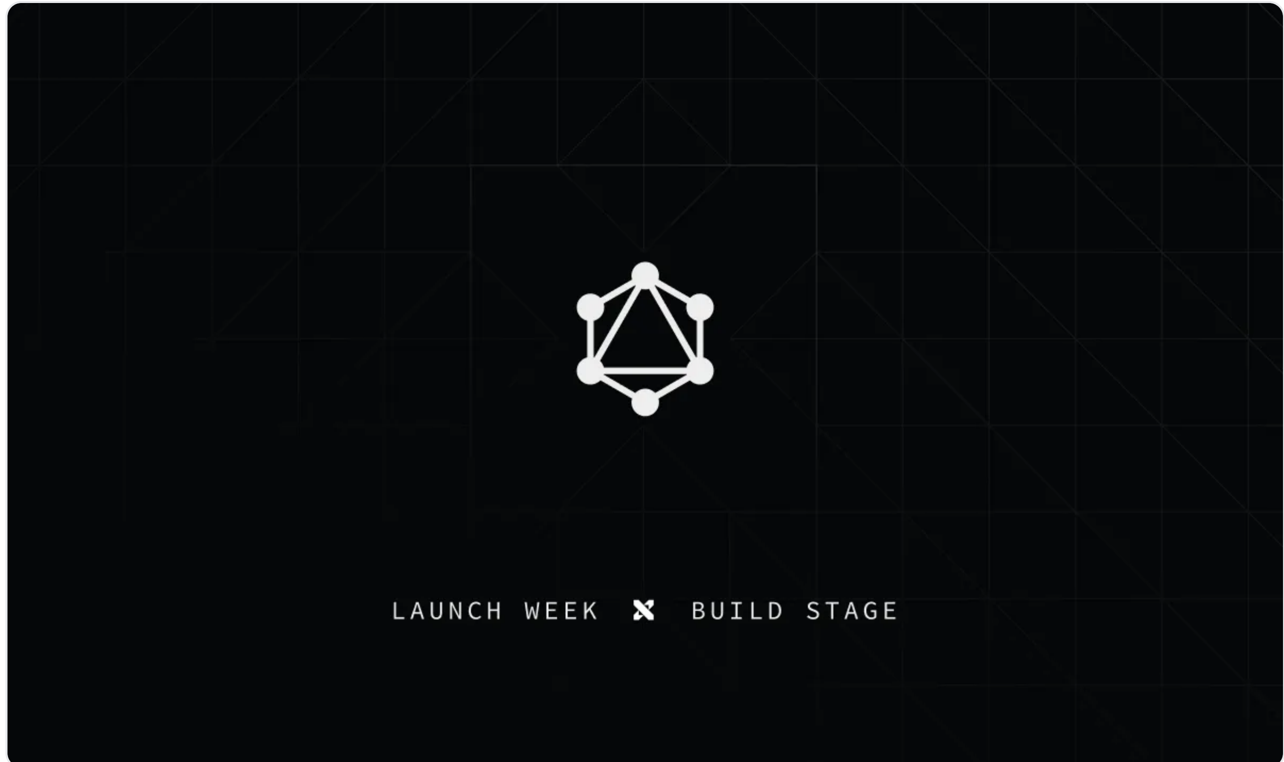
pg_graphql: Postgres functions now supported

2023-12-12 • 7 minute read



Supabase GraphQL (pg_graphql) 1.4+ supports the most requested feature: Postgres functions a.k.a. User Defined Functions (UDFs). This addition marks a significant improvement in GraphQL flexibility at Supabase, both as a novel approach to defining entry points into the Graph and as an escape hatch for users to implement custom/complex operations.

As with all entities in Supabase GraphQL, UDFs support is based on automatically reflecting parts of the SQL schema. The feature allow for the execution of custom SQL logic within GraphQL queries to help support complex, user defined, server-side operations with a simple GraphQL interface.

Minimal Example

Consider a functio

We only collect analytics essential to ensuring smooth operation of our services.

Accept Opt out Learn more

```
1 create function "addNums"(a int, b int default 1)
```

```
create function "addNums"(a int, b int default 1)
returns int
immutable
language sql
as $$
    select a + b;
$$;
```

when reflected in the GraphQL schema, the function is exposed as:

```
type Query {
  addNums(a: Int!, b: Int): Int
}
```

To use this entry point, you could run:

```
query {
  addNums(a: 2, b: 3)
}
```

which returns the JSON payload:

```
{
  "data": {
    "addNums": 5
  }
}
```

Supabase GraphQL does its best to reflect a coherent GraphQL API from all the information known to the SQL layer. For example, the argument `a` is non-null because it doesn't have a default value while `b` can be omitted since it does have a default. We also detected that this UDF can be displayed in the `Query` type rather than the `Mutation` type because the function was declared as `immutable`, which means it can not edit the database. Of the other function volatility categories, `stable` similarly translates into a `Query` field while `volatile` (the default) becomes a `Mutation` field.

