

---

## A functional query tool for Elixir and PostgreSQL.

Our goal with creating Moebius is to try and keep as close as possible to the functional nature of Elixir and, at the same time, the goodness that is PostgreSQL. We think working with a database should feel like a natural extension of the language, with as little abstraction wonkery as possible.

Moebius is *not* an ORM. There are no mappings, no schemas, no migrations; only queries and data. We embrace PostgreSQL as much as possible, surfacing the goodness so you be a hero.

## Documentation

API documentation is available at <http://hexdocs.pm/moebius>

## Installation

Installing Moebius involves a few small steps:

1. Add moebius to your list of dependencies in `mix.exs`:

```
1 def deps do
2   [{:moebius, "~> 4.0.2"}]
3 end
```

2. Add the db child process to your `Application` module's supervision tree:

```
1 children = [
2   Moebius.Db
3 ]
```

Run `mix deps.get` and you'll be good to go.

## Connecting to PostgreSQL

There are various ways to connect to a database with Moebius. You can use a formal, supervised definition or just roll with our default. Either way, you start off by adding connection info in your `config.exs`:

```
1 config :moebius, connection: [
2   hostname: "localhost",
3   username: "username",
4   password: "password",
5   database: "my_db"
6 ],
```

---

```
7 scripts: "test/db"
```

You can also use a URL if you like:

```
1 config :moebius, connection: [  
2   url: "postgresql://user:password@host/database"  
3 ],  
4 scripts: "test/db"
```

You can also configure custom Postgres Extensions:

```
1 config :moebius,  
2   connection: [url: "postgresql://user:password@host/database"],  
3   types: PostgresTypes
```

And define your custom types in your application under `lib/postgres_types.ex`

```
1 types = [Geo.PostGIS.Extension, Some.Custom.Extension]  
2 opts = [json: Jason]  
3  
4 Postgrex.Types.define(PostgresTypes, types, opts)
```

If you want to use environment variables, just set things using `System.env`.

Under the hood, Moebius uses the Postgrex driver to manage connections and connection pooling. Connections are supervised, so if there's an error any transaction pending will be rolled back effectively (more on that later). The settings you provide in `:connection` will be passed directly to Postgrex (aside from `:url`, which we parse).

You might be wondering what the `scripts` entry is? Moebius can execute SQL files directly for you - we'll get to that in a bit.

## Supervision and Databases

Moebius formalizes the concept of a database connection, so you can supervise each independently, or not at all. This allows for a lot of flexibility. You don't have to do it this way, but it really helps.

**You don't need to do any of this** - we have a default DB setup for you. However, if you want a formalized, supervised module for your database, here's how you do it.

First, create a module for your database:

```
1 defmodule MyApp.Db do  
2   use Moebius.Database  
3  
4   # helper/repo methods go here  
5 end
```

---

Next, in your `Application` file, add this new module to your supervision tree:

```
1 def start(_type, _args) do
2   start_db
3   #...
4 end
5
6 def start_db do
7   #create a child process
8   children = [
9     {MyApp.Db, [Moebius.get_connection]}
10  ]
11  Supervisor.start_link children, strategy: :one_for_one
12 end
```

That's it. Now, when your app starts you'll have a supervised database you can use as needed. The function `Moebius.get_connection/0` will look for a key called `:connection` in your `config.exs`. If you want to connect to multiple databases, name these connections something meaningful, then pass that to `Moebius.get_connection/1`.

For instance, you might have a sales database and an accounting one; or you might have a read-only connection and a write-only one to spread the load. For this, just specify each as needed:

```
1 config :moebius, read_only: [
2   url: "postgres://user:password@host/database"
3 ],
4 write_only: [
5   url: "postgres://user:password@host/database"
6 ],
7 scripts: "test/db"
```

You can now use these in your database module:

```
1 def start(_type, _args) do
2   start_db
3   #...
4 end
5
6 def start_db do
7   #create a worker
8   read_only_db_worker = worker(MyApp.Db, [Moebius.get_connection(:
9     read_only)])
10  write_only_db_worker = worker(MyApp.Db, [Moebius.get_connection(:
11    write_only)])
12  Supervisor.start_link [read_only_db_worker, write_only_db_worker],
13    strategy: :one_for_one
14 end
```

It bears repeating: *you don't need to do any of this*, we have a default database setup for you. However

---

supporting multiple connections was very high on our list so this is how we chose to do it (with many thanks to Peter Hamilton for the idea).

