EMBRACE **THE** DATABASE **WITH ECTO**

ABOUT ME

- JOSH BRANCHAUD
- SOFTWARE DEVELOPER AT HASHROCKET

I HAVE SOME OPINIONS ABOUT DATABASES.

THE DATABASE IS YOUR FRIEND

... BUT DATABASES ARE SCARY!

- COMPOSITE INDEXES
- FULL OUTER JOINS
- COMMON TABLE EXPRESSIONS
- EXPLAIN ANALYZE OUTPUT

YOUR DATABASE IS NOT JUST A

DUMB DATA STORE

IT IS A POWERFUL COMPUTATION ENGINE

THE DATA STORED IN YOUR DATABASE IS THE SINGLE MOST IMPORTANT ASSET IN THE LIFE OF YOUR PRODUCT/BUSINESS.

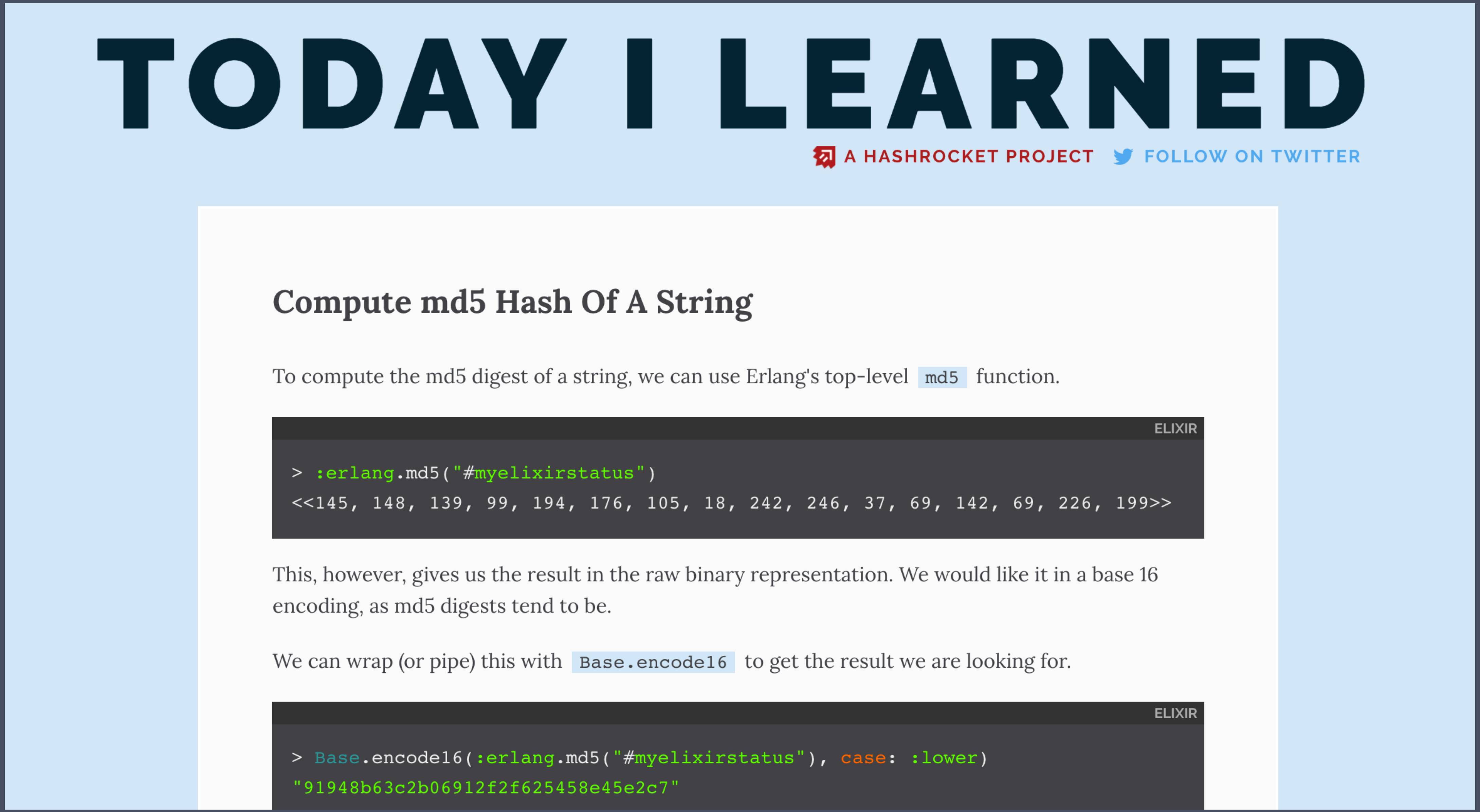
THE BEST DATABASE FOR WEB APPLICATION IS...

