Ash Training

ElixirConf EU 2024

Ash Training Agenda

Time	Event
08:30	Registration and welcome
09:00	**Session 1**
10:30	Tea and Coffee Break
11:00	**Session 2**
12:30	Lunch
13:30	**Session 3**
15:00	Tea and Coffee Break
15:30	**Session 4**
17:00	Finish!

Resources

What is a Resource?

- the primary concept in Ash
- usually a **domain object** in your system
    - but doesn't have to be
- a **noun** or **entity** in your domain model (eg **User**, **Order**, **Product**)
- "Model your domain (with Resources), and derive the rest"

Resources define the following

Actions: the verbs or commands for your Resources

Attributes: the data fields (with validations)

Relationships: the relationships between Resources

Data Layer: how Ash will persist data (by default it doesn't)

Note: *Resources define much, much more that we'll cover later*

Resource example

A minimal `Profile` resource with a `UUID` primary key, a `string` attribute name, and `create` and `read` actions.

```elixir
defmodule Account.Profile do
  use Ash.Resource, domain: Account

  actions do
    read :read
    create :create do
      accept [:name]
    end
  end

  attributes do
    uuid_primary_key :id
    attribute :name, :string
  end
end
```

Domain

Domains are **groupings of related resources** with shared configuration.

You can think of them like a **Phoenix Context** or a **Service**.

Note: *Was previously called **"Api"** before Ash 3.0*

Domain example

Resources must also be defined in a Domain.

```elixir
defmodule Account do
  use Ash.Domain

  resources do
    resource Account.Profile
  end
end
```

Actions are the foundation of a Resource

- Ash Resources model both data and operations (data access and transformation)
- This a fundamental difference between Ash and Ecto
- All access to a Resource's data is done via an Action
    - This allows Ash to consistently apply rules like validation, authorisation, etc

Remember: A Resource without Actions is useless!

Mandatory Attributes

By default name is optional. How can we enforce it?

```
attribute :name, :string do
  allow_nil? false
end
```

Keyword List vs Block syntax

This is called block syntax, and is usually preferred.

```
attribute :name, :string do
  allow_nil? false
end
```

can also be written inline using keyword list syntax

```
attribute :name, :string, allow_nil?: false
```

Constraining Attribute values

We want to add an attribute called status to our profile.

- type `:atom`
- can only be the values `:published` or `:unpublished`
- defaults to `:unpublished`
- cannot be `nil`

Published Attribute

```
attribute :status, :atom do
  allow_nil? false
  constraints [one_of: [:unpublished, :published]]
  default :unpublished
end
```

Timestamps

Ash provides create_timestamp and update_timestamp to keep track of when the Resource was first created, and when it was last updated.

```
  create_timestamp :created_at
  update_timestamp :updated_at
```

Putting the Attributes together

```
attributes do
  uuid_primary_key :id
  attribute :name, :string, allow_nil?: false
  attribute :status, :atom do
    allow_nil? false
    constraints [one_of: [:unpublished, :published]]
    default :unpublished
  end

  create_timestamp :created_at
  update_timestamp :updated_at
end
```

Data Layers

What is a Data Layer?

By default Ash does not persist any data.

An Ash Data Layer specifies where your Resource's data will be stored.

Which Data Layer?

- AshPostgres
- AshSqlite
- AshCsv
- Ets
- Mnesia

More to come!

You should generally default to `AshPostgres`

Specifying a Data Layer

```
defmodule Account.Profile do
  use Ash.Resource,
    domain: Account,
    data_layer: AshPostgres.DataLayer

  postgres do
    table "profiles"
    repo Account.Repo
  end

  ...
end
```

Questions?

🔬 Lab 0

Setup and Intro to Resources

- Clone the repo
  `git clone git@github.com:ash-project/ash_training`
- `cd ash_training`
- `mix setup`
- open `labs/0-resources.md` in your editor
- follow the instructions!
  # [fit] Actions

What is an Action?