The rest of the examples you see below use our default database.

## The Basic Query Flow

When querying the database (read or write), you construct the query and then pass it to the database you want:

```
1 { :ok, result } = Moebius.Query.db(:users) |> Moebius.Db.first
```

In this example, `db(:users)` initiates the `QueryCommand`, we can filter it, sort it, do all kinds of things. To run it, however, we need to pass it to the database we want to execute against.

The default database is `Moebius.Db`, but you can make your own with a dedicated connection as needed (see above).

Let's see some more examples.

## Simple Examples

The API is built around the concept of transforming raw data from your database into something you need, and we try to make it feel as *functional* as possible. We lean on Elixir's `|>` operator for this, and it's the core of the API.

This returns a user with the id of 1.

```
1 { :ok, result } =  
2   db(:users)  
3   |> filter(name: "Steve")  
4   |> sort(:city, :desc)  
5   |> limit(10)  
6   |> offset(2)  
7   |> Moebius.Db.run
```

Hopefully it's fairly straightforward what this query returns. All users named Steve sorted by city... skipping the first two, returning the next 10.

## Operators

An “=” (Equal) query happens when you pass a column name and a value:

---

```
1 {:ok, result} =
2   db(:users)
3   |> filter(name: "mark")
4   |> Moebius.Db.run
5
6 # or, if you want to be more precise, specify the `eq` key:
7
8 {:ok, result} =
9   db(:users)
10  |> filter(:name, eq: "mark"])
11  |> Moebius.Db.run
```

A “!=” (Not Equal) query happens when you specify the `neq` key:

```
1 {:ok, result} =
2   db(:users)
3   |> filter(:name, neq: "mark")
4   |> Moebius.Db.run
```

A “>” (Greater Than) query happens when you specify the `gt` key:

```
1 {:ok, result} =
2   db(:users)
3   |> filter(:order_count, gt: 5)
4   |> Moebius.Db.run
```

Additionally, the following comparison operators are available:

- “<” (Less Than): `lt`
- “>=” (Greater Than or Equal To): `gte`
- “<=” (Less Than or Equal To) `lte`

An “IN” query happens when you pass an array:

```
1 {:ok, result} =
2   db(:users)
3   |> filter(:name, ["mark", "biff", "skip"])
4   |> Moebius.Db.run
5
6 # or, if you want to be more precise, specify the `in` key:
7
8 {:ok, result} =
9   db(:users)
10  |> filter(:name, in: ["mark", "biff", "skip"])
11  |> Moebius.Db.run
```

A “NOT IN” query happens when you specify the `not_in` or `nin` key:

```
1 {:ok, result} =
```

---

```
2 db(:users)
3 |> filter(:name, not_in: ["mark", "biff", "skip"])
4 |> Moebius.Db.run
```

If you prefer a more SQL-like syntax, you can use the following aliases:

- db: `from`
- filter: `where`
- sort: `order_by`

```
1 {:ok, result} =
2   from(:users)
3   |> where(name: "Steve")
4   |> where(:order_count, gt: 5)
5   |> order_by(id: :asc, name: :desc)
```

If you don't want to deal with my abstractions, just use SQL:

```
1 {:ok, result} = "select * from users where id=1 limit 1 offset 1;" |>
  Moebius.Db.run
```

## Full Text indexing

One of the great features of PostgreSQL is the ability to do intelligent full text searches. We support this functionality directly:

```
1 {:ok, result} =
2   db(:users)
3   |> search(for: "Mike", in: [:first, :last, :email])
4   |> Moebius.Db.run
```

The `search` function builds a `tsvector` search on the fly for you and executes it over the columns you send in. The results are ordered in descending order using `ts_rank`.

## JSONB Support

Moebius supports using PostgreSQL as a document store in its entirety. Get your project off the ground and don't worry about migrations - just store documents, and you can normalize if you need to later on.