PostgreSQL

AGENDA

- DATA INTEGRITY
- SCHEMALESS QUERIES
- ECTO'S ESCAPE HATCH
- ENHANCING ECTO WITH CUSTOM FUNCTIONS

OUR SAMPLE DATA SOURCE



TODAY I LEARNED

A HASHROCKET PROJECT FOLLOW ON TWITTER

Compute md5 Hash Of A String

To compute the md5 digest of a string, we can use Erlang's top-level `md5` function.

```elixir
> :erlang.md5("#myelixirstatus")
<<145, 148, 139, 99, 194, 176, 105, 18, 242, 246, 37, 69, 142, 69, 226, 199>>
```

This, however, gives us the result in the raw binary representation. We would like it in a base 16 encoding, as md5 digests tend to be.

We can wrap (or pipe) this with `Base.encode16` to get the result we are looking for.

```elixir
> Base.encode16(:erlang.md5("#myelixirstatus"), case: :lower)
"91948b63c2b06912f2f625458e45e2c7"
```

WHAT IS TIL?

TIL IS AN OPEN-SOURCE PROJECT BY THE TEAM AT HASHROCKET THAT CATALOGUES THE SHARING & ACCUMULATION OF KNOWLEDGE AS IT HAPPENS DAY-TO-DAY.

(CHECK IT OUT - TIL.HASHROCKET.COM)

TIL'S DB SCHEMA

- POSTS
- DEVELOPERS
- CHANNELS

POSTS TABLE

```
> \d posts
                                      Table "public.posts"
    Column    |            Type             |                       Modifiers
--------------+-----------------------------+----------------------------------------------------
 id           | integer                     | not null default nextval('posts_id_seq'::regclass)
 developer_id | integer                     | not null
 body         | text                        | not null
 created_at   | timestamp without time zone | not null
 updated_at   | timestamp without time zone | not null
 channel_id   | integer                     | not null
 title        | character varying           | not null
 slug         | character varying           | not null
 likes        | integer                     | not null default 1
 tweeted      | boolean                     | not null default false
 published_at | timestamp with time zone    |
 max_likes    | integer                     | not null default 1
Indexes:
    "posts_pkey" PRIMARY KEY, btree (id)
    "index_posts_on_channel_id" btree (channel_id)
    "index_posts_on_developer_id" btree (developer_id)
Check constraints:
    "likes" CHECK (likes >= 0)
Foreign-key constraints:
    "fk_rails_447dc2e0a3" FOREIGN KEY (channel_id) REFERENCES channels(id)
    "fk_rails_b3ec63b3ac" FOREIGN KEY (developer_id) REFERENCES developers(id)
```

DEVELOPERS TABLE

```
> \d developers
                                       Table "public.developers"
     Column     |            Type             |                         Modifiers
----------------+-----------------------------+-----------------------------------------------------------
 id             | integer                     | not null default nextval('developers_id_seq'::regclass)
 email          | character varying           | not null
 username       | character varying           | not null
 created_at     | timestamp without time zone | not null
 updated_at     | timestamp without time zone | not null
 twitter_handle | character varying           |
 admin          | boolean                     | not null default false
 editor         | character varying           | default 'Text Field'::character varying
 slack_name     | character varying           |
Indexes:
    "developers_pkey" PRIMARY KEY, btree (id)
Referenced by:
    TABLE "posts" CONSTRAINT "fk_rails_b3ec63b3ac" FOREIGN KEY (developer_id) REFERENCES developers(id)
```

