

```
file: ./content/docs/basic-usage.mdx
meta: {
  "title": "Basic Usage",
  "description": "Getting started with Better Auth"
}
```

Better Auth provides built-in authentication support for:

- \* \*\*Email and password\*\*
- \* \*\*Social provider (Google, GitHub, Apple, and more)\*\*

But also can easily be extended using plugins, such as: [username](/docs/plugins/username), [magic link](/docs/plugins/magic-link), [passkey](/docs/plugins/passkey), [email-otp](/docs/plugins/email-otp), and more.

### ## Email & Password

To enable email and password authentication:

```
```ts title="auth.ts"
import { betterAuth } from "better-auth"

export const auth = betterAuth({
  emailAndPassword: { // [!code highlight]
    enabled: true // [!code highlight]
  } // [!code highlight]
})
```
```

### ### Sign Up

To sign up a user you need to call the client method `signUp.email` with the user's information.

```
```ts title="sign-up.ts"
import { authClient } from "@lib/auth-client"; //import the auth client // [!code highlight]

const { data, error } = await authClient.signUp.email({
  email, // user email address
  password, // user password -> min 8 characters by default
  name, // user display name
  image, // User image URL (optional)
  callbackURL: "/dashboard" // A URL to redirect to after the user verifies their email (optional)
}, {
  onRequest: (ctx) => {
    //show loading
  },
  onSuccess: (ctx) => {
    //redirect to the dashboard or sign in page
  },
  onError: (ctx) => {
    // display the error message
    alert(ctx.error.message);
  },
});
```
```

By default, the users are automatically signed in after they successfully sign up. To disable this behavior you can set `autoSignIn` to `false`.

```
```ts title="auth.ts"
import { betterAuth } from "better-auth"

export const auth = betterAuth({
  emailAndPassword: {
    enabled: true,
```

```

    autoSignIn: false //defaults to true // [!code highlight]
  },
})
```

```

### ### Sign In

To sign a user in, you can use the `signIn.email` function provided by the client.

```

```ts title="sign-in"
const { data, error } = await authClient.signIn.email({
  /**
   * The user email
   */
  email,
  /**
   * The user password
   */
  password,
  /**
   * A URL to redirect to after the user verifies their email (optional)
   */
  callbackURL: "/dashboard",
  /**
   * remember the user session after the browser is closed.
   * @default true
   */
  rememberMe: false
}, {
  //callbacks
})
```

```

<Callout type="warn">  
 Always invoke client methods from the client side. Don't call them from the server.  
 </Callout>

### ### Server-Side Authentication

To authenticate a user on the server, you can use the `auth.api` methods.

```

```ts title="server.ts"
import { auth } from "../auth"; // path to your Better Auth server instance

const response = await auth.api.signInEmail({
  body: {
    email,
    password
  },
  asResponse: true // returns a response object instead of data
});
```

```

<Callout>  
 If the server cannot return a response object, you'll need to manually parse and set cookies. But for frameworks like Next.js we provide [a plugin](/docs/integrations/next#server-action-cookies) to handle this automatically  
 </Callout>

### ## Social Sign-On

Better Auth supports multiple social providers, including Google, GitHub, Apple, Discord, and more. To use a social provider, you need to configure the ones you need in the `socialProviders` option on your `auth` object.

```

```ts title="auth.ts"
import { betterAuth } from "better-auth";

```

```
export const auth = betterAuth({
  socialProviders: { // [!code highlight]
    github: { // [!code highlight]
      clientId: process.env.GITHUB_CLIENT_ID!, // [!code highlight]
      clientSecret: process.env.GITHUB_CLIENT_SECRET!, // [!code highlight]
    } // [!code highlight]
  }, // [!code highlight]
})
```,
```

### ### Sign in with social providers

To sign in using a social provider you need to call `signIn.social`. It takes an object with the following properties:

```
```ts title="sign-in.ts"
import { authClient } from "@lib/auth-client"; //import the auth client // [!code highlight]

await authClient.signIn.social({
  /**
   * The social provider ID
   * @example "github", "google", "apple"
   */
  provider: "github",
  /**
   * A URL to redirect after the user authenticates with the provider
   * @default "/"
   */
  callbackURL: "/dashboard",
  /**
   * A URL to redirect if an error occurs during the sign in process
   */
  errorCallbackURL: "/error",
  /**
   * A URL to redirect if the user is newly registered
   */
  newUserCallbackURL: "/welcome",
  /**
   * disable the automatic redirect to the provider.
   * @default false
   */
  disableRedirect: true,
});
```,
```

You can also authenticate using `idToken` or `accessToken` from the social provider instead of redirecting the user to the provider's site. See social providers documentation for more details.

