Database Access and security Row Level Security

Row Level Security

Secure your data using Postgres Row Level Security.

When you need granular authorization rules, nothing beats Postgres's .

Row Level Security in Supabase

RLS is incredibly powerful and flexible, allowing you to write complex SQL rules that fit your unique business needs. RLS can be combined with for end-to-end user security from the browser to the database.

RLS is a Postgres primitive and can provide " " to protect your data from malicious actors even when accessed through third-party tooling.

Policies

are Postgres's rule engine. Policies are easy to understand once you get the hang of them. Each policy is attached to a table, and the policy is executed every time a table is accessed.

You can just think of them as adding a WHERE clause to every query. For example a policy like this ...

.. would translate to this whenever a user tries to select from the todos table:

Enabling Row Level Security

You can enable RLS for any table using the enable row level security clause:

Row Level Security (RLS)

Supabase allows convenient and secure data access from the browser, as long as you enable RLS.

RLS must always be enabled on any tables stored in an exposed schema. By default, this is the public schema. RLS is enabled by default on tables created with the Table Editor in the dashboard. If you create one in raw SQL or with the SQL editor, remember to enable RLS yourself:

12 alter table <schema\_name>.<table\_name> enable row level security;

Supabase Auth

defense in depth

Policies

123

create policy "Individuals can view their own todos." on todos for select

using ( (select auth.uid()) = user\_id );

1234

select \* from todos

where auth.uid() = todos.user\_id;

*-- Policy is implicitly added.*

Is this helpful?

ON THIS PAGE

Row Level Security in Supabase

Policies

Enabling Row Level Security

Authenticated and unauthenticated roles

Creating policies SELECT policies INSERT policies UPDATE policies DELETE policies Views Helper functions auth.uid() auth.jwt() MFA Bypassing Row Level Security

RLS performance recommendations

Add indexes

Call functions with select

Add filters to every query

Use security definer functions

Minimize joins

Specify roles in your policies

More resources

Database

DOCS Search K

Once you have enabled RLS, no data will be accessible via the when using the public anon key, until you create policies.

Authenticated and unauthenticated roles

Supabase maps every request to one of the roles:

These are actually . You can use these roles within your Policies using the TO clause:

Creating policies

Policies are SQL logic that you attach to a Postgres table. You can attach as many policies as you want to each table.

Supabase provides some that simplify RLS if you're using Supabase Auth. We'll use these helpers to illustrate some basic policies:

SELECT policies

You can specify select policies with the using clause.

Let's say you have a table called profiles in the public schema and you want to enable read access to everyone.

1 alter table "table\_name" enable row level security;

API

anon : an unauthenticated request (the user is not logged in)

authenticated : an authenticated request (the user is logged in)

Postgres Roles

123456789

10

11

create policy "Profiles are viewable by everyone" on profiles for select

to authenticated, anon using ( true );

*-- OR*

create policy "Public profiles are viewable only by authenticated users"

on profiles for select to authenticated using ( true );

Anonymous user vs the anon key

Using the anon Postgres role is different from an in Supabase Auth. An anonymous user assumes the authenticated role to access the database and can be differentiated from a permanent user by checking the is\_anonymous claim in the JWT. anonymous user

helpers

123456789

10

11

12

*-- 1. Create table*

create table profiles ( id uuid primary key,

user\_id references auth.users,

avatar\_url text );

*-- 2. Enable RLS*

alter table profiles enable row level security;

*-- 3. Create Policy*

create policy "Public profiles are visible to everyone."

Alternatively, if you only wanted users to be able to see their own profiles:

INSERT policies

You can specify insert policies with the with check clause. The with check expression ensures that any new row data adheres to the policy constraints.

Let's say you have a table called profiles in the public schema and you only want users to be able to create a profile for themselves. In that case, we want to check their User ID matches the value that they are trying to insert:

UPDATE policies

You can specify update policies by combining both the using and with check expressions.

The using clause represents the condition that must be true for the update to be allowed, and with check clause ensures that the updates made adhere to the policy constraints.

Let's say you have a table called profiles in the public schema and you only want users to be able to update their own profile.

You can create a policy where the using clause checks if the user owns the profile being updated. And the with check clause ensures that, in the resultant row, users do not change the user\_id to a value that is not equal to their User ID, maintaining that the modified profile still meets the ownership condition.

13

14

15

on profiles for select

to anon *-- the Postgres Role (recommended)*

using ( true ); *-- the actual Policy*

123

create policy "User can see their own profile only." on profiles

for select using ( (select auth.uid()) = user\_id );

123456789

10

11

12

13

14

15

*-- 1. Create table*

create table profiles ( id uuid primary key,

user\_id uuid references auth.users,

avatar\_url text );

*-- 2. Enable RLS*

alter table profiles enable row level security;

*-- 3. Create Policy*

create policy "Users can create a profile." on profiles for insert

|  |  |
| --- | --- |
| to authenticated | *-- the Postgres Role (recommended)* |
| with check ( (select auth.uid()) = user\_id ); | *-- the actual Policy* |

123456789

10

11

12

*-- 1. Create table*

create table profiles ( id uuid primary key,

user\_id uuid references auth.users,

avatar\_url text );

*-- 2. Enable RLS*

alter table profiles enable row level security;

*-- 3. Create Policy*

create policy "Users can update their own profile."