CHANNELS TABLE

```
> \d channels
                                        Table "public.channels"
     Column      |            Type             |                        Modifiers
-----------------+-----------------------------+----------------------------------------------------------
 id              | integer                     | not null default nextval('channels_id_seq'::regclass)
 name            | text                        | not null
 created_at      | timestamp without time zone | not null
 updated_at      | timestamp without time zone | not null
 twitter_hashtag | character varying(20)       | not null
Indexes:
    "channels_pkey" PRIMARY KEY, btree (id)
Check constraints:
    "twitter_hashtag_alphanumeric_constraint" CHECK (twitter_hashtag::text ~ '^[\w\d]+$'::text)
Referenced by:
    TABLE "posts" CONSTRAINT "fk_rails_447dc2e0a3" FOREIGN KEY (channel_id) REFERENCES channels(id)
```

DATA

SO MANY ANSWERS JUST WAITING TO BE ASKED THE RIGHT QUESTION

ASKING QUESTIONS

HOW DO WE ASK QUESTIONS OF OUR DATA?

WE NEED A MEDIATOR

WHAT IS THE BEST MEDIATOR BETWEEN US AND OUR DATA?

SQL

SQL IS THE BEST WAY TO TALK TO OUR SQL DATABASE

HOW MANY POSTS ARE THERE?

```
sql> select count(*) from posts;
 count
-------
  1066
(1 row)
```

WHAT ABOUT ELIXIR AND ECTO?

ECTO

ECTO IS A DOMAIN SPECIFIC LANGUAGE FOR WRITING QUERIES AND INTERACTING WITH DATABASES IN ELIXIR.

HOW MANY POSTS ARE THERE?

```
iex> Repo.one(from p in "posts", select: count(p.id))
1066

17:16:36.573 [debug] QUERY OK source="posts" db=10.8ms queue=0.2ms
SELECT count(p0."id") FROM "posts" AS p0 []
```

QUERIES ARE JUST

DATA

QUERIES AS DATA

- #ECTO.QUERY STRUCT
- YOU BUILD THEM UP AS YOU GO
- YOU CAN INSPECT THEM

DATA INTEGRITY

YOUR DATABASE IS THE ULTIMATE **GATEKEEPER**

- MANY CLIENTS, MICROSERVICES
- APP-LEVEL VALIDATIONS VS DB-LEVEL VALIDATIONS
- DRY IT UP!

DATA INTEGRITY

- ENFORCE PARTICULAR DATATYPE
- USING BETTER, CUSTOM DATA TYPES (E.G. UUID, BIGINT, AND CITEXT)

```
execute("create extension if not exists citext;")

create table(:developers, primary_key: false) do
  add :id, :uuid, primary_key: true
  add :email, :citext
end
```

DATA INTEGRITY

- ENFORCE PRESENCE
- NOT NULL CONSTRAINTS

```
execute("create extension if not exists citext;")

create table(:developers, primary_key: false) do
  add :id, :uuid, primary_key: true
  add :email, :citext, null: false
end
```

DATA INTEGRITY

- ENFORCE RELATIONSHIPS
- FOREIGN KEY CONSTRAINTS

```
create table(:posts) do
  add :title, :varchar, null: false
  add :body, :text, null: false

  add :developer_id, references(:developers, type: :uuid)
end
```

DATA INTEGRITY

- ENFORCE MORE GENERAL RELATIONSHIPS
- CHECK CONSTRAINTS

```
create table(:posts) do
  add :title, :varchar, null: false
  add :body, :text, null: false
  add :likes, :smallint, null: false, default: 0

  add :developer_id, references(:developers, type: :uuid)
end

create constraint(:posts, "ensure_positive_likes", check: "likes >= 0")
```