### ## Signout

To signout a user, you can use the `signOut` function provided by the client.

```
```ts title="user-card.tsx"
await authClient.signOut();
```,
```

you can pass `fetchOptions` to redirect onSuccess

```
```ts title="user-card.tsx"
await authClient.signOut({
  fetchOptions: {
    onSuccess: () => {
      router.push("/login"); // redirect to login page
    },
  },
},
```

```
});
```,
```

## ## Session

Once a user is signed in, you'll want to access the user session. Better Auth allows you easily to access the session data from the server and client side.

### #### Client Side

#### ##### Use Session

Better Auth provides a `useSession` hook to easily access session data on the client side. This hook is implemented using nanostore and has support for each supported framework and vanilla client, ensuring that any changes to the session (such as signing out) are immediately reflected in your UI.

```
<Tabs items={["React", "Vue", "Svelte", "Solid", "Vanilla"]} defaultValue="React">
  <Tab value="React">
    ```tsx title="user.tsx"
    import { authClient } from "@lib/auth-client" // import the auth client // [!code highlight]

    export function User(){

      const { // [!code highlight]
        data: session, // [!code highlight]
        isPending, //loading state // [!code highlight]
        error, //error object // [!code highlight]
        refetch //refetch the session
      } = authClient.useSession() // [!code highlight]

      return (
        //...
      )
    }
    ```
  </Tab>

  <Tab value="Vue">
    ```vue title="index.vue"
    <script setup lang="ts">
    import { authClient } from "~/lib/auth-client" // [!code highlight]

    const session = authClient.useSession() // [!code highlight]
    </script>

    <template>
      <div>
        <div>
          <pre>{{ session.data }}</pre>
          <button v-if="session.data" @click="authClient.signOut()">
            Sign out
          </button>
        </div>
      </div>
    </template>
    ```
  </Tab>

  <Tab value="Svelte">
    ```svelte title="user.svelte"
    <script lang="ts">
    import { authClient } from "$lib/auth-client"; // [!code highlight]

    const session = authClient.useSession(); // [!code highlight]
    </script>
```

```

    <p>
      {$session.data?.user.email}
    </p>
    ...
  </Tab>

  <Tab value="Vanilla">
    ```ts title="user.svelte"
    import { authClient } from "~/lib/auth-client"; //import the auth client

    authClient.useSession.subscribe((value)=>{
      //do something with the session //
    })
    ...
  </Tab>

  <Tab value="Solid">
    ```tsx title="user.tsx"
    import { authClient } from "~/lib/auth-client"; // [!code highlight]

    export default function Home() {
      const session = authClient.useSession() // [!code highlight]
      return (
        <pre>{JSON.stringify(session(), null, 2)}</pre>
      );
    }
    ...
  </Tab>
</Tabs>

```

#### #### Get Session

If you prefer not to use the hook, you can use the `getSession` method provided by the client.

```

```ts title="user.tsx"
import { authClient } from "@lib/auth-client" // import the auth client // [!code highlight]

const { data: session, error } = await authClient.getSession()
```

```

You can also use it with client-side data-fetching libraries like [TanStack Query](https://tanstack.com/query/latest).

#### ### Server Side

The server provides a `session` object that you can use to access the session data. It requires request headers object to be passed to the `getSession` method.

