

[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 \(RLS\)](#).

## Row Level Security in Supabase



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:

```
1 alter table <schema_name>.<table_name>
2 enable row level security;
```

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

RLS is a Postgres primitive and can provide "[defense in depth](#)" to protect your data from malicious actors even when accessed through third-party tooling.

## Policies

[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 ...

```
1 create policy "Individuals can view their own todos."
2 on todos for select
3 using ( (select auth.uid()) = user_id );
```

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

```
1 select *
2 from todos
3 where auth.uid() = todos.user_id;
4 -- Policy is implicitly added.
```

## Enabling Row Level Security

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

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](#)

```
1 alter table "table_name" enable row level security;
```

Once you have enabled RLS, no data will be accessible via the [API](#) when using the public `anon` key, until you create policies.

## Authenticated and unauthenticated roles

Supabase maps every request to one of the roles:

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

These are actually [Postgres Roles](#). You can use these roles within your Policies using the `TO` clause:

```
1 create policy "Profiles are viewable by everyone"
2 on profiles for select
3 to authenticated, anon
4 using ( true );
5
6 -- OR
7
8 create policy "Public profiles are viewable only by authenticated users"
9 on profiles for select
10 to authenticated
11 using ( true );
```



### Anonymous user vs the anon key

Using the `anon` Postgres role is different from an [anonymous user](#) 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.

## 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 [helpers](#) 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 -- 1. Create table
2 create table profiles (
3   id uuid primary key,
4   user_id references auth.users,
5   avatar_url text
6 );
7
8 -- 2. Enable RLS
9 alter table profiles enable row level security;
10
11 -- 3. Create Policy
12 create policy "Public profiles are visible to everyone."
```

```

13 on profiles for select
14 to anon          -- the Postgres Role (recommended)
15 using ( true ); -- the actual Policy

```

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

```

1 create policy "User can see their own profile only."
2 on profiles
3 for select using ( (select auth.uid()) = user_id );

```

## 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:

```

1 -- 1. Create table
2 create table profiles (
3   id uuid primary key,
4   user_id uuid references auth.users,
5   avatar_url text
6 );
7
8 -- 2. Enable RLS
9 alter table profiles enable row level security;
10
11 -- 3. Create Policy
12 create policy "Users can create a profile."
13 on profiles for insert
14 to authenticated          -- the Postgres Role (recommended)
15 with check ( (select auth.uid()) = user_id ); -- the actual Policy

```

## 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.

```

1 -- 1. Create table
2 create table profiles (
3   id uuid primary key,
4   user_id uuid references auth.users,
5   avatar_url text
6 );
7
8 -- 2. Enable RLS
9 alter table profiles enable row level security;
10
11 -- 3. Create Policy
12 create policy "Users can update their own profile."

```

```

13 on profiles for update
14 to authenticated -- the Postgres Role (recommended)
15 using ( (select auth.uid()) = user_id ) -- checks if the existing row complies with the policy
16 with check ( (select auth.uid()) = user_id ); -- checks if the new row complies with the policy expression

```

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).



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

## 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:

```

1 -- 1. Create table
2 create table profiles (
3   id uuid primary key,
4   user_id uuid references auth.users,
5   avatar_url text
6 );
7
8 -- 2. Enable RLS
9 alter table profiles enable row level security;
10
11 -- 3. Create Policy
12 create policy "Users can delete a profile."
13 on profiles for delete
14 to authenticated -- the Postgres Role (recommended)
15 using ( (select auth.uid()) = user_id ); -- the actual Policy

```

## 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`.

```

1 create view <VIEW_NAME>
2 with(security_invoker = true)
3 as select <QUERY>

```

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.

`auth.jwt()`



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.

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:

- `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.

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:

```
1 create policy "User is in team"
2 on my_table
3 to authenticated
4 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.

## MFA

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

```
1 create policy "Restrict updates."
2 on profiles
3 as restrictive
4 for update
5 to authenticated using (
6 (select auth.jwt()->>'aal') = 'aal2'
7 );
```

## 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.



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

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

```
1 alter role "role_name" with bypassrls;
```

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

# 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 `tests`, we have a few recommendations for RLS:

## Add indexes

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

```
1 create policy "rls_test_select" on test_table
2 to authenticated
3 using ( (select auth.uid()) = user_id );
```

You can add an index like:

```
1 create index userid
2 on test_table
3 using btree (user_id);
```

## Benchmarks

Test	Before (ms)	After (ms)	% Improvement	Change
<a href="#">test1-indexed</a>	171	< 0.1	99.94%	<a href="#">► Details</a>

## Call functions with `select`

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

```
1 create policy "rls_test_select" on test_table
2 to authenticated
3 using ( auth.uid() = user_id );
```

You can do:

```
1 create policy "rls_test_select" on test_table
2 to authenticated
3 using ( (select auth.uid()) = user_id );
```

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.

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

## Benchmarks

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

## 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):

```
1  const { data } = supabase
2    .from('table')
3    .select()
```

You should always add a filter:

```
1  const { data } = supabase
2    .from('table')
3    .select()
4    .eq('user_id', userId)
```

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
<a href="#">test3-addfilter</a>	171	9	94.74%	► Details

## 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:

```
1  create policy "rls_test_select" on test_table
2  to authenticated
3  using (
4    exists (
5      select 1 from roles_table
6      where (select auth.uid()) = user_id and role = 'good_role'
7    )
8  );
```

We can instead create a `security definer` function which can scan `roles_table` without any RLS penalties:

```
1  create function private.has_good_role()
2  returns boolean
3  language plpgsql
4  security definer -- will run as the creator
5  as $$
```

```
6 begin
7     return exists (
8         select 1 from roles_table
9         where (select auth.uid()) = user_id and role = 'good_role'
10    );
11 end;
12 $$;
13
14 -- Update our policy to use this function:
15 create policy "rls_test_select"
16 on test_table
17 to authenticated
18 using ( private.has_good_role() );
```



Security-definer functions should never be created in a schema in the "Exposed schemas" inside your [API settings](#).

## 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` :

```
1 create policy "rls_test_select" on test_table
2 to authenticated
3 using (
4     (select auth.uid()) in (
5         select user_id
6         from team_user
7         where team_user.team_id = team_id -- joins to the source "test_table.team_id"
8     )
9 );
```

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

```
1 create policy "rls_test_select" on test_table
2 to authenticated
3 using (
4     team_id in (
5         select team_id
6         from team_user
7         where user_id = (select auth.uid()) -- no join
8     )
9 );
```

In this case you can also consider using a `security definer` function to bypass RLS on the join table:



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.

## Benchmarks

Test	Before (ms)	After (ms)	% Improvement	Change
<a href="#">test5-fixed-join</a>	9,000	20	99.78%	<a href="#">► Details</a>

## Specify roles in your policies



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

```
1 create policy "rls_test_select" on rls_test
2 using ( auth.uid() = user_id );
```

Use:

```
1 create policy "rls_test_select" on rls_test
2 to authenticated
3 using ( (select auth.uid()) = user_id );
```

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
test6-To-role	170	< 0.1	99.78%	► Details

## More resources

- [Testing your database](#)
- [Row Level Security and Supabase Auth](#)
- [RLS Guide and Best Practices](#)
- [Community repo on testing RLS using pgTAP and dbdev](#)

Edit this page on GitHub [↗](#)

- 🔧 Need some help? [Contact support](#)
- 📄 Latest product updates? [See Changelog](#)
- ✅ Something's not right? [Check system status](#)