DATA INTEGRITY

```
def up do
  execute("create extension if not exists citext;")
  execute("create extension if not exists pgcrypto;")

  create table(:developers, primary_key: false) do
    add :id, :uuid, primary_key: true, default: fragment("gen_random_uuid()")
    add :email, :citext, null: false
    add :created_at, :timestamptz, null: false, default: fragment("now()")
    add :updated_at, :timestamptz, null: false, default: fragment("now()")
  end

  create table(:posts) do
    add :title, :varchar, null: false
    add :body, :text, null: false
    add :likes, :smallint, null: false, default: 0

    add :developer_id, references(:developers, type: :uuid)

    add :created_at, :timestamptz, null: false, default: fragment("now()")
    add :updated_at, :timestamptz, null: false, default: fragment("now()")
  end

  create constraint(:posts, "ensure_positive_likes", check: "likes >= 0")
end
```

DATA INTEGRITY

```
def down do
  drop table(:posts)
  drop table(:developers)

  execute("drop extension if exists pgcrypto;")
  execute("drop extension if exists citext;")
end
```

SCHEMALESS QUERIES

```
iex> Repo.one(from p in "posts", select: count(p.id))
```

- ALWAYS START WITH A `from` CLAUSE
- `Repo.one`, `Repo.all`, ETC. TO EXECUTE
- `import Ecto.Query` AND `alias MyApp.Repo`

HOW MANY DEVELOPERS ARE THERE?

```
iex> Repo.one(from d in "developers", select: fragment("count(*)"))

17:19:01.195 [debug] QUERY OK source="developers" db=1.0ms queue=2.9ms
SELECT count(*) FROM "developers" AS d0 []

32
```

HOW MANY POSTS BY CHANNEL?

FIRST, LET'S JOIN channels ON posts

```
iex> posts_and_channels = from(p in "posts",
    join: c in "channels",
    on: p.channel_id == c.id)

#Ecto.Query<from p in "posts", join: c in "channels", on: p.channel_id == c.id>
```

HOW MANY POSTS BY CHANNEL?

USE `group_by` WITH `count(p.id)` AS OUR AGGREGATOR

```
iex> Repo.all(from([p,c] in posts_and_channels,
    group_by: c.name,
    select: {count(p.id), c.name}))

16:12:31.539 [debug] QUERY OK source="posts" db=6.8ms
SELECT count(p0."id"), c1."name" FROM "posts" AS p0 INNER JOIN "channels" AS c1 ON p0."channel_id" = c1."id" GROUP BY c1."name" []

[{13, "clojure"}, {5, "react"}, {102, "rails"}, {201, "vim"}, {59, "workflow"},
 {110, "command-line"}, {121, "sql"}, {73, "elixir"}, {1, "erlang"},
 {6, "design"}, {28, "testing"}, {5, "go"}, {15, "mobile"}, {67, "javascript"},
 {32, "devops"}, {125, "ruby"}, {17, "html-css"}, {63, "git"}, {23, "emberjs"}]
```

HOW MANY POSTS BY CHANNEL?

CLEAN UP THE RESULT WITH AN `order_by` CLAUSE

```
> Repo.all(from([p,c] in posts_and_channels,
    group_by: c.name,
    order_by: [desc: count(p.id)],
    select: {count(p.id), c.name}))

16:13:43.516 [debug] QUERY OK source="posts" db=7.3ms
SELECT count(p0."id"), c1."name" FROM "posts" AS p0 INNER JOIN "channels" AS c1 ON p0."channel_id" = c1."id" GROUP BY c1."name" ORDER BY count(p0."id") DESC []

[{201, "vim"}, {125, "ruby"}, {121, "sql"}, {110, "command-line"},
 {102, "rails"}, {73, "elixir"}, {67, "javascript"}, {63, "git"},
 {59, "workflow"}, {32, "devops"}, {28, "testing"}, {23, "emberjs"},
 {17, "html-css"}, {15, "mobile"}, {13, "clojure"}, {6, "design"}, {5, "go"},
 {5, "react"}, {1, "erlang"}]
```

HOW MANY POSTS ON AVERAGE PER DEVELOPER?