Start by importing `Moebius.DocumentQuery` and saving a document:

```
1 import Moebius.DocumentQuery
2
3 {:ok, new_user} =
```

---

```
4 db(:friends)
5 |> Moebius.Db.save(email: "test@test.com", name: "Moe Test")
```

Two things happened for us here. The first is that `friends` did not exist as a document table in our database, but `save/2` did that for us. This is the table that was created on the fly:

```
1 create table NAME(
2   id serial primary key not null,
3   body jsonb not null,
4   search tsvector,
5   created_at timestamptz not null default now(),
6   updated_at timestamptz not null default now()
7 );
8
9 -- index the search and jsonb fields
10 create index idx_NAME_search on NAME using GIN(search);
11 create index idx_NAME on NAME using GIN(body jsonb_path_ops);
```

The entire `DocumentQuery` module works off the premise that this is how you will store your JSONB docs. Note the `tsvector` field? That's PostgreSQL's built in full text indexing. We can use that if we want during by adding `searchable/1` to the pipe:

```
1 import Moebius.DocumentQuery
2
3 {:ok, new_user} =
4   db(:friends)
5   |> searchable([:name])
6   |> Moebius.Db.save(email: "test@test.com", name: "Moe Test")
```

By specifying the searchable fields, the `search` field will be updated with the values of the name field.

Now, we can query our document using full text indexing which is optimized to use the GIN index created above:

```
1 {:ok, user} =
2   db(:friends)
3   |> search("test.com")
4   |> Moebius.Db.run
```

Or we can do a simple filter:

```
1 {:ok, user} =
2   db(:friends)
3   |> contains(email: "test@test.com")
4   |> Moebius.Db.run
```

This query is optimized to use the `@` (or “contains” operator), using the *other* GIN index specified above. There's more we can do...

---

```
1 { :ok, users } =  
2   db(:friends)  
3   |> filter(:money_spent, ">", 100)  
4   |> Moebius.Db.run
```

This runs a full table scan so is not terribly optimal, but it does work if you need it once in a while. You can also use the existence (?) operator, which is very handy for querying arrays. In the library, it is implemented as `exists`:

```
1 { :ok, buddies } =  
2   db(:friends)  
3   |> exists(:tags, "best")  
4   |> Moebius.Db.run
```

This will allow you to query embedded documents and arrays rather easily, but again doesn't use the JSONB-optimized GIN index. You *can* index for using existence, have a look at the PostgreSQL docs.

## Using Structs

If you're a big fan of structs, you can use them directly on `save` and we'll send that same struct back to you, complete with an `id`:

```
1 defmodule Candy do  
2   defstruct [  
3     id: nil,  
4     sticky: true,  
5     chocolate: "gooey"  
6   ]  
7 end  
8  
9 yummy = %Candy{}  
10 { :ok, res } = db(:monkies) |> Moebius.Db.save(yummy)  
11 #res = %Candy{id: 1, sticky: true, chocolate: "gooey"}
```

I've been using this functionality constantly with another project I'm working on and it's helped me tremendously.

## SQL Files

I built this for MassiveJS and I liked the idea, which is this: *some people love SQL*. I'm one of those people. I'd much rather work with a SQL file than muscle through some weird abstraction.

With this library you can do that. Just create a scripts directory and specify it in the config (see above), then execute your file without an extension. Pass in whatever parameters you need:



---

```
1 {:ok, result} = sql_file(:my_groovy_query, "a param") |> Moebius.Db.run
```

I highly recommend this approach if you have some difficult SQL you want to write (like a windowing query or CTE). We use this approach to build our test database - have a look at our tests and see.

## Adding, Updating, Deleting (Non-Documents)

Inserting is pretty straightforward:

```
1 {:ok, result} =
2   db(:users)
3   |> insert(email: "test@test.com", first: "Test", last: "User")
4   |> Moebius.Db.run
```

Updating can work over multiple rows, or just one, depending on the filter you use:

```
1 {:ok, result} =
2   db(:users)
3   |> filter(id: 1)
4   |> update(email: "maggot@test.com")
5   |> Moebius.Db.run
```

The filter can be a single record, or affect multiple records:

```
1 {:ok, result} =
2   db(:users)
3   |> filter("id > 100")
4   |> update(email: "test@test.com")
5   |> Moebius.Db.run
6
7 {:ok, result} =
8   db(:users)
9   |> filter("email LIKE $2", "%test")
10  |> update(email: "ox@test.com")
11  |> Moebius.Db.run
```

Deleting works exactly the same way as `update`, but returns the count of deleted items in the result:

```
1 {:ok, result} =
2   db(:users)
3   |> filter("email LIKE $2", "%test")
4   |> delete
5   |> Moebius.Db.run
6
7 #result.deleted = 10, for instance
```

---

## Bulk Inserts

Moebius supports bulk insert operations transactionally. We've fine-tuned this capability quite a lot (thanks to Jon Atten) and, on our local machines, have achieved ~60,000 writes per second. This, of course, will vary by machine, configuration, and use.

