to make it work, as only specific types of users can delete artists.

06/lib/tunez_web/live/artists/show_live.ex
```
def handle_event("destroy_artist", _params, socket) do
  case Tunez.Music.destroy_artist(
    socket.assigns.artist,
➤   actor: socket.assigns.current_user
  ) do
    # ...
```

Tunez.Music.destroy_album/2, in Tunez.Artists.ShowLive. We haven't added policies for albums yet, but it does no harm to start tweaking our templates to support them now.

06/lib/tunez_web/live/artists/show_live.ex
```
def handle_event("destroy_album", %{"id" => album_id}, socket) do
```

```
  case Tunez.Music.destroy_album(
    album_id,
➤   actor: socket.assigns.current_user
  ) do
    # ...
```

Not too onerous! Moving forward, we'll add the actor to every action we call, to avoid this kind of rework.

Updating forms to identify the actor

This is the biggest change for Tunez, as we create and edit both artists and albums via forms. There are two parts to this: setting the actor when building the forms, and ensuring that the form is submittable.

We don't want to show the form at all if the user wouldn't be able to submit it, so we need to run the submittable check before rendering, in the mount/3 functions of Tunez.Artists.FormLive:

06/lib/tunez_web/live/artists/form_live.ex

```
def mount(%{"id" => artist_id}, _session, socket) do
  artist = Tunez.Music.get_artist_by_id!(artist_id)

  form =
    Tunez.Music.form_to_update_artist(
      artist,
➤     actor: socket.assigns.current_user
    )
➤ |> AshPhoenix.Form.ensure_can_submit!()

  # ...

def mount(_params, _session, socket) do
  form =
    Tunez.Music.form_to_create_artist(
➤     actor: socket.assigns.current_user
    )
➤   |> AshPhoenix.Form.ensure_can_submit!()
```

AshPhoenix.Form.ensure_can_submit!/1[16] is a neat little helper function that authorizes the configured action and data in the form using our defined policies, to make sure it is submittable. If the authorization fails, then the form can't be submitted, and an exception will be raised.

We can make the same changes to the mount/3 functions in Tunez.Albums.FormLive:

06/lib/tunez_web/live/albums/form_live.ex

```
def mount(%{"id" => album_id}, _session, socket) do
  album = Tunez.Music.get_album_by_id!(album_id, load: [:artist])
```

16. https://hexdocs.pm/ash_phoenix/AshPhoenix.Form.html#ensure_can_submit!/1

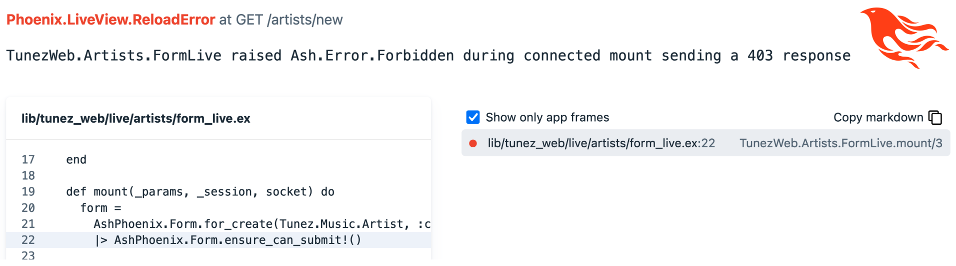
```
  form =
    Tunez.Music.form_to_update_album(
      album,
      actor: socket.assigns.current_user
    )
    |> AshPhoenix.Form.ensure_can_submit!()

  # ...

def mount(%{"artist_id" => artist_id}, _session, socket) do
  artist = Tunez.Music.get_artist_by_id!(artist_id)

  form =
    Tunez.Music.form_to_create_album(
      transform_params: fn _form, params, _context ->
        Map.put(params, "artist_id", artist.id)
      end,
      actor: socket.assigns.current_user
    )
    |> AshPhoenix.Form.ensure_can_submit!()
```

Now if you click any of the buttons that link to pages focussed on forms, when not logged in as a user with the correct role, an exception will be raised and you'll get a standard Phoenix error page:



```
Phoenix.LiveView.ReloadError at GET /artists/new

TunezWeb.Artists.FormLive raised Ash.Error.Forbidden during connected mount sending a 403 response

lib/tunez_web/live/artists/form_live.ex

  end

  def mount(_params, _session, socket) do
    form =
      AshPhoenix.Form.for_create(Tunez.Music.Artist, :c
      |> AshPhoenix.Form.ensure_can_submit!()

lib/tunez_web/live/artists/form_live.ex:22    TunezWeb.Artists.FormLive.mount/3
```

That works well for forms — but what about entire pages? Maybe we've built an admin-only area, or we've added an Artist version history page that only editors can see. We can't use the same form helpers to ensure access, but we can prevent users from accessing what they shouldn't.