- an operation or command that can be performed on a Resource
- 4 main types **Create, Read, Update** or **Destroy**
- can and should be named using Domain language where possible
- Actions can also be defined on the Domain *(since Ash 3.0)*

Use Domain Language (Not just CRUD)

For example:

- `publish` an `Article` resource making it publicly visible
- `like` or `unlike` a `Tweet` by updating the `User/Tweet` relationship
- `archive` an `Article` by soft-deleting it

These are really update action types

Default Actions Example

Actions are defined on a **Resource** in an actions block

If you just need CRUD actions, the defaults are all you need:

```
actions do
  defaults [
    :read,
    :destroy,
    create: [:name],
    update: [:name]
  ]
end
```

Accepted Attributes

Create and Update **Action**s must explicitly accept attributes they will set

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

Profile Resource

```elixir
defmodule Account.Profile do
  use Ash.Resource, domain: Account

  actions do
    read :read
    create :create do
      accept [:name]
    end
  end

  attributes do
    uuid_primary_key :id
    attribute :name, :string, allow_nil?: false
    attribute :status, :atom do
      allow_nil? false
      constraints [one_of: [:unpublished, :published]]
      default :unpublished
    end

    create_timestamp :created_at
    update_timestamp :updated_at
  end
end
```

Calling `create` Actions

To call a create Action, we use `Ash.Changeset.for_create/3` to create a changeset. Then we call `Ash.create!()`

```
profile =
  Account.Profile
  |> Ash.Changeset.for_create(:create, %{name: "My Name"})
  |> Ash.create!()
```

Note: There are nicer ways to call actions, that we'll get to later on

Action errors

Now if we create a Profile we should see an error:

```
Account.Profile
|> Ash.Changeset.for_create(:create, %{})
|> Ash.create!()
```

produces the following:

```
** (Ash.Error.Invalid) Input Invalid

* attribute name is required
```

Read All Profiles

Now, read all the generated Profiles.

```
Account.Profile
|> Ash.read!()
```

Filtered Reads

Fetching the "Joe Armstrong" Profile requires a filter

```
require Ash.Query

[joe] =
  Account.Profile
  |> Ash.Query.filter(name == "Joe Armstrong")
  |> Ash.read!()
```

Sorting and Limits

What if we want to get the latest Profile?

We set our `created_at` timestamp now we can sort on it.

```
Account.Profile
|> Ash.Query.sort(created_at: :desc)
|> Ash.Query.limit(1)
|> Ash.read_one!()
```

🔬 Lab 1
Basic Actions and Attributes

💡 Review existing application UI

☕ Break time!

---# [fit] Relationships

Relationships

Relationships describe the connections between resources, they enable:

- Loading related data
- Filtering on related data
- Managing related records through changes on a single resource
- Authorizing based on the state of related data

Relationship Basics

A Relationship exists between a source resource and a destination resource. They are defined in the `relationships` block of the source resource.

```elixir
defmodule Content.Post do
  use Ash.Resource

  attributes do
    uuid_primary_key :id
    attribute :title, :string
  end

  relationships do
    belongs_to :author, Account.Profile
  end
end
```

Kinds of Relationships

There are 4 kinds of relationships:

- `belongs_to`
- `has_one`
- `has_many`
- `many_to_many`

Belongs To

`belongs_to` links a `source_attribute` to a `destination_attribute` on another Resource

```
# on Content.Post
belongs_to :author, Account.Profile
```

The source attribute on `Content.Post` is `:author_id` and the destination attribute on `Account.Profile` is `:id`.

Has One

`has_one` is similar to a `belongs_to` except the reference attribute is on the destination resource, instead of the source.

```
# on Account.Profile
has_one :avatar, Account.Avatar
```

The source attribute on `Account.Profile` is `:id` and the destination attribute on `Account.Avatar` is `:profile_id`. This expects that `profile_id` is unique on Avatar.

Has Many

`has_many` relationship is similar to a `has_one` except that the destination attribute is not unique, and will produce a list of related items.

```
# on Account.Profile
has_many :posts, Content.Post
```

The `source_attribute` on `Account.Profile` is `:id` and the `destination_attribute` on `Content.Post` is `:profile_id`.

Many To Many

A `many_to_many` relationship can be used to relate many source resources to many destination resources.

To achieve this, the `source_attribute` and `destination_attribute` are defined on a **join resource**.

A `many_to_many` relationship can be thought of as a combination of a `has_many` relationship on the source/destination resources and a `belongs_to` relationship on the join resource.

🔬 Lab 2

Relationships

Advanced Actions

Custom Read Actions with `prepare`

For `read` Actions, you can add custom behavior with `prepare`.

Let's create a meaningful `latest` Action which sorts by the most recently created Profiles.

```
read :latest do
  prepare build(sort: [created_at: :desc])
end
```

Remember: **actions should be meaningful in your domain, not just CRUD**.