FIRST, LET'S GET POST COUNTS FOR EACH DEVELOPER

```
iex> post_counts = from(p in "posts",
                  group_by: p.developer_id,
                  select: %{post_count: count(p.id), developer_id: p.developer_id})

#Ecto.Query<from p in "posts", group_by: [p.developer_id],
 select: %{post_count: count(p.id), developer_id: p.developer_id}>
```

HOW MANY POSTS ON AVERAGE PER DEVELOPER?

```
iex> Repo.all(post_counts)

10:29:09.177 [debug] QUERY OK source="posts" db=5.8ms
SELECT count(p0."id"), p0."developer_id" FROM "posts" AS p0 GROUP BY p0."developer_id" []
[%{developer_id: 14, post_count: 6}, %{developer_id: 25, post_count: 43},
 %{developer_id: 32, post_count: 1}, %{developer_id: 27, post_count: 2},
 %{developer_id: 8, post_count: 332}, %{developer_id: 17, post_count: 1},
 %{developer_id: 15, post_count: 23}, %{developer_id: 1, post_count: 1},
 %{developer_id: 10, post_count: 18}, %{developer_id: 26, post_count: 78},
 %{developer_id: 11, post_count: 15}, %{developer_id: 4, post_count: 130},
 %{developer_id: 18, post_count: 14}, %{developer_id: 30, post_count: 10},
 %{developer_id: 16, post_count: 3}, %{developer_id: 33, post_count: 1},
 %{developer_id: 6, post_count: 3}, %{developer_id: 19, post_count: 9},
 %{developer_id: 29, post_count: 82}, %{developer_id: 2, post_count: 236},
 %{developer_id: 23, post_count: 10}, %{developer_id: 31, post_count: 5},
 %{developer_id: 20, post_count: 8}, %{developer_id: 5, post_count: 3},
 %{developer_id: 13, post_count: 3}, %{developer_id: 22, post_count: 12},
 %{developer_id: 9, post_count: 10}, %{developer_id: 24, post_count: 4},
 %{developer_id: 7, post_count: 3}]
```

HOW MANY POSTS ON AVERAGE PER DEVELOPER?

```
iex> Repo.aggregate(subquery(post_counts), :avg, :post_count)

10:29:45.425 [debug] QUERY OK db=13.0ms queue=0.1ms
SELECT avg(s0."post_count") FROM (SELECT count(p0."id") AS "post_count",
p0."developer_id" AS "developer_id" FROM "posts" AS p0 GROUP BY
p0."developer_id") AS s0 []

#Decimal<36.7586206896551724>
```

Ecto.Repo.aggregate

:avg | :count | :max | :min | :sum

```
iex> Repo.aggregate("posts", :count, :id)

10:02:11.862 [debug] QUERY OK source="posts" db=21.8ms
SELECT count(p0."id") FROM "posts" AS p0 []

1066
```

SCHEMALESS QUERIES

LET'S TRY SOMETHING A BIT MORE COMPLEX

WHAT IS THE CHANNEL AND TITLE OF EACH DEVELOPER'S MOST LIKED POST IN 2016?

COMPLEX QUERIES

WRITING COMPLEX QUERIES IS ALL ABOUT BUILDING THE SOLUTION FROM THE GROUND UP.

PIECE BY PIECE.

WHAT IS THE CHANNEL AND TITLE OF EACH DEVELOPER'S MOST LIKED POST IN 2016?

FIRST, LET'S JOIN OUR TABLES TOGETHER

```
iex> posts_devs_channels = from(p in "posts",
     join: d in "developers", on: d.id == p.developer_id,
     join: c in "channels", on: c.id == p.channel_id)

#Ecto.Query<from p in "posts", join: d in "developers",
 on: d.id == p.developer_id, join: c in "channels", on: c.id == p.channel_id>
```

WHAT IS THE CHANNEL AND TITLE OF EACH DEVELOPER'S MOST LIKED POST IN 2016?

