supabase DOCS

Search

JavaScript Client Library

@supabase/supabase-js

This reference documents every object and method available in Supabase's isomorphic JavaScript library, `supabase-js`. You can use `supabase-js` to interact with your Postgres database, listen to database changes, invoke Deno Edge Functions, build login and user management functionality, and manage large files.

Install as package

You can install @supabase/supabase-js via the terminal.

npm Yarn pnpm

Terminal

```
npm install @supabase/supabase-js
```

Install via CDN

Installing

```
<script src="https://cdn.jsdelivr.net/npm/@supabase/supabase-js@2"></script>
//or
<script src="https://unpkg.com/@supabase/supabase-js@2"></script>
```

Use at runtime in Deno

You can use supabase-js in the Deno runtime via JSR:

```
import { createClient } from 'jsr:@supabase/supabase-js@2'
```

Create a new client for use in the browser.

You can initialize a new Supabase client using the `createClient()` method.

The Supabase client is your entrypoint to the rest of the Supabase functionality and is the easiest way to interact with everything we offer within the Supabase ecosystem.

Parameters

Initializing

`options` `Optional` SupabaseClientOptions

Details

Creating a client | With a custom domain | With additional parameters | With custom schemas | Custom fetch implementation | React Native options with AsyncStorage | React Native options with Expo SecureStore

```
1 import { createClient } from '@supabase/supabase-js'
2
3 // Create a single supabase client for interacting with your database
4 const supabase = createClient('https://xyzcompany.supabase.co', 'public-anon-ke
```

`supabase-js` has TypeScript support for type inference, autocompletion, type-safe queries, and more.

With TypeScript, `supabase-js` detects things like `not null` constraints and generated columns. Nullable columns are typed as `T | null` when you select the column. Generated columns will show a type error when you insert to it.

`supabase-js` also detects relationships between tables. A referenced table with one-to-many relationship is typed as `T[]`. Likewise, a referenced table with many-to-one relationship is typed as `T | null`.

Generating TypeScript Types

You can use the Supabase CLI to generate the types. You can also generate the types from the dashboard.

Terminal

```
supabase gen types typescript --project-id abcdefghijklmnopqrst > database.type
```

These types are generated from your database schema. Given a table `public.movies`, the generated types will look like:

```
create table public.movies (
  id bigint generated always as identity primary key,
  name text not null,
  data jsonb null
);
```

./database.types.ts

```ts
export type Json = string | number | boolean | null | { [key: string]: Json | u

export interface Database {
  public: {
    Tables: {
      movies: {
        Row: {              // the data expected from .select()
          id: number
          name: string
          data: Json | null
        }
        Insert: {           // the data to be passed to .insert()
          id?: never        // generated columns must not be supplied
          name: string      // `not null` columns with no default must be supp
          data?: Json | null // nullable columns can be omitted
        }
        Update: {           // the data to be passed to .update()
          id?: never
          name?: string     // `not null` columns are optional on .update()
          data?: Json | null
        }
      }
    }
  }
}
```

Using TypeScript type definitions

You can supply the type definitions to `supabase-js` like so:

Javascript Reference v2.0

./index.tsx

```tsx
import { createClient } from '@supabase/supabase-js'
import { Database } from './database.types'

const supabase = createClient<Database>(
  process.env.SUPABASE_URL,
  process.env.SUPABASE_ANON_KEY
)
```

Helper types for Tables and Joins

You can use the following helper types to make the generated TypeScript types easier to use.

TypeScript support

./database-generated.types.ts

```ts
export type Json = // ...

export interface Database {
  // ...
}
```

./database.types.ts

```ts
import { MergeDeep } from 'type-fest'
import { Database as DatabaseGenerated } from './database-generated.types'
export { Json } from './database-generated.types'

// Override the type for a specific column in a view:
export type Database = MergeDeep<
  DatabaseGenerated,
  {
    public: {
      Views: {
        movies_view: {
          Row: {
            // id is a primary key in public.movies, so it must be `not null`
            id: number
          }
        }
      }
    }
  }
>
```

You can also override the type of an individual successful response if needed:

```ts
const { data } = await supabase.from('countries').select().returns<MyType>()
```

The generated types provide shorthands for accessing tables and enums.

./index.ts

```ts
import { Database, Tables, Enums } from "./database.types.ts";

// Before 😕
let movie: Database['public']['Tables']['movies']['Row'] = // ...

// After 😍
let movie: Tables<'movies'>
```

Response types for complex queries

`supabase-js` always returns a `data` object (for success), and an `error` object (for unsuccessful requests).

These helper types provide the result types from any query, including nested types for database joins.

Given the following schema with a relation between cities and countries, we can get the nested `CountriesWithCities` type:

```sql
create table countries (
  "id" serial primary key,
  "name" text
);

create table cities (
  "id" serial primary key,
  "name" text,
  "country_id" int references "countries"
);
```

```
import { QueryResult, QueryData, QueryError } from '@supabase/supabase-js'

const countriesWithCitiesQuery = supabase
  .from("countries")
  .select(`
    id,
    name,
    cities (
      id,
      name
    )
  `);
type CountriesWithCities = QueryData<typeof countriesWithCitiesQuery>;

const { data, error } = await countriesWithCitiesQuery;
if (error) throw error;
const countriesWithCities: CountriesWithCities = data;
```

Perform a SELECT query on the table or view.

By default, Supabase projects return a maximum of 1,000 rows. This setting can be changed in your project's API settings. It's recommended that you keep it low to limit the payload size of accidental or malicious requests. You can use `range()` queries to paginate through your data.

`select()` can be combined with Filters

`select()` can be combined with Modifiers

`apikey` is a reserved keyword if you're using the Supabase Platform and should be avoided as a column name.

Fetch data

`options` REQUIRED object

Named parameters

Details

Getting your data | Selecting specific columns | Query referenced tables | Query referenced tables through a join table | Query the same referenced table multiple times | Query nested foreign tables through a join table | Filtering through referenced tables | Querying referenced table with count | Querying with count option | Querying JSON data | Querying referenced table with inner join | Switching schemas per query

```
const { data, error } = await supabase
  .from('countries')
  .select()
```

Data source

Response

Insert data

Perform an INSERT into the table or view.

Parameters

`values` REQUIRED Union: expand to see options

The values to insert. Pass an object to insert a single row or an array to insert multiple rows.

Details

`options` `Optional` object

Named parameters

Details

Create a record | Create a record and return it | Bulk create

```
const { error } = await supabase
  .from('countries')
  .insert({ id: 1, name: 'Denmark' })
```

Data source

Response

Update data

Perform an UPDATE on the table or view.

`update()` should always be combined with Filters to target the item(s) you wish to update.

Parameters

`values` REQUIRED Row

The values to update with

`options` REQUIRED object

Named parameters

Details

Updating your data | Update a record and return it | Updating JSON data

```
const { error } = await supabase
  .from('countries')
  .update({ name: 'Australia' })
  .eq('id', 1)
```

Data source

Response

Upsert data

Perform an UPSERT on the table or view. Depending on the column(s) passed to `onConflict`, `.upsert()` allows you to perform the equivalent of `.insert()` if a row with the corresponding `onConflict` columns doesn't exist, or if it does exist, perform an alternative action depending on `ignoreDuplicates`.

> Primary keys must be included in `values` to use upsert.

Parameters

`values` REQUIRED Union: expand to see options

The values to upsert with. Pass an object to upsert a single row or an array to upsert multiple rows.

Details

`options` Optional object

Named parameters

Details

Upsert your data | Bulk Upsert your data | Upserting into tables with constraints

```
const { data, error } = await supabase
  .from('countries')
  .upsert({ id: 1, name: 'Albania' })
  .select()
```

Data source

Response

Delete data

Perform a DELETE on the table or view.

`delete()` should always be combined with filters to target the item(s) you wish to delete.

If you use `delete()` with filters and you have RLS enabled, only rows visible through `SELECT` policies are deleted. Note that by default no rows are visible, so you need at least one `SELECT` / `ALL` policy that makes the rows visible.

When using `delete().in()`, specify an array of values to target multiple rows with a single query. This is particularly useful for batch deleting entries that share common criteria, such as deleting users by their IDs. Ensure that the array you provide accurately represents all records you intend to delete to avoid unintended data removal.

Parameters

`options` REQUIRED object

Named parameters

Details

Delete a single record | Delete a record and return it | Delete multiple records

```
const response = await supabase
  .from('countries')
  .delete()
  .eq('id', 1)
```

Data source

Response

Call a Postgres function

Perform a function call.

You can call Postgres functions as *Remote Procedure Calls*, logic in your database that you can execute from anywhere. Functions are useful when the logic rarely changes—like for password resets and updates.

```
create or replace function hello_world() returns text as $$
  select 'Hello world';
$$ language sql;
```

To call Postgres functions on Read Replicas, use the `get: true` option.

Parameters

`fn` REQUIRED FnName

The function name to call

`args` REQUIRED Fn['Args']

The arguments to pass to the function call

`options` REQUIRED object

Named parameters

Details

Call a Postgres function without arguments | Call a Postgres function with arguments | Bulk processing | Call a Postgres function with filters | Call a read-only Postgres function

```
const { data, error } = await supabase.rpc('hello_world')
```

Data source

> Response

Using filters

Filters allow you to only return rows that match certain conditions.

Filters can be used on `select()`, `update()`, `upsert()`, and `delete()` queries.

If a Postgres function returns a table response, you can also apply filters.

Applying Filters | Chaining | Conditional Chaining | Filter by values within a JSON column | Filter referenced tables

```
const { data, error } = await supabase
  .from('cities')
  .select('name, country_id')
  .eq('name', 'The Shire')    // Correct

const { data, error } = await supabase
  .from('cities')
  .eq('name', 'The Shire')    // Incorrect
  .select('name, country_id')
```

> Notes

Column is equal to a value

Match only rows where `column` is equal to `value`.

Parameters

`column` REQUIRED Union: expand to see options

The column to filter on

Details

`value` REQUIRED NonNullable

The value to filter with

With `select()`

```
const { data, error } = await supabase
  .from('countries')
  .select()
  .eq('name', 'Albania')
```

Data source

Response

Column is not equal to a value

Match only rows where `column` is not equal to `value`.

