Understanding API keys

Supabase gives you fine-grained control over which application components are allowed to access your project through API keys.

API keys provide the first layer of authentication for data access. Auth then builds upon that. This chart covers the differences:

Responsibility	Question	Answer
API keys	**What** is accessing the project?	Web page, mobile app, server, Edge Function...
Supabase Auth	**Who** is accessing the project?	Monica, Jian Yang, Gavin, Dinesh, Laurie, Fiona...

Overview

An API key authenticates an application component to give it access to Supabase services. An application component might be a web page, a mobile app, or a server. The API key *does not* distinguish between users, only between applications.

There are 4 types of API keys that can be used with Supabase:

Type	Format	Privileges	Availability	Use
Publishable key	`sb_publishable_...`	Low	Platform	Safe to expose online: web page, mobile or desktop app, GitHub actions, CLIs, source code.
Secret keys	`sb_secret_...`	Elevated	Platform	**Only use in backend components of your app:** servers, already secured APIs (admin panels), Edge Functions, microservices, etc. They provide *full*

Type	Format	Privileges	Availability	Use
				access to your project's data, bypassing Row Level Security.
`anon`	JWT (long lived)	Low	Platform, CLI	Exactly like the publishable key.
`service_role`	JWT (long lived)	Elevated	Platform, CLI	Exactly like secret keys.

`anon` and `service_role` keys are based on the project's JWT secret. They are generated when your project is created and can only be changed when you rotate the JWT secret. This can cause significant issues in production applications. Use the publishable and secret keys instead.

Changes to API keys

Supabase is changing the way keys work to improve project security and developer experience. You can read the full announcement, but in the transition period, you can use both the current `anon` and `service_role` keys and the new publishable key with the form `sb_publishable_xxx` which will replace the older keys.

Where to find keys

You can find API keys in a couple of different places.

In most cases, you can get the correct key from the Project's **Connect** dialog, but if you want a specific key, you can find all keys in the API Keys section of a Project's Settings page:

For legacy keys, copy the `anon` key for client-side operations and the `service_role` key for server-side operations from the **Legacy API Keys** tab.

For new keys, open the **API Keys** tab, if you don't have a publishable key already, click **Create new API Keys**, and copy the value from the **Publishable key** section.

`anon` and publishable keys

The `anon` and publishable keys secure the public components of your application. Public components run in environments where it is impossible to secure any secrets. These include:

Web pages, where the key is bundled in source code.

- Mobile or desktop applications, where the key is bundled inside the compiled packages or executables.
- CLI, scripts, tools, or other pre-built executables.
- Other publicly available APIs that return the key without prior additional authorization.

These environments are always considered public because anyone can retrieve the key from the source code or build artifacts. Obfuscation can increase the difficulty, but never eliminate the possibility. (In general, obfuscation, Turing test challenges, and specialized knowledge do not count as authorization for the purpose of securing secrets.)

Using the `anon` or publishable key does not mean that your user is anonymous. (Thinking of both these keys as publishable rather than `anon` makes the mental model clearer.)

Your application can be authenticated with the publishable key, while your user is authenticated (via Supabase Auth) with their personal JWT:

Key	**User logged in via Supabase Auth**	**Postgres role used for RLS, etc.**
Publishable key	No	`anon`
`anon`	No	`anon`
Publishable key	Yes	`authenticated`
`anon`	Yes	`authenticated`

Protection

These keys provide first-layer protection to your project's data, performance and bill, such as:

- Providing basic Denial-of-Service protection, by requiring a minimal threshold of knowledge.
- Protecting your bill by ignoring bots, scrapers, automated vulnerability scanners and other well meaning or random Internet activity.

Security considerations

The publishable and `anon` keys are not intended to protect from the following, since key retrieval is always possible from a public component:

- Static or dynamic code analysis and reverse engineering attempts.
- Use of the Network inspector in the browser.
- Cross-site request forgery, cross-site scripting, phishing attacks.
- Man-in-the-middle attacks.

