Get Your CRUD Together

Andrew Ek, ElixirConf EU 2022

Principal Engineer at Launch Scout

`@ektastrophe` on Twitter

`andrew.ek@launchscout.com`

https://github.com/andrewek/prairie


Some Horror Stories


Motivations

Motivations

→ Improve speed to market

→ Reduce cognitive load

→ Maintain fast-enough performance

One Version of Layers

1. Work-starter
2. Container
3. Bounded Context
4. Data Access
5. Functional Core


The Data Access Layer

The Data Access Layer

Our data might live in:

→ A database

→ A GenServer

→ An ETS Table

→ Some remote service or third party application

The Data Access Layer

With a "conventional" approach we might see these problems:

→ Variations in data representation

→ Failed book-keeping after business events

→ Small changes have big effects

→ Overreliance on callbacks

Patterns I Like

→ Strict separation between business logic and book-keeping

→ Finding aggregates in context

→ Composable Queries

→ Sensible defaults (e.g. `GenericRepo`)

Business Logic or Book-Keeping?

Finding Aggregates in Context


Composable Queries

Composable Queries

→ Pipeline-able operations

→ Takes a query and returns a query

→ Reads like Elixir, rather than like SQL

Composable Queries

→ Describes meaning, rather than implementation

→ Standardizes "shape" of data

→ Reduces cognitive overhead


The GenericRepo

The `GenericRepo`

→ List all (or paginate)

→ Get one

→ Create

→ Update

→ Delete

→ Count

The GenericRepo


Some Variants

Some Variants

Instead of `%Something{}` or `nil`, consider `{:ok, %Something{}}` and `{:error, :not_found}`

Or `{:error, Something, :not_found}`

Some Variants

You might want callbacks (e.g. Pub/Sub) upon certain successful operations

Some Variants

```
def update(%Ecto.Changeset{} = changeset) do
  changeset
  |> Repo.update()
  |> maybe_publish(:updated)
end

defp maybe_publish({:ok, record}, action) do
  publish(record, action)

  {:ok, record}
end

defp maybe_publish(result, _action) do
  result
end
```

Some Variants

```
def update(%Ecto.Changeset{} = changeset, callback_fn \\ nil) do
  changeset
  |> Repo.update()
  |> maybe_success_callback(callback_fn)
end

defp maybe_success_callback({:ok, record}, callback_fn) when is_function(callback_fn) do
  callback_fn.(record)

  {:ok, record}
end

defp maybe_success_callback(result, _callback_fn) do
  result
end
```

Some Variants

Depending on your interface, consider using many small changesets rather than one big one

Some Variants

Don't be afraid to access the same data in different ways depending on context. A given table may appear in several contexts.

```elixir
defmodule Prairie.Bison.Repo do
  use GenericRepo,
    schema: Prairie.Bison.Bison,
    base_repo: Prairie.Repo,
    default_preloads: [
      prairie: [],
      keepers: [],
      offspring: []
    ]
end
```

```
defmodule Prairie.VeterinaryCare.BisonRepo do
  use GenericRepo,
    schema: Prairie.Bison.Bison,
    base_repo: Prairie.Repo,
    default_preloads: [
      check_up_charts: [],
      records: [],
      veterinarian: [],
      vaccinations: []
    ]
end
```

Some Variants

Consider telemetry and other monitoring. At some point you may outgrow this approach, and it's good to know when this happens.

Some Variants

We can use GenServers, ETS tables, Redis, etc., to cache data.


