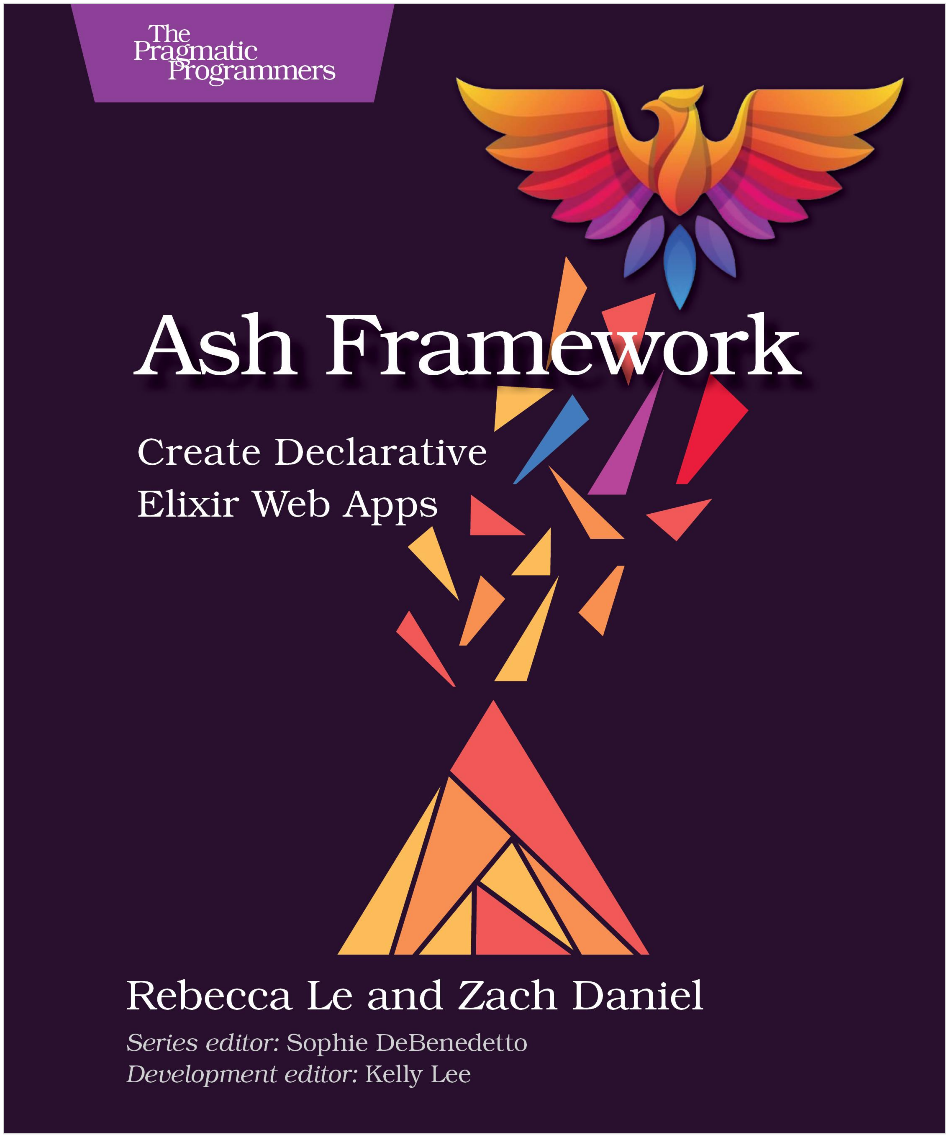
The Pragmatic Programmers
Ash Framework
Create Declarative Elixir Web Apps
Rebecca Le and Zach Daniel
Series editor: Sophie DeBenedetto
Development editor: Kelly Lee

Oh, CRUD! — Defining Basic Actions

An *action* describes an operation that can be performed for a given resource; it is the *verb* to a resource's *noun*. Actions can be loosely broken down into four types:

- Creating new persisted records (rows in the database table);
- Reading one or more existing records;
- Updating an existing record; and
- Destroying (deleting) an existing record.

These four types of actions are very common in web applications, and are often shortened to the acronym CRUD.

Ash also supports *generic actions* for any action that doesn't fit into any of those four categories. We won't be covering those in this book, but you can read the online documentation[15] about them.

With a bit of creativity, we can use these four basic action types to describe almost any kind of action we might want to perform in an app.

Registering for an account? That's a type of create action on a User resource.

Searching for products to purchase? That sounds like a read action on a Product resource.

Publishing a blog post? It could be a create action if the user is writing the post from scratch, or an update action if they're publishing an existing saved draft.

In Tunez, we'll have functionality for users to list artists and view details of a specific artist (both read actions), create and update artist records (via forms), and also destroy artist records; so we'll want to use all four types of actions. This is a great time to learn how to define and run actions using Ash, with some practical examples.

In our Artist resource, we can add an empty block for actions, and then start filling it out with what we want to be able to do:

01/lib/tunez/music/artist.ex

```
defmodule Tunez.Music.Artist do
  # ...

  actions do
```

15. https://hexdocs.pm/ash/generic-actions.html

```
  end
end
```

Let's start with creating records with a create action, so we have some data to use when testing out other types of actions.

Defining a create action

Actions are defined by adding them to the actions block in a resource. At their most basic, they require a type (one of the four mentioned earlier — create, read, update and destroy), and a name. The name can be any atom you like, but should describe what the action is actually supposed to do. It's common to give the action the same name as the action type, until you know you need something different.