When using the publishable or `anon` key, access to your project's data is guarded by Postgres via the built-in `anon` and `authenticated` roles. For full protection make sure:

- You have enabled Row Level Security on all tables.
- You regularly review your Row Level Security policies for permissions granted to the `anon` and `authenticated` roles.
- You do not modify the role's attributes without understanding the changes you are making.

Your project's Security Advisor constantly checks for common security problems with the built-in Postgres roles. Make sure you carefully review each finding before dismissing it.

`service_role` and secret keys

Unlike the `anon` and publishable key, the `service_role` and secret keys allow elevated access to your project's data. It is meant to be used only in secure, developer-controlled components of your application, such as:

- Servers that implement prior authorization themselves, such as Edge Functions, microservices, traditional or specialized web servers.
- Periodic jobs, queue processors, topic subscribers.
- Admin and back-office tools, with prior authorization checks only.
- Data processing pipelines, such as for analytics, reports, backups, or database synchronization.

Never expose your `service_role` and secret keys publicly. Your data is at risk. **Do not:**

- Add in web pages, public documents, source code, bundle in executables or packages for mobile, desktop or CLI apps.

- Send over chat applications, email or SMS to your peers.
- Never use in a browser, even on `localhost`!
- Do not pass in URLs or query params, as these are often logged.
- Be careful passing them in request headers without prior log sanitization.
- Take extra care logging even potentially **invalid API keys**. Simple typos might reveal the real key in the future.
- Reveal, copy, use or manipulate on hardware devices without full disk encryption and which you do not directly own or control (such as public computers, friend's laptop, etc.)

Ensure you handle them with care and using secure coding practices.

Secret keys and the `service_role` JWT-based API key authorize access to your project's data via the built-in `service_role` Postgres role. By design, this role has full access to your project's data. It also uses the `BYPASSRLS` attribute, skipping any and all Row Level Security policies you attach.

The secret key is an improvement over the old JWT-based `service_role` key, and we recommend using it where possible. It adds more checks to prevent misuse, specifically:

- You cannot use a secret key in the browser (matches on the `User-Agent` header) and it will always reply with HTTP 401 Unauthorized.
- You don't need to have any secret keys if you are not using them.

Best practices for handling secret keys

Below are some starting guidelines on how to securely work with secret keys:

- Always work with secret keys on computers you fully own or control.
- Use secure & encrypted send tools to share API keys with others (often provided by good password managers), but prefer the API Keys dashboard instead.
- Prefer encrypting them when stored in files or environment variables.
- Do not add in source control, especially for CI scripts and tools. Prefer using the tool's native secrets capability instead.
- Prefer using a separate secret key for each separate backend component of your application, so that if one is found to be vulnerable or to have leaked the key you will only need to change it and not all.

Even though a secret key will always return HTTP 401 Unauthorized error when used in a browser, it does not mean that attackers will not use it with other tools. Delete immediately!

If you must include them in logs, log the first few random characters (but never more than 6).

If you wish to log or store which valid API key was used, store it as a SHA256 hash.

What to do if a secret key or `service_role` has been leaked or compromised?

Don't rush if this has happened, or you are suspecting it has. Make sure you have fully considered the situation and have remediated the root cause of the suspicion or vulnerability **first**. Consider using the OWASP Risk Rating Methodology as an easy way to identify the severity of the incident and to plan your next steps.

Rotating a secret key (`sb_secret_...`) is easy and painless. Use the API Keys dashboard to create a new secret API key, then replace it with the compromised key. Once all components are using the new key, delete the compromised one.

Deleting a secret key is irreversible and once done it will be gone forever.

If you are still using the JWT-based `service_role` key, there are two options.

1. **Strongly recommended:** Replace the `service_role` key with a new secret key instead. Follow the guide from above as if you are rotating an existing secret key.
2. Rotate your project's JWT secret. This operation is only recommended if you suspect that the JWT secret has leaked itself. Consider switching your `anon` JWT-based key to the publishable key, and all `service_role` JWT-based keys to secret keys. Only then rotate the JWT secret. Check the FAQ below if you use the JWT-based keys in mobile, desktop or CLI applications!