Parameters

`column` REQUIRED Union: expand to see options

The column to filter on

Details

`value` REQUIRED Union: expand to see options

The value to filter with

Details

With `select()`

```
const { data, error } = await supabase
  .from('countries')
  .select()
  .neq('name', 'Albania')
```

Data source

Response

Column is greater than a value

Match only rows where `column` is greater than `value`.

Parameters

`column` REQUIRED Union: expand to see options

The column to filter on

Details

`value` REQUIRED Union: expand to see options

The value to filter with

Details

With `select()`

```
const { data, error } = await supabase
  .from('countries')
  .select()
  .gt('id', 2)
```

Data source

Response

Notes

Column is greater than or equal to a value

Match only rows where `column` is greater than or equal to `value`.

Parameters

`column` REQUIRED Union: expand to see options

The column to filter on

Details

`value` REQUIRED Union: expand to see options

The value to filter with

Details

With `select()`

```
const { data, error } = await supabase
  .from('countries')
  .select()
  .gte('id', 2)
```

Data source

Response

Column is less than a value

Match only rows where `column` is less than `value`.

Parameters

`column` REQUIRED Union: expand to see options

The column to filter on

Details

`value` REQUIRED Union: expand to see options

The value to filter with

Details

With `select()`

```
const { data, error } = await supabase
  .from('countries')
  .select()
  .lt('id', 2)
```

Data source

Response

Column is less than or equal to a value

Match only rows where `column` is less than or equal to `value`.

Parameters

`column` REQUIRED Union: expand to see options

The column to filter on

Details

`value` REQUIRED Union: expand to see options

The value to filter with

Details

With `select()`

```
const { data, error } = await supabase
  .from('countries')
  .select()
  .lte('id', 2)
```

Data source

Response

Column matches a pattern

Match only rows where `column` matches `pattern` case-sensitively.

Parameters

`column` REQUIRED Union: expand to see options

The column to filter on

Details

`pattern` REQUIRED string

The pattern to match with

With `select()`

```
const { data, error } = await supabase
  .from('countries')
  .select()
  .like('name', '%Alba%')
```

Data source

Response

Column matches a case-insensitive pattern

Match only rows where `column` matches `pattern` case-insensitively.

Parameters

`column` REQUIRED Union: expand to see options

The column to filter on

Details

`pattern` REQUIRED string

The pattern to match with

With `select()`

```
const { data, error } = await supabase
  .from('countries')
  .select()
  .ilike('name', '%alba%')
```

> Data source

> Response

Column is a value

Match only rows where `column` IS `value`.

Parameters

`column` REQUIRED Union: expand to see options

The column to filter on

Details

`value` REQUIRED Union: expand to see options

The value to filter with

Details

Checking for nullness, true or false

```
const { data, error } = await supabase
  .from('countries')
  .select()
  .is('name', null)
```

Data source

Response

Notes

Column is in an array

Match only rows where `column` is included in the `values` array.

Parameters

`column` REQUIRED Union: expand to see options

The column to filter on

Details

`values` REQUIRED Array<Row['ColumnName']>

The values array to filter with

With `select()`

```
const { data, error } = await supabase
  .from('countries')
  .select()
  .in('name', ['Albania', 'Algeria'])
```

Data source

Response

Column contains every element in a value

Only relevant for jsonb, array, and range columns. Match only rows where `column` contains every element appearing in `value`.

Parameters

`column` REQUIRED Union: expand to see options

The jsonb, array, or range column to filter on

Details

`value` REQUIRED Union: expand to see options

The jsonb, array, or range value to filter with

Details

On array columns | On range columns | On `jsonb` columns

```
const { data, error } = await supabase
  .from('issues')
  .select()
  .contains('tags', ['is:open', 'priority:low'])
```

Data source

Response

Contained by value

Only relevant for jsonb, array, and range columns. Match only rows where every element appearing in `column` is contained by `value`.

Parameters

`column` REQUIRED Union: expand to see options

The jsonb, array, or range column to filter on

Details

`value` REQUIRED Union: expand to see options

The jsonb, array, or range value to filter with

Details

On array columns | On range columns | On `jsonb` columns

```
const { data, error } = await supabase
  .from('classes')
  .select('name')
  .containedBy('days', ['monday', 'tuesday', 'wednesday', 'friday'])
```

Data source

Response

Greater than a range

Only relevant for range columns. Match only rows where every element in `column` is greater than any element in `range`.

Parameters

`column` REQUIRED Union: expand to see options

The range column to filter on

Details

`range` REQUIRED string

The range to filter with

With `select()`

```
const { data, error } = await supabase
  .from('reservations')
  .select()
  .rangeGt('during', '[2000-01-02 08:00, 2000-01-02 09:00)')
```

Data source

Response

Notes

Greater than or equal to a range

Only relevant for range columns. Match only rows where every element in `column` is either contained in `range` or greater than any element in `range`.

Parameters

`column` REQUIRED Union: expand to see options

The range column to filter on

Details

`range` REQUIRED string

The range to filter with

With `select()`

```
const { data, error } = await supabase
  .from('reservations')
  .select()
  .rangeGte('during', '[2000-01-02 08:30, 2000-01-02 09:30)')
```

Data source

Response

Notes

Less than a range

Only relevant for range columns. Match only rows where every element in `column` is less than any element in `range`.

Parameters

`column` REQUIRED Union: expand to see options

The range column to filter on

Details

`range` REQUIRED string

The range to filter with

With `select()`

```
const { data, error } = await supabase
  .from('reservations')
  .select()
  .rangeLt('during', '[2000-01-01 15:00, 2000-01-01 16:00)')
```

Data source

Response

Notes

Less than or equal to a range

Only relevant for range columns. Match only rows where every element in `column` is either contained in `range` or less than any element in `range`.

Parameters

`column` REQUIRED Union: expand to see options

The range column to filter on

Details

`range` REQUIRED string

The range to filter with

With `select()`

```
const { data, error } = await supabase
  .from('reservations')
  .select()
  .rangeLte('during', '[2000-01-01 14:00, 2000-01-01 16:00)')
```

Data source

Response

Notes

Mutually exclusive to a range

Only relevant for range columns. Match only rows where `column` is mutually exclusive to `range` and there can be no element between the two ranges.

Parameters

`column` REQUIRED Union: expand to see options

The range column to filter on

Details

`range` REQUIRED string

The range to filter with

With `select()`

```
const { data, error } = await supabase
```

```
  .from('reservations')
  .select()
  .rangeAdjacent('during', '[2000-01-01 12:00, 2000-01-01 13:00)')
```

Data source

Response

Notes

With a common element

Only relevant for array and range columns. Match only rows where `column` and `value` have an element in common.

Parameters

`column` REQUIRED Union: expand to see options

The array or range column to filter on

Details

`value` REQUIRED Union: expand to see options

The array or range value to filter with

Details

On array columns | On range columns

```
const { data, error } = await supabase
  .from('issues')
  .select('title')
  .overlaps('tags', ['is:closed', 'severity:high'])
```

> Data source

> Response

Match a string

Only relevant for text and tsvector columns. Match only rows where `column` matches the query string in `query`.

For more information, see Postgres full text search.

Parameters

`column` REQUIRED Union: expand to see options

The text or tsvector column to filter on

Details

`query` REQUIRED string

The query text to match with

`options` Optional object

Named parameters

Details

Text search | Basic normalization | Full normalization | Websearch

```
const result = await supabase
  .from("texts")
  .select("content")
  .textSearch("content", `'eggs' & 'ham'`, {
    config: "english",
  });
```

Data source

Response

Match an associated value

Match only rows where each column in `query` keys is equal to its associated value. Shorthand for multiple `.eq()` s.

Parameters

`query` REQUIRED Union: expand to see options

The object to filter with, with column names as keys mapped to their filter values

Details

With `select()`

```
const { data, error } = await supabase
  .from('countries')
  .select('name')
  .match({ id: 2, name: 'Albania' })
```

> Data source

> Response

Don't match the filter

Match only rows which doesn't satisfy the filter.

not() expects you to use the raw PostgREST syntax for the filter values.

```
.not('id', 'in', '(5,6,7)')  // Use `()` for `in` filter
.not('arraycol', 'cs', '{"a","b"}')  // Use `cs` for `contains
```

Parameters

`column` REQUIRED Union: expand to see options

The column to filter on

Details

`operator` REQUIRED Union: expand to see options

The operator to be negated to filter with, following PostgREST syntax

Details

`value` REQUIRED Union: expand to see options

The value to filter with, following PostgREST syntax

Details

With `select()`

```
const { data, error } = await supabase
  .from('countries')
  .select()
  .not('name', 'is', null)
```

Data source

Response

Match at least one filter

Match only rows which satisfy at least one of the filters.

or() expects you to use the raw PostgREST syntax for the filter names and values.

```
.or('id.in.(5,6,7), arraycol.cs.{"a","b"}')  // Use `()` for `
.or('id.in.(5,6,7), arraycol.cd.{"a","b"}')  // Use `cd` for `
```

Parameters

`filters` REQUIRED string

The filters to use, following PostgREST syntax

`options` REQUIRED object

Named parameters

Details

With `select()` Use `or` with `and` Use `or` on referenced tables

```
const { data, error } = await supabase
  .from('countries')
  .select('name')
  .or('id.eq.2,name.eq.Algeria')
```

Data source

Response

Match the filter

Match only rows which satisfy the filter. This is an escape hatch - you should use the specific filter methods wherever possible.

filter() expects you to use the raw PostgREST syntax for the filter values.

```
.filter('id', 'in', '(5,6,7)')  // Use `()` for `in` filter
.filter('arraycol', 'cs', '{"a","b"}')  // Use `cs` for `conta
```

Parameters

`column` REQUIRED Union: expand to see options

The column to filter on

Details

`operator` REQUIRED Union: expand to see options

The operator to filter with, following PostgREST syntax

Details

`value` REQUIRED unknown

The value to filter with, following PostgREST syntax

With `select()` On a referenced table

```
const { data, error } = await supabase
  .from('countries')
  .select()
  .filter('name', 'in', '("Algeria","Japan")')
```

Data source

Response

Using modifiers

Filters work on the row level—they allow you to return rows that only match certain conditions without changing the shape of the rows. Modifiers are everything that don't fit that definition—allowing you to change the format of the response (e.g., returning a CSV string).