Running the Actions

```
Account.Profile
|> Ash.Query.for_read(:latest)
|> Ash.read!()
```

Builtin vs Custom prepares

`build` is a builtin prepare function (or "preparation")

```
prepare build(sort: [created_at: :desc])
```

but we can call our own code by providing a Module

```
prepare Account.Profile.Preparations.SortByMostRecentlyCreated
```

Custom prepare Module

To define a Preparation we use `Ash.Resource.Preparation` and define a prepare/3 function.

```
defmodule Account.Profile.Preparations.SortByMostRecentlyCreated do
  use Ash.Resource.Preparation

  def prepare(query, _, _) do
    Ash.Query.build(query, sort: [created_at: :desc])
  end
end
```

Custom Actions with `change` and `validate`

For `create`, `update` and `destroy` Actions, you can add custom behavior with `change` and validations with `validate`.

Let's create a meaningful `publish` Action for Profile which changes the `status`.

```
update :publish do
  # We don't want to accept any input here
  accept []

  change set_attribute(:status, :published)
  validate string_length(:name, min: 2, max: 255)
end
```

Remember: **actions should be meaningful in your domain, not just CRUD**.

Running the Actions

Given a Profile in a `profile` variable

```
profile
|> Ash.Changeset.for_update(:publish)
|> Ash.update!()
```

Builtin vs Custom change

`set_attribute` is a builtin change

```
  change set_attribute(:status, :published)
```

but we can call our own code by providing a Module

```
  change Account.Profile.Changes.Publish
```

Custom change Module

To define a change we use `Ash.Resource.Change` and define a change/3 function.

```
defmodule Account.Profile.Changes.Publish do
  use Ash.Resource.Change

  def change(changeset, _, _) do
    Ash.Changeset.force_change_attribute(changeset, :status, :published)
  end
end
```

Builtin vs Custom `validate`

`string_length` is a builtin validation

```
  validate string_length(:name, min: 2, max: 255)
```

but we can call our own code by providing a Module

```
  validate Account.Profile.Validations.CheckNameLength
```

Custom `validate` Module

To define a validation we use `Ash.Resource.Validation` and define a `validate/3` function.

```
defmodule Account.Profile.Validations.CheckNameLength do
  use Ash.Resource.Validation

  def validate(changeset, _, _) do
    name = Ash.Changeset.get_attribute(changeset, :name)
    length = String.length(name)

    if length >= 2 and length <= 255 do
      :ok
    else
      {:error, field: :name, message: "must be at least 2 characters and less than 255"}
    end
  end
end
```

Identities

What is an Identity?

Identities declare that a record can be uniquely identified by some attributes.

The primary key of the Resource is an Identity by default.

```
identities do
  identity <name>, <keys>
  ...
end
```

Identity example

To make the Profile name unique add this section to the Resource

```
identities do
  identity :profile_name, [:name]
end
```

How Does Ash Handle Identities?

Allows fields to be passed to `Ash.get/3`

```
   Ash.get(Resource, %{email: "foo@bar.com"})
```

Create unique constraints in the database automatically for each identity (AshPostgres)

Upserts

Ash automatically handles upserting on primary key, but you need to specify upsert behaviour for other attributes.

```
create :create_or_publish do
  accept [:name]
  change set_attribute(status: :published)
  upsert? true
  upsert_identity :profile_name
end
```

🔬 Lab 3

Advanced Actions

Calculations and Aggregates

Calculations

Calculations are derived fields. They can reference attributes, calculations and aggregates.

```
defmodule Resource do
  ...

  calculations do
    calculate <name>, <type>, expr(<expression>)
  end
end
```

Split name in the Profile resource

```
defmodule Account.Profile do
  use Ash.Resource,
    domain: Tutorial.Accounts

  actions do
    defaults [:read]

    create :create do
      accept [:first_name, :last_name]
    end
  end

  attributes do
    uuid_primary_key :id
    attribute :first_name, :string, allow_nil?: false
    attribute :last_name, :string, allow_nil?: false
  end
end
```

Calculate `full_name`

Add the `calculations` section to the Resource

```
calculations do
  calculate :full_name, :string, expr(first_name <> " " <> last_name)
  # or reference a Module
  calculate :full_name, :string, Account.Profile.Calculations.FullName
end
```

`full_name` can now be loaded on demand or used in filters, sorts, or other calculations.