Blocking pages from unauthorized access

When we installed AshAuthenticationPhoenix, one file that the installer created was the TunezWeb.LiveUserAuth module, in lib/tunez_web/live_user_auth.ex. We haven't looked at that file yet, but we will now!

It contains several on_mount function definitions, that do different things based on the authenticated user (or lack of) — the live_user_optional function head will make sure there's always a current_user set in the socket assigns, even if it's nil; the live_user_required function head will redirect away if there's no user logged in, and the live_no_user function head will redirect away if there *is* a user logged in!

These are LiveView-specific helper functions,[17] that can be called at the root level of any liveview like so:

```
defmodule Tunez.Accounts.ForAuthenticatedUsersOnly do
  use TunezWeb, :live_view

  # or :live_user_optional, or :live_no_user
  on_mount {TunezWeb.LiveUserAuth, :live_user_required}

  # ...
```

So to block a liveview from unauthenticated users, we could drop that on_mount call with :live_user_required in that module, and the job would be done!

We can add more function heads to the TunezWeb.LiveUserAuth module as well for custom behaviour, such as role-based function heads.

06/lib/tunez_web/live_user_auth.ex

```
defmodule TunezWeb.LiveUserAuth do
  # ...

  def on_mount([role_required: role_required], _, _, socket) do
    current_user = socket.assigns[:current_user]

    if current_user && current_user.role == role_required do
      {:cont, socket}
    else
      socket =
        socket
        |> Phoenix.LiveView.put_flash(:error, "Unauthorized!")
        |> Phoenix.LiveView.redirect(to: ~p"/")

      {:halt, socket}
    end
  end
end
```

This would allow us to write on_mount calls in a liveview like:

```
defmodule Tunez.Accounts.ForAdminsOnly do
  use TunezWeb, :live_view

  on_mount {TunezWeb.LiveUserAuth, role_required: :admin}

  # ...
```

Now we can secure all of our pages really neatly, both those that are form-based, and those that aren't. We still shouldn't see any shiny tempting buttons

17. https://hexdocs.pm/phoenix_live_view/Phoenix.LiveView.html#on_mount/1

for things we can't access, though, so let's hide them if the user can't perform the actions.

Hiding calls to action that the actor can't perform

There are buttons sprinkled throughout our liveviews and components — buttons for creating, editing, and deleting artists; and for creating, updating and deleting albums. We can use Ash's built-in helpers to add general authorization checks to each of them, meaning we don't have to duplicate any policy logic and we won't need to update any templates if our policy rules change.

Ash.can?

The first helper function we'll look at is Ash.can?.[18] This is a pretty low-level function, that takes a tuple representing the action to call and an actor, runs the authorization checks for the action, and returns a boolean representing whether or not the action is authorized:

```
iex(1)> Ash.can?({Tunez.Music.Artist, :create}, nil)
false
iex(2)> Ash.can?({Tunez.Music.Artist, :create}, %{role: :admin})
true
iex(3) artist = Tunez.Music.get_artist_by_id!(«uuid»)
#Tunez.Music.Artist<id: «uuid», ...>
iex(4)> Ash.can?({artist, :update}, %{role: :user})
false
iex(5)> Ash.can?({artist, :update}, %{role: :editor})
true
```

The format of the action tuple looks a lot like how you would run the action manually, as we covered in Running actions, on page ? — building a changeset for a create action with Ash.Changeset.for_create(Tunez.Music.Artist, :create, ...), or for an update action with Ash.Changeset.for_update(artist, :update, ...).

Our liveviews and components don't call actions like this though, we use code interfaces for everything because they're a lot cleaner. Ash also defines some helper functions around authorization for code interfaces that are nicer to read, so we'll look at those next.

18. https://hexdocs.pm/ash/Ash.html#can?/3