Modifiers must be specified after filters. Some modifiers only apply for queries that return rows (e.g., `select()` or `rpc()` on a function that returns a table response).

Return data after inserting

Perform a SELECT on the query result.

Parameters

`columns` `Optional` Query

The columns to retrieve, separated by commas

With `upsert()`

```
const { data, error } = await supabase
  .from('countries')
  .upsert({ id: 1, name: 'Algeria' })
  .select()
```

> Data source

Response

Order the results

Order the query result by `column`.

Parameters

`column` REQUIRED Union: expand to see options

The column to order by

Details

`options` Optional object

Named parameters

Details

With `select()` On a referenced table

```
const { data, error } = await supabase
  .from('countries')
  .select('id', 'name')
  .order('id', { ascending: false })
```

Data source

Response

Limit the number of rows returned

Limit the query result by `count`.

Parameters

`count` REQUIRED number

The maximum number of rows to return

`options` REQUIRED object

Named parameters

Details

With `select()` On a referenced table

```
const { data, error } = await supabase
  .from('countries')
  .select('name')
  .limit(1)
```

Data source

Response

Limit the query to a range

Limit the query result by starting at an offset (`from`) and ending at the offset (`from + to`). Only records within this range are returned. This respects the query order and if there is no order clause the range could behave unexpectedly. The `from` and `to` values are 0-based and inclusive: `range(1, 3)` will include the second, third and fourth rows of the query.

Parameters

`from` REQUIRED number

The starting index from which to limit the result

`to` REQUIRED number

The last index to which to limit the result

`options` REQUIRED object

Named parameters

Details

With `select()`

```
const { data, error } = await supabase
  .from('countries')
  .select('name')
  .range(0, 1)
```

Data source

Response

Set an abort signal

Set the AbortSignal for the fetch request.

You can use this to set a timeout for the request.

Parameters

`signal` REQUIRED AbortSignal

The AbortSignal to use for the fetch request

Aborting requests in-flight | Set a timeout

```
const ac = new AbortController()
ac.abort()
const { data, error } = await supabase
  .from('very_big_table')
  .select()
  .abortSignal(ac.signal)
```

> Response

> Notes

Retrieve one row of data

Return `data` as a single object instead of an array of objects.

With `select()`

```
const { data, error } = await supabase
  .from('countries')
  .select('name')
  .limit(1)
  .single()
```

> Data source

> Response

Retrieve zero or one row of data

Return `data` as a single object instead of an array of objects.

Return Type

Union: expand to see options

Details

With `select()`

```
const { data, error } = await supabase
  .from('countries')
  .select()
  .eq('name', 'Singapore')
  .maybeSingle()
```

> Data source

> Response

Retrieve as a CSV

Return `data` as a string in CSV format.

Return Type

string

Return data as CSV

```
const { data, error } = await supabase
  .from('countries')
  .select()
  .csv()
```

> Data source

> Response

> Notes

Override type of successful response

Override the type of the returned `data`.

Override type of successful response

```
const { data } = await supabase
  .from('countries')
  .select()
  .returns<MyType>()
```

> Response

Using explain

Return `data` as the EXPLAIN plan for the query.

For debugging slow queries, you can get the Postgres `EXPLAIN` execution plan of a query using the `explain()` method. This works on any query, even for `rpc()` or writes.

Explain is not enabled by default as it can reveal sensitive information about your database. It's best to only enable this for testing environments but if you wish to enable it for production you can provide additional protection by using a `pre-request` function.

Follow the Performance Debugging Guide to enable the functionality on your project.

Parameters

`options` REQUIRED object

Named parameters

Details

Return Type

Union: expand to see options

Details

Get the execution plan | Get the execution plan with analyze and verbose

```
1 const { data, error } = await supabase
2   .from('countries')
3   .select()
4   .explain()
```

Data source

Response

Notes

Overview

The auth methods can be accessed via the `supabase.auth` namespace.

By default, the supabase client sets `persistSession` to true and attempts to store the session in local storage. When using the supabase client in an environment that doesn't support local storage, you might notice the following warning message being logged:

> *"No storage option exists to persist the session, which may result in unexpected behavior when using auth. If you want to set `persistSession` to true, please provide a storage option or you may set `persistSession` to false to disable this warning."*

This warning message can be safely ignored if you're not using auth on the server-side. If you are using auth and you want to set `persistSession` to true, you will need to provide a custom storage implementation that follows this interface.

Any email links and one-time passwords (OTPs) sent have a default expiry of 24 hours. We have the following rate limits in place to guard against brute force attacks.

The expiry of an access token can be set in the "JWT expiry limit" field in your project's auth settings. A refresh token never expires and can only be used once.

Create auth client | Create auth client (server-side)

```
import { createClient } from '@supabase/supabase-js'

const supabase = createClient(supabase_url, anon_key)
```

Create a new user

Creates a new user.

By default, the user needs to verify their email address before logging in. To turn this off, disable **Confirm email** in your project.

Confirm email determines if users need to confirm their email address after signing up.

If **Confirm email** is enabled, a `user` is returned but `session` is null.

If **Confirm email** is disabled, both a `user` and a `session` are returned.

When the user confirms their email address, they are redirected to the `SITE_URL` by default. You can modify your `SITE_URL` or add additional redirect URLs in your project.

If signUp() is called for an existing confirmed user:

When both **Confirm email** and **Confirm phone** (even when phone provider is disabled) are enabled in your project, an obfuscated/fake user object is returned.

When either **Confirm email** or **Confirm phone** (even when phone provider is disabled) is disabled, the error message, `User already registered` is returned.

To fetch the currently logged-in user, refer to `getUser()`.

Parameters

`credentials` REQUIRED Union: expand to see options

Details

Return Type

Promise<Union: expand to see options>

Details

Sign up with an email and password | Sign up with a phone number and password (SMS)

Sign up with a phone number and password (whatsapp)

Sign up with additional user metadata

Sign up with a redirect URL

```
const { data, error } = await supabase.auth.signUp({
  email: 'example@email.com',
  password: 'example-password',
})
```

Response

Listen to auth events

Receive a notification every time an auth event happens.

Subscribes to important events occurring on the user's session.

Use on the frontend/client. It is less useful on the server.

Events are emitted across tabs to keep your application's UI up-to-date. Some events can fire very frequently, based on the number of tabs open. Use a quick and efficient callback function, and defer or debounce as many operations as you can to be performed outside of the callback.

Important: A callback can be an `async` function and it runs synchronously during the processing of the changes causing the event. You can easily create a dead-lock by using `await` on a call to another method of the Supabase library.

Avoid using `async` functions as callbacks.

Limit the number of `await` calls in `async` callbacks.

Do not use other Supabase functions in the callback function. If you must, dispatch the functions once the callback has finished executing. Use this as a quick way to achieve this:

```
supabase.auth.onAuthStateChange((event, session) => {
```

```
  setTimeout(async () => {
    // await on other Supabase function here
    // this runs right after the callback has finished
  }, 0)
})
```

Emitted events:

`INITIAL_SESSION`

Emitted right after the Supabase client is constructed and the initial session from storage is loaded.

`SIGNED_IN`

Emitted each time a user session is confirmed or re-established, including on user sign in and when refocusing a tab.

Avoid making assumptions as to when this event is fired, this may occur even when the user is already signed in. Instead, check the user object attached to the event to see if a new user has signed in and update your application's UI.

This event can fire very frequently depending on the number of tabs open in your application.

`SIGNED_OUT`

Emitted when the user signs out. This can be after:

A call to `supabase.auth.signOut()`.

After the user's session has expired for any reason:

User has signed out on another device.

The session has reached its timebox limit or inactivity timeout.

User has signed in on another device with single session per user enabled.

Check the User Sessions docs for more information.

Use this to clean up any local storage your application has associated with the user.

`TOKEN_REFRESHED`

Emitted each time a new access and refresh token are fetched for the signed in user.

It's best practice and highly recommended to extract the access token (JWT) and store it in memory for further use in your application.

Avoid frequent calls to `supabase.auth.getSession()` for the same purpose.

There is a background process that keeps track of when the session should be refreshed so you will always receive valid tokens by listening to this event.

The frequency of this event is related to the JWT expiry limit configured on your project.

`USER_UPDATED`

Emitted each time the `supabase.auth.updateUser()` method finishes successfully. Listen to it to update your application's UI based on new profile information.

`PASSWORD_RECOVERY`

Emitted instead of the `SIGNED_IN` event when the user lands on a page that includes a password recovery link in the URL.

Use it to show a UI to the user where they can reset their password.

Parameters

`callback` REQUIRED function

A callback function to be invoked when an auth event happens.

Details

Return Type

object

Details

Listen to auth changes | Listen to sign out | Store OAuth provider tokens on sign in | Use React Context for the User's session | Listen to password recovery events | Listen to sign in | Listen to token refresh | Listen to user updates

```
const { data } = supabase.auth.onAuthStateChange((event, session) => {
  console.log(event, session)

  if (event === 'INITIAL_SESSION') {
    // handle initial session
  } else if (event === 'SIGNED_IN') {
    // handle sign in event
  } else if (event === 'SIGNED_OUT') {
    // handle sign out event
  } else if (event === 'PASSWORD_RECOVERY') {
```

```
    // handle password recovery event
  } else if (event === 'TOKEN_REFRESHED') {
    // handle token refreshed event
  } else if (event === 'USER_UPDATED') {
    // handle user updated event
  }
})

// call unsubscribe to remove the callback
data.subscription.unsubscribe()
```

Create an anonymous user

Creates a new anonymous user.

Returns an anonymous user

It is recommended to set up captcha for anonymous sign-ins to prevent abuse. You can pass in the captcha token in the `options` param.

Parameters

`credentials` `Optional` SignInAnonymouslyCredentials

Details

Return Type

Promise<Union: expand to see options>

Details

Create an anonymous user | Create an anonymous user with custom user metadata

```
const { data, error } = await supabase.auth.signInAnonymously({
  options: {
    captchaToken
  }
});
```

Response

Sign in a user

Log in an existing user with an email and password or phone and password.