NEXT, WE CAN COMBINE `order_by` AND `distinct`

```
iex> from([posts, devs, channels] in posts_devs_channels(),
     distinct: devs.id,
     order_by: [desc: posts.likes],
     select: %{
       dev: devs.username,
       channel: channels.name,
       title: posts.title
     }
)

#Ecto.Query<from p in "posts", join: d in "developers", on: true,
 join: c in "channels", on: d.id == p.developer_id and c.id == p.channel_id,
 order_by: [desc: p.likes], distinct: [asc: d.id],
 select: %{dev: d.username, channel: c.name, title: p.title}>
```

WHAT IS THE CHANNEL AND TITLE OF EACH DEVELOPER'S MOST LIKED POST IN 2016?

NOW, LET'S CONSTRAIN THE RESULTS TO 2016

```
iex> top_of_2016 = from([posts, devs, channels] in posts_devs_channels(),
     distinct: devs.id,
     order_by: [desc: posts.likes],
     where: posts.created_at > ^Ecto.DateTime.cast!({{2016,1,1},{0,0,0}}),
     where: posts.created_at < ^Ecto.DateTime.cast!({{2017,1,1},{0,0,0}}),
     select: %{
       dev: devs.username,
       channel: channels.name,
       title: posts.title
     }
)

#Ecto.Query<from p in "posts", join: d in "developers",
 on: d.id == p.developer_id, join: c in "channels", on: c.id == p.channel_id,
 where: p.created_at > ^#Ecto.DateTime<2016-01-01 00:00:00>,
 where: p.created_at < ^#Ecto.DateTime<2017-01-01 00:00:00>,
 order_by: [desc: p.likes], distinct: [asc: d.id],
 select: %{dev: d.username, channel: c.name, title: p.title}>
```

WHAT IS THE CHANNEL AND TITLE OF EACH DEVELOPER'S MOST LIKED POST IN 2016?

```
iex> Repo.all(top_of_2016)

11:53:32.317 [debug] QUERY OK source="posts" db=13.5ms
SELECT DISTINCT ON (d1."id") d1."username", c2."name", p0."title" FROM
"posts" AS p0 INNER JOIN "developers" AS d1 ON d1."id" = p0."developer_id"
INNER JOIN "channels" AS c2 ON c2."id" = p0."channel_id" WHERE
(p0."created_at" > $1) AND (p0."created_at" < $2) ORDER BY d1."id",
p0."likes" DESC [{{2016, 1, 1}, {0, 0, 0, 0}}, {{2017, 1, 1}, {0, 0, 0, 0}}]

[%{channel: "elixir", dev: "developer2",
   title: "Invoke Elixir Functions with Apply"},
 %{channel: "workflow", dev: "developer4", title: "Ternary shortcut in PHP"},
 %{channel: "vim", dev: "developer5",
   title: "Use colorcolumn to visualize maximum line length"},
 %{channel: "ruby", dev: "developer6",
   title: "Ruby optional arguments can come before required"},
 %{channel: "ruby", dev: "developer7",
   title: "Using pessimistic gem version to catch betas"},
 ...]
```

SCHEMALESS QUERY FUNCTIONS IN ECTO 2.0

- `Ecto.Repo.update_all/3`
- `Ecto.Repo.insert_all/3`
- `Ecto.Repo.delete_all/3`

ESCAPE HATCH

ECTO CAN'T DO IT ALL, SOMETIMES WE NEED AN

ESCAPE HATCH

ONE-OFF QUERIES

USING `Ecto.Repo.query`

```
iex> Repo.query("select * from generate_series(1,5);")

12:00:14.801 [debug] QUERY OK db=1.5ms
select * from generate_series(1,5); []
{:ok,
 %Postgrex.Result{columns: ["generate_series"], command: :select,
  connection_id: 59379, num_rows: 5, rows: [[1], [2], [3], [4], [5]]}}
```

FRAGMENTS

THE `Ecto.Query.API.fragment` FUNCTION

FRAGMENTS IN MIGRATIONS

```
create table(:developers, primary_key: false) do
  add :id, :uuid, primary_key: true, default: fragment("gen_random_uuid()")
  add :email, :citext, null: false
  add :created_at, :timestamptz, null: false, default: fragment("now()")
  add :updated_at, :timestamptz, null: false, default: fragment("now()")
end
```