**\*\*Example: Using some popular frameworks\*\***

```

<Tabs items={["Next.js", "Nuxt", "Svelte", "Astro", "Hono", "TanStack"]}>
  <Tab value="Next.js">
    ```ts title="server.ts"
    import { auth } from "./auth"; // path to your Better Auth server instance
    import { headers } from "next/headers";

    const session = await auth.api.getSession({
      headers: await headers() // you need to pass the headers object.
    })
    ...
  </Tab>

  <Tab value="Remix">
    ```ts title="route.ts"
    import { auth } from "lib/auth"; // path to your Better Auth server instance

```

```
export async function loader({ request }: LoaderFunctionArgs) {
  const session = await auth.api.getSession({
    headers: request.headers
  })

  return json({ session })
}
...

```

</Tab>

```
<Tab value="Astro">
  `` astro title="index.astro"
  ---
  import { auth } from "./auth";

  const session = await auth.api.getSession({
    headers: Astro.request.headers,
  });
  ---
  <!-- Your Astro Template -->
  ...

```

</Tab>

```
<Tab value="Svelte">
  `` ts title="+page.ts"
  import { auth } from "./auth";

  export async function load({ request }) {
    const session = await auth.api.getSession({
      headers: request.headers
    })
    return {
      props: {
        session
      }
    }
  }
  ...

```

</Tab>

```
<Tab value="Hono">
  `` ts title="index.ts"
  import { auth } from "./auth";

  const app = new Hono();

  app.get("/path", async (c) => {
    const session = await auth.api.getSession({
      headers: c.req.raw.headers
    })
  });
  ...

```

</Tab>

```
<Tab value="Nuxt">
  `` ts title="server/session.ts"
  import { auth } from "~/utils/auth";

  export default defineEventHandler((event) => {
    const session = await auth.api.getSession({
      headers: event.headers,
    })
  });
  ...

```

</Tab>

```

<Tab value="TanStack">
  `` `ts title="app/routes/api/index.ts"
  import { auth } from "../auth";
  import { createAPIFileRoute } from "@tanstack/start/api";

  export const APIRoute = createAPIFileRoute("/api/$")({
    GET: async ({ request }) => {
      const session = await auth.api.getSession({
        headers: request.headers
      })
    },
  });
  `` `
</Tab>
</Tabs>

```

<Callout>  
 For more details check [session-management](/docs/concepts/session-management) documentation.  
 </Callout>

## ## Using Plugins

One of the unique features of Better Auth is a plugins ecosystem. It allows you to add complex auth related functionality with small lines of code.

Below is an example of how to add two factor authentication using two factor plugin.

<Steps>  
 <Step>  
 #### Server Configuration

To add a plugin, you need to import the plugin and pass it to the `plugins` option of the auth instance. For example, to add two factor authentication, you can use the following code:

```

`` `ts title="auth.ts"
import { betterAuth } from "better-auth"
import { twoFactor } from "better-auth/plugins" // [!code highlight]

export const auth = betterAuth({
  //...rest of the options
  plugins: [ // [!code highlight]
    twoFactor() // [!code highlight]
  ] // [!code highlight]
})
`` `

```

now two factor related routes and method will be available on the server.

</Step>

<Step>  
 #### Migrate Database

After adding the plugin, you'll need to add the required tables to your database. You can do this by running the `migrate` command, or by using the `generate` command to create the schema and handle the migration manually.

generating the schema:

```

`` `bash title="terminal"
npx @better-auth/cli generate
`` `

```

using the `migrate` command:

```

`` `bash title="terminal"

```

```
npx @better-auth/cli migrate
```

```

<Callout>

If you prefer adding the schema manually, you can check the schema required on the [two factor plugin] (/docs/plugins/2fa#schema) documentation.

</Callout>

</Step>

<Step>

#### Client Configuration

Once we're done with the server, we need to add the plugin to the client. To do this, you need to import the plugin and pass it to the `plugins` option of the auth client. For example, to add two factor authentication, you can use the following code:

```
```ts title="auth-client.ts"
import { createAuthClient } from "better-auth/client";
import { twoFactorClient } from "better-auth/client/plugins"; // [!code highlight]

const authClient = createAuthClient({
  plugins: [ // [!code highlight]
    twoFactorClient({ // [!code highlight]
      twoFactorPage: "/two-factor" // the page to redirect if a user need to verify 2nd factor // [!code highlight]
    }) // [!code highlight]
  ]) // [!code highlight]
});
```

```

now two factor related methods will be available on the client.