Requires either an email and password or a phone number and password.

Parameters

`credentials` REQUIRED Union: expand to see options

Details

Return Type

Promise<Union: expand to see options>

Details

Sign in with email and password | Sign in with phone and password

```
const { data, error } = await supabase.auth.signInWithPassword({
  email: 'example@email.com',
  password: 'example-password',
})
```

Response

Sign in with ID Token

Allows signing in with an OIDC ID token. The authentication provider used should be enabled and configured.

Parameters

`credentials` REQUIRED SignInWithIdTokenCredentials

Details

Return Type

Promise<Union: expand to see options>

Details

Sign In using ID Token

```
const { data, error } = await supabase.auth.signInWithIdToken({
  provider: 'google',
  token: 'your-id-token'
})
```

Response

Sign in a user through OTP

Log in a user using magiclink or a one-time password (OTP).

Requires either an email or phone number.

This method is used for passwordless sign-ins where a OTP is sent to the user's email or phone number.

If the user doesn't exist, `signInWithOtp()` will signup the user instead. To restrict this behavior, you can set `shouldCreateUser` in `SignInWithPasswordlessCredentials.options` to `false`.

If you're using an email, you can configure whether you want the user to receive a magiclink or a OTP.

If you're using phone, you can configure whether you want the user to receive a OTP.

The magic link's destination URL is determined by the `SITE_URL`.

See redirect URLs and wildcards to add additional redirect URLs to your project.

Magic links and OTPs share the same implementation. To send users a one-time code instead of a magic link, modify the magic link email template to include `{{ .Token }}` instead of `{{ .ConfirmationURL }}`.

See our Twilio Phone Auth Guide for details about configuring WhatsApp sign in.

Parameters

`credentials` REQUIRED Union: expand to see options

Details

Return Type

Promise<Union: expand to see options>

Details

Sign in with email | Sign in with SMS OTP | Sign in with WhatsApp OTP

```
const { data, error } = await supabase.auth.signInWithOtp({
  email: 'example@email.com',
  options: {
    emailRedirectTo: 'https://example.com/welcome'
  }
})
```

Response

Notes

Sign in a user through OAuth

Log in an existing user via a third-party provider. This method supports the PKCE flow.

This method is used for signing in using a third-party provider.

Supabase supports many different third-party providers.

Parameters

`credentials` REQUIRED SignInWithOAuthCredentials

Details

Return Type

Promise<Union: expand to see options>

Details

Sign in using a third-party provider | Sign in using a third-party provider with redirect

Sign in with scopes and access provider tokens

```
1 const { data, error } = await supabase.auth.signInWithOAuth({
2   provider: 'github'
3 })
```

Response

Sign in a user through SSO

Attempts a single-sign on using an enterprise Identity Provider. A successful SSO attempt will redirect the current page to the identity provider authorization page. The redirect URL is implementation and SSO protocol specific.

> Before you can call this method you need to establish a connection to an identity provider. Use the CLI commands to do this.

If you've associated an email domain to the identity provider, you can use the `domain` property to start a sign-in flow.

In case you need to use a different way to start the authentication flow with an identity provider, you can use the `providerId` property. For example:

Mapping specific user email addresses with an identity provider.

Using different hints to identity the identity provider to be used by the user, like a company-specific page, IP address or other tracking information.

Parameters

`params` REQUIRED Union: expand to see options

Details

Return Type

Promise<Union: expand to see options>

Details

Sign in with email domain | Sign in with provider UUID

```
// You can extract the user's email domain and use it to trigger the
// authentication flow with the correct identity provider.

const { data, error } = await supabase.auth.signInWithSSO({
  domain: 'company.com'
})

if (data?.url) {
  // redirect the user to the identity provider's authentication flow
  window.location.href = data.url
}
```

Sign out a user

Inside a browser context, `signOut()` will remove the logged in user from the browser session and log them out - removing all items from localstorage and then trigger a `"SIGNED_OUT"` event.

> In order to use the `signOut()` method, the user needs to be signed in first.
>
> By default, `signOut()` uses the global scope, which signs out all other sessions that the user is logged into as well.
>
> Since Supabase Auth uses JWTs for authentication, the access token JWT will be valid until it's expired. When the user signs out, Supabase revokes the refresh token and deletes the JWT from the client-side. This does not revoke the JWT and it will still be valid until it expires.

Parameters

`options` REQUIRED SignOut

Details

Return Type

Promise<object>

Details

Sign out

```
1 const { error } = await supabase.auth.signOut()
```

Send a password reset request

Sends a password reset request to an email address. This method supports the PKCE flow.

> The password reset flow consist of 2 broad steps: (i) Allow the user to login via the password reset link; (ii) Update the user's password.
>
> The `resetPasswordForEmail()` only sends a password reset link to the user's email. To update the user's password, see `updateUser()`.
>
> A `SIGNED_IN` and `PASSWORD_RECOVERY` event will be emitted when the password recovery link is clicked. You can use `onAuthStateChange()` to listen and invoke a callback function on these events.
>
> When the user clicks the reset link in the email they are redirected back to your application. You can configure the URL that the user is redirected to with the `redirectTo` parameter. See redirect URLs and wildcards to add additional redirect URLs to your project.
>
> After the user has been redirected successfully, prompt them for a new password and call `updateUser()`:

```
const { data, error } = await supabase.auth.updateUser({
  password: new_password
})
```

Parameters

`email` REQUIRED string

The email address of the user.

`options` REQUIRED object

Details

Return Type

Promise<Union: expand to see options>

Details

Reset password | Reset password (React)

```
const { data, error } = await supabase.auth.resetPasswordForEmail(email, {
  redirectTo: 'https://example.com/update-password',
})
```

Response

Verify and log in through OTP

Log in a user given a User supplied OTP or TokenHash received through mobile or email.

The `verifyOtp` method takes in different verification types. If a phone number is used, the type can either be `sms` or `phone_change`. If an email address is used, the type can be one of the following: `email`, `recovery`, `invite` or `email_change` (`signup` and `magiclink` types are deprecated).

The verification type used should be determined based on the corresponding auth method called before `verifyOtp` to sign up / sign-in a user.

The `TokenHash` is contained in the email templates and can be used to sign in. You may wish to use the hash with Magic Links for the PKCE flow for Server Side Auth. See this guide for more details.

Parameters

`params` REQUIRED Union: expand to see options

Details

Return Type

Promise<Union: expand to see options>

Details

Verify Signup One-Time Password (OTP) | Verify Sms One-Time Password (OTP) | Verify Email Auth (Token Hash)

```
1 const { data, error } = await supabase.auth.verifyOtp({ email, token, type: 'em
```

Response

Retrieve a session

Returns the session, refreshing it if necessary.

This method retrieves the current local session (i.e local storage).

The session contains a signed JWT and unencoded session data.

Since the unencoded session data is retrieved from the local storage medium, **do not** rely on it as a source of trusted data on the server. It could be tampered with by the sender. If you need verified, trustworthy user data, call `getUser` instead.

If the session has an expired access token, this method will use the refresh token to get a new session.

Return Type

Promise<Union: expand to see options>

Details

Get the session data

```
const { data, error } = await supabase.auth.getSession()
```

Response

Retrieve a new session

Returns a new session, regardless of expiry status. Takes in an optional current session. If not passed in, then refreshSession() will attempt to retrieve it from getSession(). If the current session's refresh token is invalid, an error will be thrown.

This method will refresh and return a new session whether the current one is expired or not.

Parameters

`currentSession` `Optional` object

The current session. If passed in, it must contain a refresh token.

Details

Return Type

Promise<Union: expand to see options>

Details

Refresh session using the current session | Refresh session using a refresh token

```
1 const { data, error } = await supabase.auth.refreshSession()
2 const { session, user } = data
```

Response

Retrieve a user

Gets the current user details if there is an existing session. This method performs a network request to the Supabase Auth server, so the returned value is authentic and can be used to base authorization rules on.

This method fetches the user object from the database instead of local session.

This method is useful for checking if the user is authorized because it validates the user's access token JWT on the server.

Should always be used when checking for user authorization on the server. On the client, you can instead use `getSession().session.user` for faster results. `getSession` is insecure on the server.

Parameters

`jwt` `Optional` string

Takes in an optional access token JWT. If no JWT is provided, the JWT from the current session is used.

Return Type

Promise<Union: expand to see options>

Details

Get the logged in user with the current existing session | Get the logged in user with a custom access token jwt

```
const { data: { user } } = await supabase.auth.getUser()
```

Response

Update a user

Updates user data for a logged in user.

In order to use the `updateUser()` method, the user needs to be signed in first.

By default, email updates sends a confirmation link to both the user's current and new email. To only send a confirmation link to the user's new email, disable **Secure email change** in your project's email auth provider settings.

Parameters

`attributes` REQUIRED UserAttributes

Details

`options` REQUIRED object

Details

Return Type

Promise<Union: expand to see options>

Details

Update the email for an authenticated user | Update the phone number for an authenticated user | Update the password for an authenticated user | Update the user's metadata | Update the user's password with a nonce

```
const { data, error } = await supabase.auth.updateUser({
  email: 'new@email.com'
})
```

Response

Notes

Retrieve identities linked to a user

Gets all the identities linked to a user.

The user needs to be signed in to call `getUserIdentities()`.

Return Type

Promise<Union: expand to see options>

Details

Returns a list of identities linked to the user

```
1 const { data, error } = await supabase.auth.getUserIdentities()
```

Response

Link an identity to a user

Links an oauth identity to an existing user. This method supports the PKCE flow.

The **Enable Manual Linking** option must be enabled from your project's authentication settings.

The user needs to be signed in to call `linkIdentity()`.

If the candidate identity is already linked to the existing user or another user, `linkIdentity()` will fail.

If `linkIdentity` is run in the browser, the user is automatically redirected to the returned URL. On the server, you should handle the redirect.

Parameters

`credentials` REQUIRED SignInWithOAuthCredentials

Details

Return Type

Promise<Union: expand to see options>

Details

Link an identity to a user

```
const { data, error } = await supabase.auth.linkIdentity({
  provider: 'github'
})
```

Response

Unlink an identity from a user

Unlinks an identity from a user by deleting it. The user will no longer be able to sign in with that identity once it's unlinked.