Return to Motivating Factors

Return to Motivating Factors

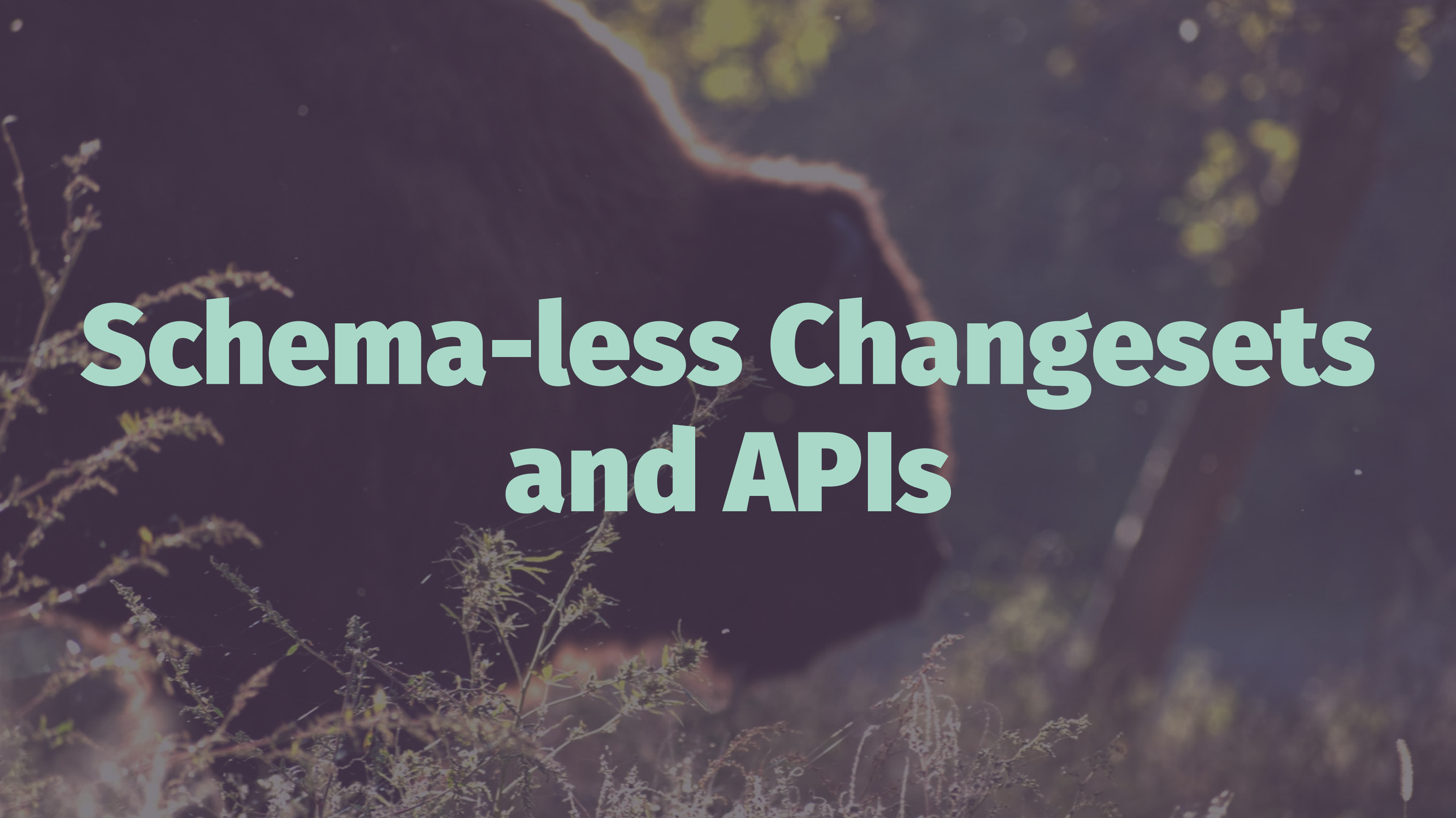
→ Reduce bugs caused by subtle variants

→ Reduce cognitive overhead

→ Improve speed to market

→ Better (more normalized) data model


Why Might We *Not* Do This?


Schema-less Changesets
and APIs

Schema-less Changesets and APIs

If we're using external APIs, we can use schema-less changesets (and a pseudo-repo) to maintain a consistent interface.

Schema-less Changesets and APIs

```elixir
defmodule Praire.VaxxaPro.Record do
  import Ecto.Changeset
  alias __MODULE__

  defstruct [:name, :date, :bison_id, :dose]

  @types %{id: :string, name: :string, dose: :integer, date: :string, bison_id: :string}

  def changeset(%Record{} = record, attrs \\ %{}) do
    {record, @types}
    |> cast(attrs, [:name, :date, :bison_id, :dose])
    |> validate_required([:name, :date, :bison_id, :dose])
  end
end
```