```
```ts title="profile.ts"
import { authClient } from "../auth-client"

const enableTwoFactor = async() => {
  const data = await authClient.twoFactor.enable({
    password // the user password is required
  }) // this will enable two factor
}

const disableTwoFactor = async() => {
  const data = await authClient.twoFactor.disable({
    password // the user password is required
  }) // this will disable two factor
}

const signInWith2Factor = async() => {
  const data = await authClient.signIn.email({
    //...
  })
  //if the user has two factor enabled, it will redirect to the two factor page
}

const verifyTOTP = async() => {
  const data = await authClient.twoFactor.verifyTOTP({
    code: "123456", // the code entered by the user
    /**
     * If the device is trusted, the user won't
     * need to pass 2FA again on the same device
     */
    trustDevice: true
  })
}
```

```

</Step>



```

<Step>
  Next step: See the <Link href="/docs/plugins/2fa">two factor plugin documentation</Link>.
</Step>
</Steps>

```

```
file: ./content/docs/comparison.mdx
```

```

meta: {
  "title": "Comparison",
  "description": "Comparison of Better Auth versus over other auth libraries and services."
}

```

```
> <p className="text-orange-200">Comparison is the thief of joy.</p>
```

Here are non detailed reasons why you may want to use Better Auth over other auth libraries and services.

### #### vs Other Auth Libraries

- \* \*\*Framework agnostic\*\* - Works with any framework, not just specific ones
- \* \*\*Advanced features built-in\*\* - 2FA, multi-tenancy, multi-session, rate limiting, and many more
- \* \*\*Plugin system\*\* - Extend functionality without forking or complex workarounds
- \* \*\*Full control\*\* - Customize auth flows exactly how you want

### #### vs Self-Hosted Auth Servers

- \* \*\*No separate infrastructure\*\* - Runs in your app, users stay in your database
- \* \*\*Zero server maintenance\*\* - No auth servers to deploy, monitor, or update
- \* \*\*Complete feature set\*\* - Everything you need without the operational overhead

### #### vs Managed Auth Services

- \* \*\*Keep your data\*\* - Users stay in your database, not a third-party service
- \* \*\*No per-user costs\*\* - Scale without worrying about auth billing
- \* \*\*Single source of truth\*\* - All user data in one place

### #### vs Rolling Your Own

- \* \*\*Security handled\*\* - Battle-tested auth flows and security practices
- \* \*\*Focus on your product\*\* - Spend time on features that matter to your business
- \* \*\*Plugin extensibility\*\* - Add custom features without starting from scratch

```
file: ./content/docs/installation.mdx
```

```

meta: {
  "title": "Installation",
  "description": "Learn how to configure Better Auth in your project."
}

```

```

<Steps>
  <Step>
    #### Install the Package

```

Let's start by adding Better Auth to your project:

```

<Tabs groupId="package-manager" persist items={}>
  <Tab value="npm">
    ```bash
    npm install better-auth
    ```
  </Tab>

  <Tab value="pnpm">
    ```bash
    pnpm add better-auth

```

```

    ` ` `
  </Tab>

  <Tab value="yarn">
    ` ` ` bash
    yarn add better-auth
    ` ` `
  </Tab>

  <Tab value="bun">
    ` ` ` bash
    bun add better-auth
    ` ` `
  </Tab>
</Tabs>

<Callout type="info">
  If you're using a separate client and server setup, make sure to install Better Auth in both parts of your project.
</Callout>
</Step>

```

```

<Step>
  #### Set Environment Variables

```

Create a `.env` file in the root of your project and add the following environment variables:

#### 1. **Secret Key**

Random value used by the library for encryption and generating hashes. **You can generate one using the button below** or you can use something like openssl.