The **Enable Manual Linking** option must be enabled from your project's authentication settings.

The user needs to be signed in to call `unlinkIdentity()`.

The user must have at least 2 identities in order to unlink an identity.

The identity to be unlinked must belong to the user.

Parameters

`identity` REQUIRED UserIdentity

Details

Return Type

Promise<Union: expand to see options>

Details

Unlink an identity

```
// retrieve all identites linked to a user
const identities = await supabase.auth.getUserIdentities()

// find the google identity
const googleIdentity = identities.find(
  identity => identity.provider === 'google'
)

// unlink the google identity
const { error } = await supabase.auth.unlinkIdentity(googleIdentity)
```

Send a password reauthentication nonce

Sends a reauthentication OTP to the user's email or phone number. Requires the user to be signed-in.

This method is used together with `updateUser()` when a user's password needs to be updated.

If you require your user to reauthenticate before updating their password, you need to enable the **Secure password change** option in your project's email provider settings.

A user is only require to reauthenticate before updating their password if **Secure password change** is enabled and the user **hasn't recently signed in**. A user is deemed recently signed in if the session was created in the last 24 hours.

This method will send a nonce to the user's email. If the user doesn't have a confirmed email address, the method will send the nonce to the user's confirmed phone number instead.

Return Type

Promise<Union: expand to see options>

Details

Send reauthentication nonce

```
1 const { error } = await supabase.auth.reauthenticate()
```

Notes

Resend an OTP

Resends an existing signup confirmation email, email change email, SMS OTP or phone change OTP.

Resends a signup confirmation, email change or phone change email to the user.

Passwordless sign-ins can be resent by calling the `signInWithOtp()` method again.

Password recovery emails can be resent by calling the `resetPasswordForEmail()` method again.

This method will only resend an email or phone OTP to the user if there was an initial signup, email change or phone change request being made.

You can specify a redirect url when you resend an email link using the `emailRedirectTo` option.

Parameters

`credentials` REQUIRED Union: expand to see options

Details

Return Type

Promise<Union: expand to see options>

Details

Resend an email signup confirmation | Resend a phone signup confirmation | Resend email change email | Resend phone change OTP

```
const { error } = await supabase.auth.resend({
  type: 'signup',
  email: 'email@example.com',
  options: {
    emailRedirectTo: 'https://example.com/welcome'
  }
})
```

Notes

Set the session data

Sets the session data from the current session. If the current session is expired, setSession will take care of refreshing it to obtain a new session. If the refresh token or access token in the current session is invalid, an error will be thrown.

This method sets the session using an `access_token` and `refresh_token`.

If successful, a `SIGNED_IN` event is emitted.

Parameters

`currentSession` REQUIRED object

The current session that minimally contains an access token and refresh token.

Details

Return Type

Promise<Union: expand to see options>

Details

Set the session

```
const { data, error } = await supabase.auth.setSession({
  access_token,
  refresh_token
})
```

Response

Notes

Exchange an auth code for a session

Log in an existing user by exchanging an Auth Code issued during the PKCE flow.

Used when `flowType` is set to `pkce` in client options.

Parameters

`authCode` REQUIRED string

Return Type

Promise<Union: expand to see options>

Details

Exchange Auth Code

```
supabase.auth.exchangeCodeForSession('34e770dd-9ff9-416c-87fa-43b31d7ef225')
```

Response

Start auto-refresh session (non-browser)

Starts an auto-refresh process in the background. The session is checked every few seconds. Close to the time of expiration a process is started to refresh the session. If refreshing fails it will be retried for as long as necessary.

> Only useful in non-browser environments such as React Native or Electron.
>
> The Supabase Auth library automatically starts and stops proactively refreshing the session when a tab is focused or not.
>
> On non-browser platforms, such as mobile or desktop apps built with web technologies, the library is not able to effectively determine whether the application is *focused* or not.
>
> To give this hint to the application, you should be calling this method when the app is in focus and calling `supabase.auth.stopAutoRefresh()` when it's out of focus.

Return Type

Promise<void>

Start and stop auto refresh in React Native

```
import { AppState } from 'react-native'

// make sure you register this only once!
AppState.addEventListener('change', (state) => {
  if (state === 'active') {
    supabase.auth.startAutoRefresh()
  } else {
    supabase.auth.stopAutoRefresh()
  }
})
```

Stop auto-refresh session (non-browser)

Stops an active auto refresh process running in the background (if any).

Only useful in non-browser environments such as React Native or Electron.

The Supabase Auth library automatically starts and stops proactively refreshing the session when a tab is focused or not.

On non-browser platforms, such as mobile or desktop apps built with web technologies, the library is not able to effectively determine whether the application is *focused* or not.

When your application goes in the background or out of focus, call this method to stop the proactive refreshing of the session.

Return Type

Promise<void>

Start and stop auto refresh in React Native

```
import { AppState } from 'react-native'

// make sure you register this only once!
AppState.addEventListener('change', (state) => {
  if (state === 'active') {
    supabase.auth.startAutoRefresh()
  } else {
    supabase.auth.stopAutoRefresh()
  }
})
```

Auth MFA

This section contains methods commonly used for Multi-Factor Authentication (MFA) and are invoked behind the `supabase.auth.mfa` namespace.

Currently, there is support for time-based one-time password (TOTP) and phone verification code as the 2nd factor. Recovery codes are not supported but users can enroll multiple factors, with an upper limit of 10.

Having a 2nd factor for recovery frees the user of the burden of having to store their recovery codes somewhere. It also reduces the attack surface since multiple recovery codes are usually generated compared to just having 1 backup factor.

Enroll a factor

Starts the enrollment process for a new Multi-Factor Authentication (MFA) factor. This method creates a new `unverified` factor. To verify a factor, present the QR code or secret to the user and ask them to add it to their authenticator app. The user has to enter the code from their authenticator app to verify it.

Use `totp` or `phone` as the `factorType` and use the returned `id` to create a challenge.

To create a challenge, see `mfa.challenge()`.

To verify a challenge, see `mfa.verify()`.

To create and verify a TOTP challenge in a single step, see `mfa.challengeAndVerify()`.

To generate a QR code for the `totp` secret in Next.js, you can do the following:

```
<Image src={data.totp.qr_code} alt={data.totp.uri} layout="fil
```

The `challenge` and `verify` steps are separated when using Phone factors as the user will need time to receive and input the code obtained from the SMS in challenge.

Parameters

`params` REQUIRED MFAEnrollParams

Details

Return Type

Promise<Union: expand to see options>

Details

Enroll a time-based, one-time password (TOTP) factor | Enroll a Phone Factor

```
const { data, error } = await supabase.auth.mfa.enroll({
  factorType: 'totp',
  friendlyName: 'your_friendly_name'
})

// Use the id to create a challenge.
// The challenge can be verified by entering the code generated from the auther
// The code will be generated upon scanning the qr_code or entering the secret
const { id, type, totp: { qr_code, secret, uri }, friendly_name } = data
const challenge = await supabase.auth.mfa.challenge({ factorId: id });
```

Response

Create a challenge

Prepares a challenge used to verify that a user has access to a MFA factor.

An enrolled factor is required before creating a challenge.

To verify a challenge, see `mfa.verify()`.

A phone factor sends a code to the user upon challenge. The channel defaults to `sms` unless otherwise specified.

Parameters

`params` REQUIRED MFAChallengeParams

Details

Return Type

Promise<Union: expand to see options>

Details

Create a challenge for a factor | Create a challenge for a phone factor | Create a challenge for a phone factor (WhatsApp)

```
1 const { data, error } = await supabase.auth.mfa.challenge({
2   factorId: '34e770dd-9ff9-416c-87fa-43b31d7ef225'
3 })
```

Response

Verify a challenge

Verifies a code against a challenge. The verification code is provided by the user by entering a code seen in their authenticator app.

To verify a challenge, please create a challenge first.

Parameters

`params` REQUIRED MFAVerifyParams

Details

Return Type

Promise<Union: expand to see options>

Details

Verify a challenge for a factor

```
const { data, error } = await supabase.auth.mfa.verify({
  factorId: '34e770dd-9ff9-416c-87fa-43b31d7ef225',
  challengeId: '4034ae6f-a8ce-4fb5-8ee5-69a5863a7c15',
  code: '123456'
})
```

Response

Create and verify a challenge

Helper method which creates a challenge and immediately uses the given code to verify against it thereafter. The verification code is provided by the user by entering a code seen in their authenticator app.

Intended for use with only TOTP factors.

An enrolled factor is required before invoking `challengeAndVerify()`.

Executes `mfa.challenge()` and `mfa.verify()` in a single step.

Parameters

`params` REQUIRED MFAChallengeAndVerifyParams

Details

Return Type

Promise<Union: expand to see options>

Details

Create and verify a challenge for a factor

```
const { data, error } = await supabase.auth.mfa.challengeAndVerify({
  factorId: '34e770dd-9ff9-416c-87fa-43b31d7ef225',
  code: '123456'
})
```

Response

Unenroll a factor

Unenroll removes a MFA factor. A user has to have an `aal2` authenticator level in order to unenroll a `verified` factor.

Parameters

`params` REQUIRED MFAUnenrollParams

Details

Return Type

Promise<Union: expand to see options>

Details

Unenroll a factor

```
const { data, error } = await supabase.auth.mfa.unenroll({
  factorId: '34e770dd-9ff9-416c-87fa-43b31d7ef225',
})
```

Response

Get Authenticator Assurance Level

Returns the Authenticator Assurance Level (AAL) for the active session.

> Authenticator Assurance Level (AAL) is the measure of the strength of an authentication mechanism.
>
> In Supabase, having an AAL of `aal1` refers to having the 1st factor of authentication such as an email and password or OAuth sign-in while `aal2` refers to the 2nd factor of authentication such as a time-based, one-time-password (TOTP) or Phone factor.
>
> If the user has a verified factor, the `nextLevel` field will return `aal2`, else, it will return `aal1`.

Return Type

Promise<Union: expand to see options>

Details

Get the AAL details of a session

```
const { data, error } = await supabase.auth.mfa.getAuthenticatorAssuranceLevel(
const { currentLevel, nextLevel, currentAuthenticationMethods } = data
```

Response

Auth Admin

Any method under the `supabase.auth.admin` namespace requires a `service_role` key.