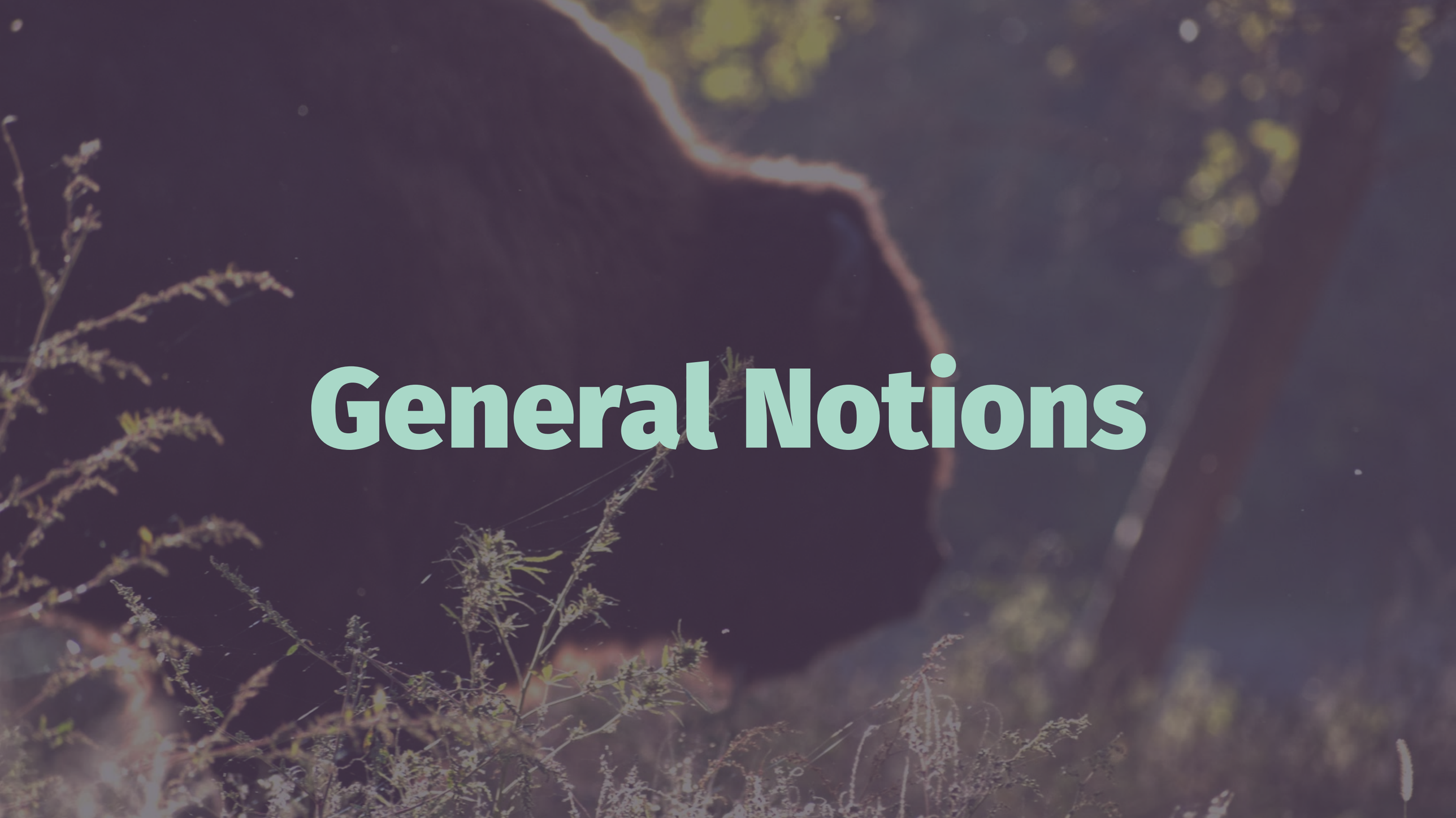
Schema-less Changesets and APIs

```
defmodule Praire.VaxxaPro.Repo do
  alias Praire.VaxxaPro.Record

  def insert(attrs \\ %{}) do
    %Record{}
    |> Record.changeset(attrs)
    |> post()
    |> parse()
  end
end
```

Schema-less Changesets and APIs

1. User input
2. Changeset (ensure well-formed)
3. Post to remote API
4. Transform result into `{:ok, Something{}}` or `{:error, changeset}`



General Notions

General Notions

→ Separate book-keeping from business logic

→ Prefer consistent representations (within a context) where possible

→ Structs, aggregates, and the "Repo" pattern help build consistency

→ Don't be afraid to normalize your database

→ Be mindful of performance v. speed to delivery


Fin

https://github.com/andrewek/prairie

`@ektastrophe` on Twitter

`andrew.ek@launchscout.com`