If no with check expression is defined, then the using expression will be used both to determine which rows are visible (normal USING case) and which new rows will be allowed to be added (WITH CHECK case).

DELETE policies

You can specify delete policies with the using clause.

Let's say you have a table called profiles in the public schema and you only want users to be able to delete their own profile:

Views

Views bypass RLS by default because they are usually created with the postgres user. This is a feature of Postgres, which automatically creates views with security definer .

In Postgres 15 and above, you can make a view obey the RLS policies of the underlying tables when invoked by anon and authenticated roles by setting security\_invoker = true .

In older versions of Postgres, protect your views by revoking access from the anon and authenticated roles, or by putting them in an unexposed schema.

Helper functions

Supabase provides some helper functions that make it easier to write Policies.

auth.uid()

Returns the ID of the user making the request.

13

14

15

16

on profiles for update

|  |  |
| --- | --- |
| to authenticated | *-- the Postgres Role (recommended)* |
| using ( (select auth.uid()) = user\_id ) | *-- checks if the existing row complies with the polic* |

with check ( (select auth.uid()) = user\_id ); *-- checks if the new row complies with the policy exp*

To perform an UPDATE operation, a corresponding is required. Without a SELECT policy, the UPDATE operation will not work as expected. SELECT policy

123456789

10

11

12

13

14

15

*-- 1. Create table*

create table profiles ( id uuid primary key,

user\_id uuid references auth.users,

avatar\_url text );

*-- 2. Enable RLS*

alter table profiles enable row level security;

*-- 3. Create Policy*

create policy "Users can delete a profile." on profiles for delete

|  |  |
| --- | --- |
| to authenticated | *-- the Postgres Role (recommended)* |
| using ( (select auth.uid()) = user\_id ); | *-- the actual Policy* |

123

create view <VIEW\_NAME> with(security\_invoker = true) as select <QUERY>

auth.jwt()

Returns the JWT of the user making the request. Anything that you store in the user's raw\_app\_meta\_data column or the raw\_user\_meta\_data column will be accessible using this function. It's important to know the distinction between these two:

The auth.jwt() function is extremely versatile. For example, if you store some team data inside app\_metadata , you can use it to determine whether a particular user belongs to a team. For example, if this was an array of IDs:

MFA

The auth.jwt() function can be used to check for . For example, you could restrict a user from updating their profile unless they have at least 2 levels of authentication (Assurance Level 2):

Bypassing Row Level Security

Supabase provides special "Service" keys, which can be used to bypass RLS. These should never be used in the browser or exposed to customers, but they are useful for administrative tasks.

You can also create new which can bypass Row Level Security using the "bypass RLS" privilege:

This can be useful for system-level access. You should never share login credentials for any Postgres Role with this privilege.

Not all information present in the JWT should be used in RLS policies. For instance, creating an RLS policy that relies on the user\_metadata claim can create security issues in your application as this information can be modified by authenticated end users.

raw\_user\_meta\_data - can be updated by the authenticated user using the supabase.auth.update() function.

It is not a good place to store authorization data.

raw\_app\_meta\_data - cannot be updated by the user, so it's a good place to store authorization data.

1234

create policy "User is in team"

on my\_table

to authenticated

using ( team\_id in (select auth.jwt() -> 'app\_metadata' -> 'teams'));

Keep in mind that a JWT is not always "fresh". In the example above, even if you remove a user from a team and update the app\_metadata field, that will not be reflected using auth.jwt() until the user's JWT is refreshed. Also, if you are using Cookies for Auth, then you must be mindful of the JWT size. Some browsers are limited to 4096 bytes for each cookie, and so the total size of your JWT should be small enough to fit inside this limitation.

Multi-Factor Authentication

1234567

create policy "Restrict updates."

on profiles as restrictive for update to authenticated using (

(select auth.jwt()->>'aal') = 'aal2' );

Supabase will adhere to the RLS policy of the signed-in user, even if the client library is initialized with a Service Key.

Postgres Roles

1 alter role "role\_name" with bypassrls;

RLS performance recommendations

Every authorization system has an impact on performance. While row level security is powerful, the performance impact is important to keep in mind. This is especially true for queries that scan every row in a table - like many select operations, including those using limit, offset, and ordering.

Based on a series of , we have a few recommendations for RLS:

Add indexes

Make sure you've added on any columns used within the Policies which are not already indexed (or primary keys). For a Policy like this:

You can add an index like:

Benchmarks

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test | Before (ms) | After (ms) | % Improvement | Change |
| 171 | < 0.1 | 99.94% | Details | test1-indexed |

Call functions with select

You can use select statement to improve policies that use functions. For example, instead of this:

You can do:

This method works well for JWT functions like auth.uid() and auth.jwt() as well as security definer Functions. Wrapping the function causes an initPlan to be run by the Postgres optimizer, which allows it to "cache" the results per-statement, rather than calling the function on each row.