These methods are considered admin methods and should be called on a trusted server. Never expose your `service_role` key in the browser.

Create server-side auth client

```
import { createClient } from '@supabase/supabase-js'

const supabase = createClient(supabase_url, service_role_key, {
  auth: {
    autoRefreshToken: false,
    persistSession: false
  }
})

// Access auth admin api
const adminAuthClient = supabase.auth.admin
```

Retrieve a user

Get user by id.

Fetches the user object from the database based on the user's id.

The `getUserById()` method requires the user's id which maps to the `auth.users.id` column.

Parameters

`uid` REQUIRED string

The user's unique identifier

This function should only be called on a server. Never expose your `service_role` key in the browser.

Return Type

Promise<Union: expand to see options>

Details

Fetch the user object using the access_token jwt

```
1 const { data, error } = await supabase.auth.admin.getUserById(1)
```

Response

List all users

Get a list of users.

Defaults to return 50 users per page.

Parameters

`params` `Optional` PageParams

An object which supports `page` and `perPage` as numbers, to alter the paginated results.

Details

Return Type

Promise<Union: expand to see options>

Details

Get a page of users | Paginated list of users

```
1 const { data: { users }, error } = await supabase.auth.admin.listUsers()
```

Create a user

Creates a new user. This function should only be called on a server. Never expose your `service_role` key in the browser.

> To confirm the user's email address or phone number, set `email_confirm` or `phone_confirm` to true. Both arguments default to false.

`createUser()` will not send a confirmation email to the user. You can use `inviteUserByEmail()` if you want to send them an email invite instead.

If you are sure that the created user's email or phone number is legitimate and verified, you can set the `email_confirm` or `phone_confirm` param to `true`.

Parameters

`attributes` REQUIRED AdminUserAttributes

Details

Return Type

Promise<Union: expand to see options>

Details

With custom user metadata | Auto-confirm the user's email | Auto-confirm the user's phone number

```
const { data, error } = await supabase.auth.admin.createUser({
  email: 'user@email.com',
  password: 'password',
  user_metadata: { name: 'Yoda' }
})
```

Response

Delete a user

Delete a user. Requires a `service_role` key.

> The `deleteUser()` method requires the user's ID, which maps to the `auth.users.id` column.

Parameters

`id` REQUIRED string

The user id you want to remove.

`shouldSoftDelete` REQUIRED boolean

If true, then the user will be soft-deleted (setting `deleted_at` to the current timestamp and disabling their account while preserving their data) from the auth schema. Defaults to false for backward compatibility.

This function should only be called on a server. Never expose your `service_role` key in the browser.

Return Type

Promise<Union: expand to see options>

Details

Removes a user

```
const { data, error } = await supabase.auth.admin.deleteUser(
  '715ed5db-f090-4b8c-a067-640ecee36aa0'
)
```

Response

Send an email invite link

Sends an invite link to an email address.

Sends an invite link to the user's email address.

The `inviteUserByEmail()` method is typically used by administrators to invite users to join the application.

Note that PKCE is not supported when using `inviteUserByEmail`. This is because the browser initiating the invite is often different from the browser accepting the invite which makes it difficult to provide the security guarantees required of the PKCE flow.

Parameters

`email` REQUIRED string

The email address of the user.

`options` REQUIRED object

Additional options to be included when inviting.

Details

Return Type

Promise<Union: expand to see options>

Details

Invite a user

```
1 const { data, error } = await supabase.auth.admin.inviteUserByEmail('email@exam
```

Response

Generate an email link

Generates email links and OTPs to be sent via a custom email provider.

The following types can be passed into `generateLink()`: `signup`, `magiclink`, `invite`, `recovery`, `email_change_current`, `email_change_new`, `phone_change`.

`generateLink()` only generates the email link for `email_change_email` if the **Secure email change** is enabled in your project's email auth provider settings.

`generateLink()` handles the creation of the user for `signup`, `invite` and `magiclink`.

Parameters

`params` REQUIRED GenerateLinkParams

Details

Return Type

Promise<Union: expand to see options>

Details

Generate a signup link | Generate an invite link | Generate a magic link | Generate a recovery link | Generate links to change current email address

```
const { data, error } = await supabase.auth.admin.generateLink({
  type: 'signup',
  email: 'email@example.com',
  password: 'secret'
})
```

> Response

Update a user

Updates the user data.

Parameters

`uid` REQUIRED string

`attributes` REQUIRED AdminUserAttributes

The data you want to update.

This function should only be called on a server. Never expose your `service_role` key in the browser.

Details

Return Type

Promise<Union: expand to see options>

Details

Updates a user's email | Updates a user's password | Updates a user's metadata | Updates a user's app_metadata | Confirms a user's email address | Confirms a user's phone number

```
const { data: user, error } = await supabase.auth.admin.updateUserById(
  '11111111-1111-1111-1111-111111111111',
  { email: 'new@email.com' }
)
```

Response

Delete a factor for a user

Deletes a factor on a user. This will log the user out of all active sessions if the deleted factor was verified.

Parameters

`params` REQUIRED AuthMFAAdminDeleteFactorParams

Details

Return Type

Promise<Union: expand to see options>

Details

Delete a factor for a user

```
const { data, error } = await supabase.auth.admin.mfa.deleteFactor({
  id: '34e770dd-9ff9-416c-87fa-43b31d7ef225',
  userId: 'a89baba7-b1b7-440f-b4bb-91026967f66b',
})
```

Response

Invokes a Supabase Edge Function.

Invokes a function

Invoke a Supabase Edge Function.

Requires an Authorization header.

Invoke params generally match the Fetch API spec.

When you pass in a body to your function, we automatically attach the Content-Type header for `Blob`, `ArrayBuffer`, `File`, `FormData` and `String`. If it doesn't match any of these types we assume the payload is `json`, serialize it and attach the `Content-Type` header as `application/json`. You can override this behavior by passing in a `Content-Type` header of your own.

Responses are automatically parsed as `json`, `blob` and `form-data` depending on the `Content-Type` header sent by your function. Responses are parsed as `text` by default.

Parameters

`functionName` REQUIRED string

The name of the Function to invoke.

`options` REQUIRED FunctionInvokeOptions

Options for invoking the Function.

Details

Return Type

Promise<Union: expand to see options>

Basic invocation | Error handling | Passing custom headers | Calling with DELETE HTTP verb | Invoking a Function in the UsEast1 region | Calling with GET HTTP verb

```
const { data, error } = await supabase.functions.invoke('hello', {
  body: { foo: 'bar' }
})
```

Subscribe to channel

Creates an event handler that listens to changes.

By default, Broadcast and Presence are enabled for all projects.

By default, listening to database changes is disabled for new projects due to database performance and security concerns. You can turn it on by managing Realtime's

replication.

You can receive the "previous" data for updates and deletes by setting the table's `REPLICA IDENTITY` to `FULL` (e.g., `ALTER TABLE your_table REPLICA IDENTITY FULL;`).

Row level security is not applied to delete statements. When RLS is enabled and replica identity is set to full, only the primary key is sent to clients.

Parameters

`type` REQUIRED Union: expand to see options

Details

`filter` REQUIRED Union: expand to see options

Details

`callback` REQUIRED function

Details

Listen to broadcast messages | Listen to presence sync | Listen to presence join | Listen to presence leave | Listen to all database changes | Listen to a specific table | Listen to inserts | Listen to updates | Listen to deletes | Listen to multiple events | Listen to row level changes

```
const channel = supabase.channel("room1")

channel.on("broadcast", { event: "cursor-pos" }, (payload) => {
  console.log("Cursor position received!", payload);
}).subscribe((status) => {
  if (status === "SUBSCRIBED") {
    channel.send({
      type: "broadcast",
      event: "cursor-pos",
      payload: { x: Math.random(), y: Math.random() },
    });
  }
});
```

Unsubscribe from a channel

Unsubscribes and removes Realtime channel from Realtime client.

> Removing a channel is a great way to maintain the performance of your project's Realtime service as well as your database if you're listening to Postgres changes. Supabase will automatically handle cleanup 30 seconds after a client is disconnected, but unused channels may cause degradation as more clients are simultaneously subscribed.

Parameters

`channel` REQUIRED @supabase/realtime-js.RealtimeChannel

The name of the Realtime channel.

Return Type

Promise<Union: expand to see options>

Details

Removes a channel

```
supabase.removeChannel(myChannel)
```

Unsubscribe from all channels

Unsubscribes and removes all Realtime channels from Realtime client.

> Removing channels is a great way to maintain the performance of your project's Realtime service as well as your database if you're listening to Postgres changes. Supabase will automatically handle cleanup 30 seconds after a client is disconnected, but unused channels may cause degradation as more clients are simultaneously subscribed.

Return Type

Promise<Array<Union: expand to see options>>

Details

Remove all channels

```
supabase.removeAllChannels()
```

Retrieve all channels

Returns all Realtime channels.

Return Type

Array<@supabase/realtime-js.RealtimeChannel>

Get all channels

```
1 const channels = supabase.getChannels()
```

Broadcast a message

Sends a message into the channel.

Broadcast a message to all connected clients to a channel.

> When using REST you don't need to subscribe to the channel
>
> REST calls are only available from 2.37.0 onwards

Parameters

`args` REQUIRED object

Arguments to send to channel

Details

`opts` REQUIRED { [key: string]: any }

Options to be used during the send process

Return Type

Promise<Union: expand to see options>

Details

Send a message via websocket | Send a message via REST

```
supabase
  .channel('room1')
  .subscribe((status) => {
    if (status === 'SUBSCRIBED') {
      channel.send({
        type: 'broadcast',
        event: 'cursor-pos',
        payload: { x: Math.random(), y: Math.random() },
      })
    }
  })
```

Response

Create a bucket

Creates a new Storage bucket

RLS policy permissions required:

`buckets` table permissions: `insert`

`objects` table permissions: none

Refer to the Storage guide on how access control works

Parameters

id REQUIRED string

A unique identifier for the bucket you are creating.

options REQUIRED object

Details

Return Type

Promise<Union: expand to see options>

Details

Create bucket

```
const { data, error } = await supabase
  .storage
  .createBucket('avatars', {
    public: false,
    allowedMimeTypes: ['image/png'],
    fileSizeLimit: 1024
  })
```

Response

Retrieve a bucket

Retrieves the details of an existing Storage bucket.