Module Calculations

Not every calculation can be created with an expression.

```
defmodule Account.Profile.Calculations.FullName do
  use Ash.Resource.Calculation

  def load(_, _, _), do: [:first_name, :last_name]

  def calculate(records, _, _) do
    Enum.map(records, fn record ->
      record.first_name <> " " <> record.last_name
    end)
  end
end
```

Expressions with `expr`

Ash expressions give you a way to define **portable calculations**.
This means that they are data layer independent! For example:

```
expr(first_name <> " " <> last_name)
```

This can be run in Elixir *or* within a data layer.
This allows efficient sorting and filtering.

Expression Examples

```
Post
|> Ash.Query.filter(full_name == "Jim Freeze")
|> Ash.read!()
```

```
SELECT *
FROM users
WHERE (first_name || ' ' || last_name) = 'Jim Freeze'
```

Expression Examples

```
expr = Ash.Expr.expr(first_name <> " " <> last_name)

profile = %Account.Profile{
  first_name: "Jim",
  last_name: "Freeze"
}

Ash.Expr.eval(expr, record: profile)
# {:ok, "Jim Freeze"}
```

Calculation example

Let's put that together

```
joe =
  Account.Profile
  |> Ash.Changeset.for_create(:create, %{first_name: "Joe", last_name: "Armstrong"})
  |> Ash.create!()
  |> Ash.load!([:full_name])
```

Loading the `full_name` calculates the field, and concats the attributes.

Aggregates

Aggregates in Ash allow for retrieving summary information over groups of related data.

Some examples:
- count of published Posts for a User
- sum of all read counts across all Posts for a User

Aggregate Example

Given a user `Profile` resource with related Posts:

```
aggregates do
  count :count_of_posts, :posts do
    filter expr(published == true)
  end
end
```

Aggregate Types

- `count` - counts related items meeting the criteria.
- `exists` - checks if any related items meet the criteria.
- `first` - gets the first related value matching the criteria. Must specify the `field`.
- `sum` - sums the related items meeting the criteria. Must specify the `field`.
- `list` - lists the related values. Must specify the `field`.

Aggregate Types (more)

- `max` - gets the maximum related value. Must specify the `field`.
- `min` - gets the minimum related value. Must specify the `field`.
- `avg` - gets the average related value. Must specify the `field`.
- `custom` - allows for a custom aggregate. Implementation depends on the data layer. Must provide an `implementation`.

🔬 Lab 4 & 5

Calculations and Aggregates

:burger: Lunch!

Actors, Authorization & Policies

Actors & Authorization

Authorization in Ash involves three things:

- `actor` - the entity (i.e User, Org, Device) performing an action
- `authorize?` - a flag that tells Ash to run authorization.
- `authorizers` - the extensions on a resource that can modify or forbid the action.

Setting actor and authorize?

All functions in Ash that may perform authorization and/or wish to use the actor accept an actor and an authorize? option.

```
Ash.Changeset.for_create(
  Post,
  %{title: "Post Title"},
  actor: current_user,
  authorize?: true
)
```

Set the actor on the query/changeset/input

The hooks on a query/changeset/input to an action may need to know the actor

```elixir
# DO THIS
Post
|> Ash.Query.for_read(:read, actor: current_user)
|> Ash.read!()

# DON'T DO THIS
Post
|> Ash.Query.for_read!(:read)
|> Ash.read!(actor: current_user)
```

Authorizers

Authorizers are in control of what happens during authorization.

Generally, you won't need to create your own authorizer, as the builtin policy authorizer `Ash.Policy.Authorizer` works well for any use case.

```
use Ash.Resource, authorizers: [Ash.Policy.Authorizer]
```

⚠ If you don't add at least one Authorizer, your Resource allows any actor to call any action.

Policies

Policies determine what actions on a resource are permitted for a given actor.

They can also filter the results of read actions to restrict the results to only records that should be visible.

Setup

You'll need to add the extension to your resource, like so:

```
use Ash.Resource, authorizers: [Ash.Policy.Authorizer]
```

Then you can start defining policies for your resource.

Policy Example

```
policies do
  policy always() do
    authorize_if always()
  end

  policy action_type(:create) do
    authorize_if IsSuperUser
    forbid_if Deactivated
    authorize_if IsAdminUser
    authorize_if HasCreatorRole
  end
end
```

Anatomy of a Policy

Each Policy defined in a resource has two parts:

1. a condition, or a list of conditions such as `action_type(:read)` or `always()`. If the condition(s) are true for an attempted action, then the policy will be applied to the action.
2. a set of **Checks**, each of which will be evaluated individually if a Policy applies to the attempted action.