Known limitations and compatibility differences

As the publishable and secret keys are no longer JWT-based, there are some known limitations and compatibility differences that you may need to plan for:

You cannot send a publishable or secret key in the `Authorization: Bearer ...` header, except if the value exactly equals the `apikey` header. In this case, your request will be forwarded down to your project's database, but will be rejected as the value is not a JWT.

Edge Functions **only support JWT verification** via the `anon` and `service_role` JWT-based API keys. You will need to use the `--no-verify-jwt` option when using publishable and secret keys. The Supabase platform does not verify the `apikey` header when using Edge Functions in this way. Implement your own `apikey`-header authorization logic inside the Edge Function code itself.

Public Realtime connections are limited to 24 hours in duration, unless the connection is upgraded and further maintained with user-level authentication via Supabase Auth or a supported Third-Party Auth provider.

Frequently asked questions

I am using JWT-based `anon` key in a mobile, desktop, or CLI application and need to rotate my `service_role` JWT secret?

If you know or suspect that the JWT secret itself is leaked, refer to the section on rotating the JWT.

If the JWT secret is secure, prefer substituting the `service_role` JWT-based key with a new secret key which you can create in the API Keys dashboard. This will prevent downtime for your application.

Can I still use my old `anon` and `service-role` API keys after enabling the publishable and secret keys?

Yes. This allows you to transition between the API keys with zero downtime by gradually swapping your clients while both sets of keys are active. See the next question for how to deactivate your keys once all your clients are switched over.

How do I deactivate the `anon` and `service_role` JWT-based API keys after moving to publishable and secret keys?

You can do this in the API Keys dashboard. To prevent downtime in your application's components, use the last used indicators on the page to confirm that these are no longer used before deactivating.

You can re-activate them should you need to.

Why are `anon` and `service_role` JWT-based keys no longer recommended?

Since the start of Supabase, the JWT-based `anon` and `service_role` keys were the right trade-off against simplicity and relative security for your project. Unfortunately they pose some real challenges in live applications, especially around rotation and security best practices.

The main reasons for preferring the publishable and secret keys (`sb_publishable_...` and `sb_secret_...`) are:

- Tight coupling between the JWT secret (which itself can be compromised, if you mint your own JWTs), the `anon` (low privilege) and `service_role` (high privilege) and `authenticated` (issued by Supabase Auth) Postgres roles.
- Inability to independently rotate each aspect of the keys, without downtime.
- Inability to roll-back an unnecessary or problematic JWT secret rotation.
- Publishing new versions of mobile applications can take days and often weeks in the app review phase with Apple's App Store and Google's Play Store. A forced rotation can cause weeks of downtime for mobile app users.
- Users may continue using desktop, CLI and mobile apps with very old versions, making rotation impossible without a forced version upgrade.
- JWTs had 10-year expiry duration, giving malicious actors more to work with.
- JWTs were self-referential and full of redundant information not necessary for achieving their primary purpose.
- JWTs are large, hard to parse, verify, and manipulate -- leading to insecure logging or bad security practices.
- They were signed with a symmetric JWT secret.

Why is there no publishable or secret keys in the CLI / self-hosting?

Publishable and secret keys are only available on the Supabase hosted platform. They are managed by our API Gateway component, which does not currently have a CLI equivalent.

We are looking into providing similar but limited in scope support for publishable or secret keys in the future. For now you can only use the `anon` and `service_role` JWT-based keys there.

For advanced users, see the following question on how these keys are implemented on the hosted platform for an idea on how to provide similar functionality for yourself.

How are publishable and secret keys implemented on the hosted platform?

When your applications use the Supabase APIs they go through a component called the API Gateway on the Supabase hosted platform. This provides us (and therefore you) with the following features:

- Observability and logging.
- Performance and request routing (such as to read-replicas).
- Security, for blocking malicious patterns or behavior on a global scale.

This API Gateway component is able to verify the API key (sent in the `apikey` request header, or for WebSocket in a query param) against your project's publishable and secret key list. If the match is found, it mints a temporary, short-lived JWT that is then forwarded down to your project's servers.

It may be possible to replicate similar behavior if you self-host by using programmable proxies such as Kong, Envoy, NGINX or similar.

—