RLS policy permissions required:

`buckets` table permissions: `select`

`objects` table permissions: none

Refer to the Storage guide on how access control works

Parameters

`id` REQUIRED string

The unique identifier of the bucket you would like to retrieve.

Return Type

Promise<Union: expand to see options>

Details

Get bucket

```
const { data, error } = await supabase
  .storage
  .getBucket('avatars')
```

Response

List all buckets

Retrieves the details of all Storage buckets within an existing project.

RLS policy permissions required:

`buckets` table permissions: `select`

`objects` table permissions: none

Refer to the Storage guide on how access control works

Return Type

Promise<Union: expand to see options>

Details

List buckets

```
const { data, error } = await supabase
  .storage
  .listBuckets()
```

Response

Update a bucket

Updates a Storage bucket

RLS policy permissions required:

- `buckets` table permissions: `select` and `update`
- `objects` table permissions: none

Refer to the Storage guide on how access control works

Parameters

`id` REQUIRED string

A unique identifier for the bucket you are updating.

`options` REQUIRED object

Details

Return Type

Promise<Union: expand to see options>

Details

Update bucket

```
const { data, error } = await supabase
  .storage
  .updateBucket('avatars', {
    public: false,
    allowedMimeTypes: ['image/png'],
    fileSizeLimit: 1024
  })
```

Response

Delete a bucket

Deletes an existing bucket. A bucket can't be deleted with existing objects inside it. You must first `empty()` the bucket.

RLS policy permissions required:

- `buckets` table permissions: `select` and `delete`
- `objects` table permissions: none

Refer to the Storage guide on how access control works

Parameters

`id` REQUIRED string

The unique identifier of the bucket you would like to delete.

Return Type

Promise<Union: expand to see options>

Details

Delete bucket

```
const { data, error } = await supabase
  .storage
  .deleteBucket('avatars')
```

Response

Empty a bucket

Removes all objects inside a single bucket.

RLS policy permissions required:

`buckets` table permissions: `select`

`objects` table permissions: `select` and `delete`

Refer to the Storage guide on how access control works

Parameters

`id` REQUIRED string

The unique identifier of the bucket you would like to empty.

Return Type

Promise<Union: expand to see options>

Details

Empty bucket

```
const { data, error } = await supabase
  .storage
  .emptyBucket('avatars')
```

Response

Upload a file

Uploads a file to an existing bucket.

RLS policy permissions required:

- `buckets` table permissions: none
- `objects` table permissions: only `insert` when you are uploading new files and `select`, `insert` and `update` when you are upserting files

Refer to the Storage guide on how access control works

For React Native, using either `Blob`, `File` or `FormData` does not work as intended. Upload file using `ArrayBuffer` from base64 file data instead, see example below.

Parameters

`path` REQUIRED string

The file path, including the file name. Should be of the format `folder/subfolder/filename.png`. The bucket must already exist before attempting to upload.

`fileBody` REQUIRED FileBody

The body of the file to be stored in the bucket.

`fileOptions` Optional FileOptions

Details

Return Type

Promise<Union: expand to see options>

Details

Upload file Upload file using `ArrayBuffer` from base64 file data

```
const avatarFile = event.target.files[0]
const { data, error } = await supabase
  .storage
  .from('avatars')
  .upload('public/avatar1.png', avatarFile, {
    cacheControl: '3600',
    upsert: false
  })
```

Response

Download a file

Downloads a file from a private bucket. For public buckets, make a request to the URL returned from `getPublicUrl` instead.

RLS policy permissions required:

- `buckets` table permissions: none
- `objects` table permissions: `select`

Refer to the Storage guide on how access control works

Parameters

`path` REQUIRED string

The full path and file name of the file to be downloaded. For example `folder/image.png`.

`options` `Optional` object

Details

Return Type

Promise<Union: expand to see options>

Details

Download file | Download file with transformations

```
const { data, error } = await supabase
  .storage
  .from('avatars')
  .download('folder/avatar1.png')
```

Response

List all files in a bucket

Lists all the files within a bucket.

RLS policy permissions required:

`buckets` table permissions: none

`objects` table permissions: `select`

Refer to the Storage guide on how access control works

Parameters

`path` `Optional` string

The folder path.

`options` `Optional` SearchOptions

Details

`parameters` `Optional` FetchParameters

Details

Return Type

Promise<Union: expand to see options>

Details

List files in a bucket | Search files in a bucket

```
const { data, error } = await supabase
  .storage
  .from('avatars')
  .list('folder', {
    limit: 100,
    offset: 0,
    sortBy: { column: 'name', order: 'asc' },
  })
```

Response

Replace an existing file

Replaces an existing file at the specified path with a new one.

RLS policy permissions required:

`buckets` table permissions: none

`objects` table permissions: `update` and `select`

Refer to the Storage guide on how access control works

For React Native, using either `Blob`, `File` or `FormData` does not work as intended. Update file using `ArrayBuffer` from base64 file data instead, see example below.

Parameters

`path` REQUIRED string

The relative file path. Should be of the format `folder/subfolder/filename.png`. The bucket must already exist before attempting to update.

`fileBody` REQUIRED Union: expand to see options

The body of the file to be stored in the bucket.

Details

`fileOptions` Optional FileOptions

Details

Return Type

Promise<Union: expand to see options>

Details

Update file Update file using `ArrayBuffer` from base64 file data

```
const avatarFile = event.target.files[0]
const { data, error } = await supabase
  .storage
  .from('avatars')
  .update('public/avatar1.png', avatarFile, {
    cacheControl: '3600',
    upsert: true
  })
```

Response

Move an existing file

Moves an existing file to a new path in the same bucket.

RLS policy permissions required:

- `buckets` table permissions: none
- `objects` table permissions: `update` and `select`

Refer to the Storage guide on how access control works

Parameters

`fromPath` REQUIRED string

The original file path, including the current file name. For example `folder/image.png`.

`toPath` REQUIRED string

The new file path, including the new file name. For example `folder/image-new.png`.

Return Type

Promise<Union: expand to see options>

Details

Move file

```
const { data, error } = await supabase
  .storage
  .from('avatars')
  .move('public/avatar1.png', 'private/avatar2.png')
```

Response

Copy an existing file

Copies an existing file to a new path in the same bucket.

RLS policy permissions required:

- `buckets` table permissions: none
- `objects` table permissions: `insert` and `select`

Refer to the Storage guide on how access control works

Parameters

`fromPath` REQUIRED string

The original file path, including the current file name. For example `folder/image.png`.

`toPath` REQUIRED string

The new file path, including the new file name. For example `folder/image-copy.png`.

Return Type

Promise<Union: expand to see options>

Details

Copy file

```
const { data, error } = await supabase
  .storage
  .from('avatars')
  .copy('public/avatar1.png', 'private/avatar2.png')
```

Response

Delete files in a bucket

Deletes files within the same bucket

RLS policy permissions required:

- `buckets` table permissions: none
- `objects` table permissions: `delete` and `select`

Refer to the Storage guide on how access control works

Parameters

paths REQUIRED Array<string>

An array of files to delete, including the path and file name. For example [`'folder/image.png'`].

Return Type

Promise<Union: expand to see options>

Details

Delete file

```
const { data, error } = await supabase
  .storage
  .from('avatars')
  .remove(['folder/avatar1.png'])
```

Response

Create a signed URL

Creates a signed URL. Use a signed URL to share a file for a fixed amount of time.

RLS policy permissions required:

`buckets` table permissions: none

`objects` table permissions: `select`

Refer to the Storage guide on how access control works

Parameters

`path` REQUIRED string

The file path, including the current file name. For example `folder/image.png`.

`expiresIn` REQUIRED number

The number of seconds until the signed URL expires. For example, `60` for a URL which is valid for one minute.

`options` Optional object

Details

Return Type

Promise<Union: expand to see options>

Details

Create Signed URL — Create a signed URL for an asset with transformations

Create a signed URL which triggers the download of the asset

```
const { data, error } = await supabase
  .storage
  .from('avatars')
  .createSignedUrl('folder/avatar1.png', 60)
```

Response

Create signed URLs

Creates multiple signed URLs. Use a signed URL to share a file for a fixed amount of time.

RLS policy permissions required:

`buckets` table permissions: none

`objects` table permissions: `select`

Refer to the Storage guide on how access control works

Parameters

`paths` REQUIRED Array<string>

The file paths to be downloaded, including the current file names. For example `['folder/image.png', 'folder2/image2.png']`.

`expiresIn` REQUIRED number

The number of seconds until the signed URLs expire. For example, `60` for URLs which are valid for one minute.

`options` Optional object

Details

Return Type

Promise<Union: expand to see options>

Details

Create Signed URLs

```
const { data, error } = await supabase
  .storage
  .from('avatars')
  .createSignedUrls(['folder/avatar1.png', 'folder/avatar2.png'], 60)
```

Response

Create signed upload URL

Creates a signed upload URL. Signed upload URLs can be used to upload files to the bucket without further authentication. They are valid for 2 hours.

RLS policy permissions required:

- `buckets` table permissions: none
- `objects` table permissions: `insert`

Refer to the Storage guide on how access control works

Parameters

`path` REQUIRED string

The file path, including the current file name. For example `folder/image.png`.

Return Type

Promise<Union: expand to see options>

Details

Create Signed Upload URL

```
const { data, error } = await supabase
  .storage
  .from('avatars')
  .createSignedUploadUrl('folder/cat.jpg')
```

Response

Upload to a signed URL

Upload a file with a token generated from `createSignedUploadUrl`.

RLS policy permissions required:

- `buckets` table permissions: none
- `objects` table permissions: none

Refer to the Storage guide on how access control works

Parameters

`path` REQUIRED string

The file path, including the file name. Should be of the format `folder/subfolder/filename.png`. The bucket must already exist before attempting to upload.

`token` REQUIRED string

The token generated from `createSignedUploadUrl`

`fileBody` REQUIRED FileBody

The body of the file to be stored in the bucket.