But that's still a pretty good number don't you think?

A bulk insert works by invoking one directly:

```
1 data = [#let's say 10,000 records or so]
2 {:ok, result} =
3   db(:people)
4   |> bulk_insert(data)
5   |> Moebius.Db.transact_batch
```

If everything works, you'll get back a result indicating the number of records inserted.

## Table Joins

Table joins can be applied for a single join or piped to create multiple joins. The table names can be either atoms or binary strings. There are a number of options to customize your joins:

```
1   :join          # set the type of join. LEFT, RIGHT, FULL, etc. defaults
   to INNER
2   :on            # specify the table to join on
3   :foreign_key   # specify the tables foreign key column
4   :primary_key   # specify the joining tables primary key column
5   :using         # used to specify a USING queries list of columns to
   join on
```

The simplest example is a basic join:

```
1 {:ok, result} =
2   db(:customer)
3   |> join(:order)
4   |> select
5   |> Moebius.Db.run
```

For multiple table joins you can specify the table that you want to join on:

```
1 {:ok, result} =
2   db(:customer)
3   |> join(:order, on: :customer)
4   |> join(:item, on: :order)
5   |> select
6   |> Moebius.Db.run
```

---

## Transactions

Transactions are facilitated by using a callback that has a `pid` on it, which you'll need to pass along to each query you want to be part of the transaction. The last execution will be returned. If there's an error, an `{:error, message}` will be returned instead and a `ROLLBACK` fired on the transaction. No need to `COMMIT`, it happens automatically:

```
1 {:ok, result} = transaction fn(pid) ->
2   new_user =
3     db(:users)
4     |> insert(pid, email: "frodo@test.com")
5     |> Moebius.Db.run(pid)
6
7   with(:logs)
8     |> insert(pid, user_id: new_user.id, log: "Hi Frodo")
9     |> Moebius.Db.run(pid)
10  new_user
11 end
```

If you're having any kind of trouble with transactions, I highly recommend you move to a SQL file or a function, which we also support. Abstractions are here to help you, but if we're in your way, by all means shove us (gently) aside.

## Aggregates

Aggregates are built with a functional approach in mind. This might seem a bit odd, but when working with any relational database, it's a good idea to think about gathering your data, grouping it, and reducing it. That's what you're doing whenever you run aggregation queries.

So, to that end, we have:

```
1 {:ok, sum} =
2   db(:products)
3   |> map("id > 1")
4   |> group(:sku)
5   |> reduce(:sum, :id)
6   |> Moebius.Db.run
```

This might be a bit verbose, but it's also very very clear to whomever is reading it after you move on. You can work with any aggregate function in PostgreSQL this way (AVG, MIN, MAX, etc).

The interface is designed with *routine* aggregation in mind - meaning that there are some pretty complex things you can do with PostgreSQL queries. If you like doing that, I fully suggest you flex our SQL File functionality and write it out there - or create yourself a cool function and call it with our Function interface.

---

## Functions

PostgreSQL allows you to do so much, especially with functions. If you want to encapsulate a good time, you can execute it with Moebius:

```
1 {:ok, party} = function(:good_time, [me, you]) |> Moebius.Db.run
```

You get the idea. If your function only returns one thing, you can specify you don't want an array back:

```
1 {:ok, no_party} = function(:bad_time, :single [me]) |> Moebius.Db.run
```

## Test

You'll need a local postgres instance running.

```
1 MIX_ENV=test mix moebius.setup
2 MIX_ENV=test mix test
```

## Help?

I would love to have your help! I do ask that if you do find a bug, please add a test to your PR that shows the bug and how it was fixed.

Thanks!