Returning Reco

We only collect analytics essential to ensuring smooth operation of our services.

Learn more

In a more realistic example, we might want to return a set of an existing object type like `Account`. For example, lets say we want to search for accounts based on their email address domains matching a string:

```
create table "Account"(
  id serial primary key,
  email varchar(255) not null
);

insert into "Account"(email)
values
  ('a@foo.com'),
  ('b@bar.com'),
  ('c@foo.com');

create function "accountsByEmailDomain"("domainToSearch" te
  returns setof "Account"
  stable
  language sql
as $$
    select
        id, email
    from
        "Account"
    where
        email ilike ('%@' || "domainToSearch");
$$;
```

Since our function is `stable`, it continues to be a field on the `Query` type. Notice that since we're returning a collection of `Account` we automatically get support for Relay style pagination on the response including `first`, `last`, `before`, `after` as well as filtering and sorting.

```
type Query {
  accountsByEmailDomain(
    domainToSearch: String!

    """
    Query the first `n` records in the collection
    """
    first: Int

    """
    Query th
    """
    last: In

    """
    Query values in the collection before the provided curs
```

We only collect analytics essential to ensuring smooth operation of our services.

Learn more

```
17     """
18     before: Cursor
19
20     """
21     Query values in the collection after the provided curs(
22     """
23     after: Cursor
24
25     """
26     Filters to apply to the results set when querying from
27     """
28     filter: AccountFilter
29
30     """
31     Sort order to apply to the collection
32     """
33     orderBy: [AccountOrderBy!]
34   ): AccountConnection
35 }
```

To complete the example, here's a call to our user defined function:

```
1  query {
2    accountsByEmailDomain(domainToSearch: "foo.com", first: 2
3      edges {
4        node {
5          id
6          email
7        }
8      }
9    }
10 }
```

and the response:

```
1  {
2    "data": {
3      "accountsByEmail": {
4        "edges": [
5          {
6            "node": {
7              "id": 1,
8              "email": "a@foo.com"
9            }
10         },
11         "n
12
13
14         }
15       ]
```

We only collect analytics essential to ensuring smooth operation of our services.

Learn more

```
16      }
17    }
18  }
```

While not shown here, any relationships defined by foreign keys on the response type `Account` are fully functional so our UDF result is completely connected to the existing Graph.

It's worth mentioning that we could have supported this query using the default `accountCollection` field that `pg_graphql` exposes on the `Query` type using an `ilike` filter so the example is only for illustrative purposes.

i.e.:

```
1  query {
2    accountCollection(filter: { email: { ilike: "%foo.com" } }) {
3      edges {
4        node {
5          id
6          email
7        }
8      }
9    }
10 }
```

would give the same result as our UDF.

Limitations

The API surface area of SQL functions is surprisingly large. In an effort to bring this feature out sooner, some lesser-used parts have not been implemented yet. Currently functions using the following features are excluded from the GraphQL API:

- Overloaded functions
- Functions with a nameless argument
- Functions returning void
- Variadic functio
- Functions that a
- Functions that a

We only collect analytics essential to ensuring smooth operation of our services.

Learn more

We look forward to implementing support for many of these features in coming releases.

Takeaways

If you're an existing Supabase user, but new to GraphQL, head over to GraphiQL built right into Supabase Studio for your project to interactively explore your projects through the GraphQL API. User defined function support is new in pg_graphql 1.4+. You can check your project's GraphQL version with:

```
1 select *
2 from pg_available_extensions
3 where name = 'pg_graphql';
```

To upgrade, check out our upgrade guide.

For new Supabase users, creating a new project will get you the latest version of Supabase GraphQL with UDF support.

If you're not ready to start a new project but want to learn more about `pg_graphql`/Supabase GraphQL, our API docs are a great place to learn about how your SQL schema is transformed into a GraphQL API.

More Launch Week X

- Day 1 - Supabase Studio update: AI Assistant and User Impersonation
- Postgres Language Server: implementing the Parser
- How design works at Supabase
- The Supabase Album
- Launch Week X Hackathon
- Launch Week X Community Meetups

Share this article

We only collect analytics essential to ensuring smooth operation of our services.

Learn more

Last post

Edge Functions: Node and native npm compatibility

12 December 2023

Next post

Supabase Studio: AI Assistant and User Impersonation

11 December 2023

Related articles

Edge Functions: Node and native npm compatibility
Supabase Studio: AI Assistant and User Impersonation
Postgres Language Server: implementing the Parser
How design works at Supabase
Supabase Launch Week X Hackathon

View all posts

Build in a weekend, scale to millions

Start your project

We only collect analytics essential to ensuring smooth operation of our services.

Learn more