`fileOptions` Optional FileOptions

Details

Return Type

Promise<Union: expand to see options>

Details

Upload to a signed URL

```
const { data, error } = await supabase
  .storage
  .from('avatars')
  .uploadToSignedUrl('folder/cat.jpg', 'token-from-createSignedUploadUrl', file
```

Response

Retrieve public URL

A simple convenience function to get the URL for an asset in a public bucket. If you do not want to use this function, you can construct the public URL by concatenating the bucket URL with the path to the asset. This function does not verify if the bucket is public. If a public URL is created for a bucket which is not public, you will not be able to download the asset.

> The bucket needs to be set to public, either via updateBucket() or by going to Storage on supabase.com/dashboard, clicking the overflow menu on a bucket and choosing "Make public"
>
> RLS policy permissions required:
>
> - `buckets` table permissions: none
> - `objects` table permissions: none
>
> Refer to the Storage guide on how access control works

Parameters

`path` REQUIRED string

The path and name of the file to generate the public URL for. For example `folder/image.png`.

`options` Optional object

Details

Return Type

object

Details

Returns the URL for an asset in a public bucket | Returns the URL for an asset in a public bucket with transformations

Returns the URL which triggers the download of an asset in a public bucket

```
const { data } = supabase
  .storage
  .from('public-bucket')
  .getPublicUrl('folder/avatar1.png')
```

Response

Release Notes

Supabase.js v2 release notes.

Install the latest version of @supabase/supabase-js

Terminal

```
npm install @supabase/supabase-js
```

Explicit constructor options

All client specific options within the constructor are keyed to the library.

See PR:

```
const supabase = createClient(apiURL, apiKey, {
  db: {
    schema: 'public',
```

```
  },
  auth: {
    storage: AsyncStorage,
    autoRefreshToken: true,
    persistSession: true,
    detectSessionInUrl: true,
  },
  realtime: {
    channels,
    endpoint,
  },
  global: {
    fetch: customFetch,
    headers: DEFAULT_HEADERS,
  },
})
```

TypeScript support

The libraries now support typescript.

v1.0

```
// previously definitions were injected in the `from()` method
supabase.from<Definitions['Message']>('messages').select('\*')
```

v2.0

```
import type { Database } from './DatabaseDefinitions'

// definitions are injected in `createClient()`
const supabase = createClient<Database>(SUPABASE_URL, ANON_KEY)

const { data } = await supabase.from('messages').select().match({ id: 1 })
```

Types can be generated via the CLI:

Terminal

```
supabase start
supabase gen types typescript --local > DatabaseDefinitions.ts
```

Data operations return minimal

`.insert()` / `.upsert()` / `.update()` / `.delete()` don't return rows by default: PR.

Previously, these methods return inserted/updated/deleted rows by default (which caused some confusion), and you can opt to not return it by specifying `returning: 'minimal'`. Now the default behavior is to not return rows. To return inserted/updated/deleted rows, add a `.select()` call at the end, e.g.:

```
const { data, error } = await supabase
    .from('my_table')
    .delete()
    .eq('id', 1)
    .select()
```

New ordering defaults

`.order()` now defaults to Postgres's default: PR.

Previously `nullsFirst` defaults to `false`, meaning `null`s are ordered last. This is bad for performance if e.g. the column uses an index with `NULLS FIRST` (which is the default direction for indexes).

Cookies and localstorage namespace

Storage key name in the Auth library has changed to include project reference which means that existing websites that had their JWT expiry set to a longer time could find their users logged out with this upgrade.

```
const defaultStorageKey = `sb-${new URL(this.authUrl).hostname.split('.')[0]}-a
```

New Auth Types

Typescript typings have been reworked. `Session` interface now guarantees that it will always have an `access_token`, `refresh_token` and `user`

./types.ts

```
interface Session {
  provider_token?: string | null
  access_token: string
  expires_in?: number
  expires_at?: number
  refresh_token: string
  token_type: string
  user: User
}
```

New Auth methods

We're removing the `signIn()` method in favor of more explicit function signatures: `signInWithPassword()`, `signInWithOtp()`, and `signInWithOAuth()`.

v1.0

```
const { data } = await supabase.auth.signIn({
  email: 'hello@example',
  password: 'pass',
})
```

v2.0

```
const { data } = await supabase.auth.signInWithPassword({
  email: 'hello@example',
  password: 'pass',
})
```

New Realtime methods

There is a new `channel()` method in the Realtime library, which will be used for our Multiplayer updates.

We will deprecate the `.from().on().subscribe()` method previously used for listening to postgres changes.

```
supabase
  .channel('any_string_you_want')
  .on('presence', { event: 'track' }, (payload) => {
    console.log(payload)
  })
  .subscribe()

supabase
  .channel('any_string_you_want')
  .on(
    'postgres_changes',
    {
      event: 'INSERT',
      schema: 'public',
      table: 'movies',
    },
    (payload) => {
      console.log(payload)
    }
  )
  .subscribe()
```

Deprecated setAuth()

Deprecated and removed `setAuth()` . To set a custom `access_token` jwt instead, pass the custom header into the `createClient()` method provided: (PR)

All changes

`supabase-js`

`shouldThrowOnError` has been removed until all the client libraries support this option (PR).

`postgrest-js`

TypeScript typings have been reworked PR

Use `undefined` instead of `null` for function params, types, etc. (https://github.com/supabase/postgrest-js/pull/278)

Some features are now obsolete: (https://github.com/supabase/postgrest-js/pull/275)

filter shorthands (e.g. `cs` vs. `contains`)

`body` in response (vs. `data`)

`upsert`ing through the `.insert()` method

`auth` method on `PostgrestClient`

client-level `throwOnError`

`gotrue-js`

`supabase-js` client allows passing a `storageKey` param which will allow the user to set the key used in local storage for storing the session. By default, this will be namespace-d with the supabase project ref. (PR)

`signIn` method is now split into `signInWithPassword`, `signInWithOtp`, `signInWithOAuth` (PR)

Deprecated and removed `session()`, `user()` in favour of using `getSession()` instead. `getSession()` will always return a valid session if a user is already logged in, meaning no more random logouts. (PR)

Deprecated and removed setting for `multitab` support because `getSession()` and gotrue's reuse interval setting takes care of session management across multiple tabs (PR)

No more throwing of random errors, gotrue-js v2 always returns a custom error type: (PR)

`AuthSessionMissingError`

Indicates that a session is expected but missing

`AuthNoCookieError`

Indicates that a cookie is expected but missing

`AuthInvalidCredentialsError`

Indicates that the incorrect credentials were passed

Renamed the `api` namespace to `admin`, the `admin` namespace will only contain methods that should only be used in a trusted server-side environment with the service role key

Moved `resetPasswordForEmail`, `getUser` and `updateUser` to the `GoTrueClient` which means they will be accessible from the `supabase.auth` namespace in `supabase-`

`js` instead of having to do `supabase.auth.api` to access them

Removed `sendMobileOTP` , `sendMagicLinkEmail` in favor of `signInWithOtp`

Removed `signInWithEmail` , `signInWithPhone` in favor of `signInWithPassword`

Removed `signUpWithEmail` , `signUpWithPhone` in favor of `signUp`

Replaced `update` with `updateUser`

`storage-js`

Return types are more strict. Functions types used to indicate that the data returned could be null even if there was no error. We now make use of union types which only mark the data as null if there is an error and vice versa. (PR)

The `upload` and `update` function returns the path of the object uploaded as the `path` parameter. Previously the returned value had the bucket name prepended to the path which made it harder to pass the value on to other storage-js methods since all methods take the bucket name and path separately. We also chose to call the returned value `path` instead of `Key` (PR)

`getPublicURL` only returns the public URL inside the data object. This keeps it consistent with our other methods of returning only within the data object. No error is returned since this method cannot does not throw an error (PR)

signed urls are returned as `signedUrl` instead of `signedURL` in both `createSignedUrl` and `createSignedUrls` (PR)

Encodes URLs returned by `createSignedUrl` , `createSignedUrls` and `getPublicUrl` (PR)

`createsignedUrl` used to return a url directly and and within the data object. This was inconsistent. Now we always return values only inside the data object across all methods. (PR)

`createBucket` returns a data object instead of the name of the bucket directly. (PR)

Fixed types for metadata (PR)

Better error types make it easier to track down what went wrong quicker.

`SupabaseStorageClient` is no longer exported. Use `StorageClient` instead. (PR).

`realtime-js`

`RealtimeSubscription` class no longer exists and replaced by `RealtimeChannel`.

`RealtimeClient`'s `disconnect` method now returns type of `void`. It used to return type of `Promise<{ error: Error | null; data: boolean }>`.

Removed `removeAllSubscriptions` and `removeSubscription` methods from `SupabaseClient` class.

Removed `SupabaseRealtimeClient` class.

Removed `SupabaseQueryBuilder` class.

Removed `SupabaseEventTypes` type.

> Thinking about renaming this to something like `RealtimePostgresChangeEvents` and moving it to `realtime-js` v2.

Removed `.from('table').on('INSERT', () => {}).subscribe()` in favor of new Realtime client API.

`functions-js`

supabase-js v1 only threw an error if the fetch call itself threw an error (network errors, etc) and not if the function returned HTTP errors like 400s or 500s. We have changed this behaviour to return an error if your function throws an error.

We have introduced new error types to distinguish between different kinds of errors. A `FunctionsHttpError` error is returned if your function throws an error, `FunctionsRelayError` if the Supabase Relay has an error processing your function and `FunctionsFetchError` if there is a network error in calling your function.

The correct content-type headers are automatically attached when sending the request if you don't pass in a `Content-Type` header and pass in an argument to your function. We automatically attach the content type for `Blob`, `ArrayBuffer`, `File`, `FormData`, `String`. If it doesn't match any of these we assume the payload is `json`, we serialise the payload as JSON and attach the content type as `application/json`.

`responseType` does not need to be explicitly passed in. We parse the response based on the `Content-Type` response header sent by the function. We support parsing the responses as `text`, `json`, `blob`, `form-data` and are parsed as `text` by default.

Need some help? Contact support

Latest product updates? See Changelog

Something's not right? Check system status

—

Contributing

Author Styleguide

Open Source

SupaSquad

Privacy Settings