FRAGMENTS IN QUERIES

```
iex> Repo.one(from d in "developers", select: fragment("count(*)"))

17:19:01.195 [debug] QUERY OK source="developers" db=1.0ms queue=2.9ms
SELECT count(*) FROM "developers" AS d0 []
32
```

FRAGMENTS IN QUERIES

LET'S REVISIT THIS QUERY. CAN WE USE THE `between` CONSTRUCT?

```
iex> top_of_2016 = from([posts, devs, channels] in posts_devs_channels(),
       distinct: devs.id,
       order_by: [desc: posts.likes],
       where: posts.created_at > ^Ecto.DateTime.cast!({{2016,1,1},{0,0,0}}),
       where: posts.created_at < ^Ecto.DateTime.cast!({{2017,1,1},{0,0,0}}),
       select: %{
         dev: devs.username,
         channel: channels.name,
         title: posts.title
       }
)
```

FRAGMENTS IN QUERIES

```
iex> from([posts, devs, channels] in posts_devs_channels(),
     distinct: devs.id,
     order_by: [desc: posts.likes],
     where: fragment("? between ? and ?",
                     posts.created_at,
                     ^Ecto.DateTime.cast!({{2016,1,1},{0,0,0}}),
                     ^Ecto.DateTime.cast!({{2017,1,1},{0,0,0}})
                   ),
     select: %{
       dev: devs.username,
       channel: channels.name,
       title: posts.title
     }
)

#Ecto.Query<from p in "posts", join: d in "developers",
 on: d.id == p.developer_id, join: c in "channels", on: c.id == p.channel_id,
 where: fragment("? between ? and ?", p.created_at, ^#Ecto.DateTime<2016-01-01 00:00:00>, ^#Ecto.DateTime<2017-01-01 00:00:00>),
 order_by: [desc: p.likes], distinct: [asc: d.id],
 select: %{dev: d.username, channel: c.name, title: p.title}>
```

ONE STEP FURTHER

FROM CLUNKY **FRAGMENTS TO** ELEGANT **CUSTOM FUNCTIONS**

CUSTOM FUNCTIONS

WE CAN DO BETTER THAN THIS AND DRY UP OUR CODE

```
where: fragment("? between ? and ?",
                posts.created_at,
                ^Ecto.DateTime.cast!({{2016,1,1},{0,0,0}}),
                ^Ecto.DateTime.cast!({{2017,1,1},{0,0,0}})
              ),
```

CUSTOM FUNCTIONS

DEFINE A `CustomFunctions` MODULE WITH REUSABLE FRAGMENTS

```
defmodule CustomFunctions do
  defmacro between(operand, left, right) do
    quote do
      fragment("? between ? and ?", unquote(operand), unquote(left), unquote(right))
    end
  end
end
```

CUSTOM FUNCTIONS

TO USE IT, FIRST `import CustomFunctions`, THEN

```
iex> from([posts, devs, channels] in posts_devs_channels(),
     distinct: devs.id,
     order_by: [desc: posts.likes],
     where: between(posts.created_at,
                    ^Ecto.DateTime.cast!({{2016,1,1},{0,0,0}}),
                    ^Ecto.DateTime.cast!({{2017,1,1},{0,0,0}})
                  ),
     select: %{
       dev: devs.username,
       channel: channels.name,
       title: posts.title
     }
)

#Ecto.Query<from p in "posts", join: d in "developers",
 on: d.id == p.developer_id, join: c in "channels", on: c.id == p.channel_id,
 where: fragment("? between ? and ?", p.created_at, ^#Ecto.DateTime<2016-01-01 00:00:00>, ^#Ecto.DateTime<2017-01-01 00:00:00>),
 order_by: [desc: p.likes], distinct: [asc: d.id],
 select: %{dev: d.username, channel: c.name, title: p.title}>
```

THAT'S IT

THANKS! QUESTIONS?

- JOSH BRANCHAUD
- SOFTWARE DEVELOPER AT HASHROCKET
- TWITTER: @JBRANCHA