01/lib/tunez/music/artist.ex

```
actions do
  create :create do
  end
end
```

To create an Artist record, we need to provide the data to be stored — in this case, the name and biography attributes, in a map. (The other attributes, such as timestamps, will be automatically managed by Ash.) We call these the attributes that the action *accepts*, and can list them in the action with the accept macro.

01/lib/tunez/music/artist.ex

```
actions do
  create :create do
    accept [:name, :biography]
  end
end
```

And that's actually all we need to do to create the most basic create action. Ash knows that the core of what a create action should do is create a data layer record from provided data, so that's exactly what it will do when we run it.

Running actions

There are two basic ways we can run actions: the generic query/changeset method, and the more direct code interface method. We can test them both out in an iex session:

```
$ iex -S mix
```

Creating records via a changeset

If you've used Ecto before, this pattern may be familiar to you:

- Create a *changeset* (a set of data changes to apply to the resource)
- Pass that changeset to Ash for processing

In code, this might look like the following:

```
Tunez.Music.Artist
|> Ash.Changeset.for_create(:create, %{
  name: "Valkyrie's Fury",
  biography: "A power metal band hailing from Tallinn, Estonia"
})
|> Ash.create()
```

We specify the action that the changeset should be created for, with the data that we want to save. When we pipe that changeset into Ash, it will handle running all of the validations and creating the record in the database.

```
iex(1)> Tunez.Music.Artist |>
...(1)> Ash.Changeset.for_create(:create, %{
...(1)>   name: "Valkyrie's Fury",
...(1)>   biography: "A power metal band hailing from Tallinn, Estonia"
...(1)> }) |>
...(1)> Ash.create()
{:ok,
 #Tunez.Music.Artist<
   __meta__: #Ecto.Schema.Metadata<:loaded, "artists">,
   id: [uuid],
   name: "Valkyrie's Fury",
   biography: "A power metal band hailing from Tallinn, Estonia",
   ...
 >}
```

The record is inserted into the database, and then returned as part of an :ok tuple. You can verify this in your database client of choice, for example, using psql tunez_dev in your terminal to connect using the inbuilt command-line client:

```
tunez_dev=# select * from artists;
-[ RECORD 1 ]----------------------------------------------
id          | [uuid]
name        | Valkyrie's Fury
biography   | A power metal band hailing from Tallinn, Estonia
inserted_at | [now]
updated_at  | [now]
```

What happens if we submit invalid data, such as an Artist without a name?

```
iex(2)> Tunez.Music.Artist |>
...(2)> Ash.Changeset.for_create(:create, %{name: ""}) |>
...(2)> Ash.create()
```

```
{:error,
 %Ash.Error.Invalid{
   changeset: #Ash.Changeset<...>,
   errors: [
     %Ash.Error.Changes.Required{
       field: :name,
       type: :attribute,
       resource: Tunez.Music.Artist,
       ...
```

The record *isn't* inserted into the database, and we get an error record back telling us what the issue is: the name is required. Later on in this chapter, we'll see how these returned errors are used when we integrate the actions into our web interface.

Like a lot of other Elixir libraries, most Ash functions return data in :ok and :error tuples. This is handy because it lets you easily pattern match on the result, to handle the different scenarios. To raise an error instead of returning an error tuple, you can use the bang version of a function ending in an exclamation mark, ie. Ash.create! instead of Ash.create.

Creating records via a code interface

If you're familiar with Ruby on Rails or ActiveRecord, this pattern may be more familiar to you. It allows us to skip the step of manually creating a changeset, and lets us call the action directly as a function.

Code interfaces can be defined on either a domain module or on a resource directly. We'd generally recommend defining them on domains, similar to Phoenix contexts, because it lets the domain act as a solid boundary with the rest of your application. With all your resources listed in your domain, it also gives a great overview of all your functionality in one place.

To enable this, we can use Ash's define macro when including the Artist resource in our Tunez.Music domain:

01/lib/tunez/music.ex
```
resources do
  resource Tunez.Music.Artist do
    define :create_artist, action: :create
  end
end
```

This will connect our domain function create_artists, to the create action of the resource. Once you've done this, if you recompile within iex the new function will now be available, complete with auto-generated documentation:

```
iex(2)> h Tunez.Music.create_artist
```

```
              def create_artist(params_or_opts \\ %{}, opts \\ [])
Calls the create action on Tunez.Music.Artist.
```

You can call it like any other function, with the data to be inserted into the database:

```
iex(7)> Tunez.Music.create_artist(%{
...(7)>   name: "Valkyrie's Fury",
...(7)>   biography: "A power metal band hailing from Tallinn, Estonia"
...(7)> })
{:ok, #Tunez.Music.Artist<...>}
```

When would I use changesets instead of code interfaces, or vice versa?

Under the hood, the code interface is creating a changeset and passing it to the domain, but that repetitive logic is hidden away. So there's no real functional benefit, but the code interface is easier to use and more readable.

Where the changeset method shines is around forms on the page — we'll see shortly how AshPhoenix provides a thin layer over the top of changesets to allow all of Phoenix's existing form helpers to work seamlessly with Ash changesets instead of Ecto changesets.

We've provided some sample content for you to play around with — you can import it into your database by running the following on the command line:

```
$ mix run priv/repo/seeds/01-artists.exs
```

There are other seed files in there too, but we'll mention those when we get to them!

Now that we have some data in our database, we can look at other types of actions.