Benchmarks

tests

indexes

123

create policy "rls\_test\_select" on test\_table to authenticated

using ( (select auth.uid()) = user\_id );

123

create index userid on test\_table

using btree (user\_id);

123

create policy "rls\_test\_select" on test\_table to authenticated

using ( auth.uid() = user\_id );

123

create policy "rls\_test\_select" on test\_table to authenticated

using ( (select auth.uid()) = user\_id );

You can only use this technique if the results of the query or function do not change based on the row data.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test | Before (ms) | After (ms) | % Improvement | Change |
| 179 | 9 | 94.97% | Details | test2a-wrappedSQL-uid |
| 11,000 | 7 | 99.94% | Details | test2b-wrappedSQL-isadmin |
| 11,000 | 10 | 99.91% | Details | test2c-wrappedSQL-two-functions |
| 178,000 | 12 | 99.993% | Details | test2d-wrappedSQL-sd-fun |
| 173000 | 16 | 99.991% | Details | test2e-wrappedSQL-sd-fun-array |

Add filters to every query

Policies are "implicit where clauses," so it's common to run select statements without any filters. This is a bad pattern for performance. Instead of doing this (JS client example):

You should always add a filter:

Even though this duplicates the contents of the Policy, Postgres can use the filter to construct a better query plan.

Benchmarks

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test | Before (ms) | After (ms) | % Improvement | Change |
| 171 | 9 | 94.74% | Details | test3-addfilter |

Use security definer functions

A "security definer" function runs using the same role that created the function. This means that if you create a role with a superuser (like postgres ), then that function will have bypassrls privileges. For example, if you had a policy like this:

We can instead create a security definer function which can scan roles\_table without any RLS penalties:

123

const { data } = supabase .from('table') .select()

1234

const { data } = supabase .from('table') .select() .eq('user\_id', userId)

12345678

create policy "rls\_test\_select" on test\_table to authenticated

using ( exists (

select 1 from roles\_table

where (select auth.uid()) = user\_id and role = 'good\_role' )

);

12345

create function private.has\_good\_role() returns boolean

language plpgsql

security definer *-- will run as the creator* as $$

Minimize joins

You can often rewrite your Policies to avoid joins between the source and the target table. Instead, try to organize your policy to fetch all the relevant data from the target table into an array or set, then you can use an IN or ANY operation in your filter.

For example, this is an example of a slow policy which joins the source test\_table to the target team\_user :

We can rewrite this to avoid this join, and instead select the filter criteria into a set:

In this case you can also consider to bypass RLS on the join table:

Benchmarks

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test | Before (ms) | After (ms) | % Improvement | Change |
| 9,000 | 20 | 99.78% | Details | test5-fixed-join |

Specify roles in your policies

6789

10

11

12

13

14

15

16

17

18

begin

return exists (

select 1 from roles\_table

where (select auth.uid()) = user\_id and role = 'good\_role' );

end; $$;

*-- Update our policy to use this function:*

create policy "rls\_test\_select" on test\_table

to authenticated

using ( private.has\_good\_role() );

Security-definer functions should never be created in a schema in the "Exposed schemas" inside your API settings`.

123456789

create policy "rls\_test\_select" on test\_table to authenticated

using (

(select auth.uid()) in ( select user\_id

from team\_user

where team\_user.team\_id = team\_id *-- joins to the source "test\_table.team\_id"*

) );

123456789

create policy "rls\_test\_select" on test\_table to authenticated

using (

team\_id in ( select team\_id from team\_user

where user\_id = (select auth.uid()) *-- no join* )

);

using a security definer function

If the list exceeds 1000 items, a different approach may be needed or you may need to analyze the approach to ensure that the performance is acceptable.

Need some help? Contact support

Latest product updates? See Changelog Something's not right? Check system status

© Supabase Inc — Co ntributing Author Styleguide Open Source SupaSquad

Privacy Settings

Always use the Role of inside your policies, specified by the TO operator. For example, instead of this query:

Use:

This prevents the policy ( (select auth.uid()) = user\_id ) from running for any anon users, since the execution stops at the to authenticated step.

Benchmarks

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test | Before (ms) | After (ms) | % Improvement | Change |
| 170 | < 0.1 | 99.78% | Details | test6-To-role |

More resources

Edit this page on GitHub

12 create policy "rls\_test\_select" on rls\_test using ( auth.uid() = user\_id );

123

create policy "rls\_test\_select" on rls\_test to authenticated

using ( (select auth.uid()) = user\_id );

Testing your database Row Level Security and Supabase Auth RLS Guide and Best Practices

Community repo on testing RLS using pgTAP and dbdev