How a Policy is processed

If more than one policy applies to any given attempted action (eg. an admin actor calls a read action) then **all applicable policies must pass** for the action to be performed.

A Policy will evaluate to either:

- `:forbidden`
- `:authorized`

Policy checks

If no check produces a result then the Policy result is `:forbidden`

- `authorize_if`
  - if `true` the whole policy is `:authorized`
  - else move to next check
- `authorize_unless`
  - if `false` the whole policy is `:authorized`
  - else move to next check
- `forbid_if`
  - if `true` the whole policy is `:forbidden`
  - else move to next check
- `forbid_unless`
  - if `false` the whole policy is `:forbidden`
  - else move to next check

Policy Example

```
policies do
  policy action_type(:create) do
    authorize_if IsSuperUser
    forbid_if Deactivated
    authorize_if IsAdminUser
    authorize_if HasCreatorRole
  end
end
```

We check those from top to bottom, so the first one of those that returns `:authorized` or `:forbidden` determines the entire outcome.

Bypass Policies

A bypass Policy is just like a regular policy, except if a bypass passes, then other policies after it *do not need to pass*.

This can be useful for writing complex access rules, or for a simple rule like "an admin can do anything" without needing to specify it as part of every other policy.

Bypass Example

```
policies do
  bypass IsSuperUser do
    authorize_if always()
  end
end
```

Lab 6

Policies

Code Interfaces

Why do we need Code Interfaces?

Using `Changesets` and `Querys` directly to act on resources is a bit unwieldy.

Code Interfaces simplify how we use our defined actions, and offer a clean and rich interface to our Domain.

They can be defined on **Resources** (*or* the **Domain** since Ash 3.0)

Code Interface Example

In this example, we will define it on the **Domain**.

```
defmodule Account do
  use Ash.Domain

  resources do
    resource Account.Profile do
      define :create_profile, args: [:name], action: :create
    end
  end
end
```

Using Code Interfaces

Create a profile with 1 line

```
Account.Profile.create_profile!("Joe Armstrong")
```

instead of 3 lines

```
Account.Profile
|> Ash.Changeset.for_create(:create, %{name: "Joe Armstrong"})
|> Ash.create!()
```

Nice! the bang version of the function is created as well as the normal version

🔬 Lab 7

Code Interfaces

Spark

https://github.com/ash-project/spark

What is a DSL?

A little language that is designed for a very specific purpose.

https://en.wikipedia.org/wiki/Domain-specific_language

https://martinfowler.com/dsl.html

https://martinfowler.com/books/dsl.html

Why DSLs?

- Concise Representation
  - Minimal + No extraneous syntax
- Safer
  - Validation + Fewer options less room for error
- Expressive power
- Readable by SMEs
  - Allow contributions from domain experts
  - Better, more powerful and expressive abstractions
- Declarative (less programming)

Issues with macro based DSLs in Elixir

- More code, more decisions, more complexity
- Harder to maintain (it's a special snowflake)
- Harder to test
- Non standard
- Lacks some utilities like formatter help no parens makes DSLs more readable typically
- Doesn't have LS support for editor hints
- Not easily extensible
- Still need to document your DSL

What if we could build **Elixir DSLs** without writing macros?

Why Spark?

- Easier and safer
  - no need to write macros, less plumbing
- Standardised structure
  - focus on designing the DSL to solve the business problem
  - don’t write your own special snowflake
- Extensibility
  - other libs can come and extend
- Autocomplete and hover help via Elixir Language Server
- Auto docs, no docs for struct data declarations
- Formatting helpers (no parens for your DSL generated for you)

Spark History

- Started as part of Ash core
- Powers all of the Ash ecosystem
- Battle tested
    - but needs some doc love
- Split out relatively recently
    - Aug 2022 (!18 months)
    - Was part of Ash core from the beginning

Spark Basic Concepts

`Spark.Dsl.Section`

- must have at least one top level section
- can be hidden with `top_level?: true`
- a singleton - there can be only one
  - multiple instances are merged together automatically
- can contain nested section and entities

`Spark.Dsl.Entity`

- repeatable list items
- entities can contain other entities
- some thoughts about merging with Section, but big impacts on Ash ecosystem