```

` ` ` txt title=".env"
BETTER_AUTH_SECRET=
` ` `

```

```

<GenerateSecret />

```

#### 2. **Set Base URL**

```

` ` ` txt title=".env"
BETTER_AUTH_URL=http://localhost:3000 #Base URL of your app
` ` `

```

```

</Step>

```

```

<Step>
  #### Create A Better Auth Instance

```

Create a file named `auth.ts` in one of these locations:

- \* Project root
- \* `lib/` folder
- \* `utils/` folder

You can also nest any of these folders under `src/`, `app/` or `server/` folder. (e.g. `src/lib/auth.ts`, `app/lib/auth.ts`).

And in this file, import Better Auth and create your auth instance. Make sure to export the auth instance with the variable name `auth` or as a `default` export.

```

` ` ` ts title="auth.ts"
import { betterAuth } from "better-auth";

export const auth = betterAuth({
  //...
})

```

```

  ...
</Step>

```

```

<Step>
#### Configure Database

```

Better Auth requires a database to store user data. You can easily configure Better Auth to use SQLite, PostgreSQL, or MySQL, with Kysely handling queries and migrations for these databases.

```

<Tabs items=["sqlite", "postgres", "mysql"]>
<Tab value="sqlite">
  ```ts title="auth.ts"
  import { betterAuth } from "better-auth";
  import Database from "better-sqlite3";

  export const auth = betterAuth({
    database: new Database("./sqlite.db"),
  })
  ...
</Tab>

<Tab value="postgres">
  ```ts title="auth.ts"
  import { betterAuth } from "better-auth";
  import { Pool } from "pg";

  export const auth = betterAuth({
    database: new Pool({
      // connection options
    })
  })
  ...
</Tab>

<Tab value="mysql">
  ```ts title="auth.ts"
  import { betterAuth } from "better-auth";
  import { createPool } from "mysql2/promise";

  export const auth = betterAuth({
    database: createPool({
      // connection options
    })
  })
  ...
</Tab>
</Tabs>

```

You can also provide any Kysely dialect or a Kysely instance to the `database` option.

**\*\*Example with LibsqlDialect:\*\***

```

```ts title="auth.ts"
import { betterAuth } from "better-auth";
import { LibsqlDialect } from "@libsql/kysely-libsql";

const dialect = new LibsqlDialect({
  url: process.env.TURSO_DATABASE_URL || "",
  authToken: process.env.TURSO_AUTH_TOKEN || "",
})

export const auth = betterAuth({
  database: {
    dialect,
    type: "sqlite"
  }
})

```

```

    }
  });
  ...

```

### **\*\*Adapters\*\***

If you prefer to use an ORM or if your database is not supported by Kysely, you can use one of the built-in adapters.

```

<Tabs items=["prisma", "drizzle", "mongodb"]>
  <Tab value="prisma">
    ```ts title="auth.ts"
    import { betterAuth } from "better-auth";
    import { prismaAdapter } from "better-auth/adapters/prisma";
    // If your Prisma file is located elsewhere, you can change the path
    import { PrismaClient } from "@generated/prisma";

    const prisma = new PrismaClient();
    export const auth = betterAuth({
      database: prismaAdapter(prisma, {
        provider: "sqlite", // or "mysql", "postgresql", ...etc
      }),
    });
    ...
  </Tab>

  <Tab value="drizzle">
    ```ts title="auth.ts"
    import { betterAuth } from "better-auth";
    import { drizzleAdapter } from "better-auth/adapters/drizzle";
    import { db } from "@db"; // your drizzle instance

    export const auth = betterAuth({
      database: drizzleAdapter(db, {
        provider: "pg", // or "mysql", "sqlite"
      })
    });
    ...
  </Tab>

  <Tab value="mongodb">
    ```ts title="auth.ts"
    import { betterAuth } from "better-auth";
    import { mongodbAdapter } from "better-auth/adapters/mongodb";
    import { client } from "@db"; // your mongodb client

    export const auth = betterAuth({
      database: mongodbAdapter(client)
    });
    ...
  </Tab>
</Tabs>
</Step>

```

### **<Step>** **#### Create Database Tables**

Better Auth includes a CLI tool to help manage the schema required by the library.

**\* \*\*Generate\*\*:** This command generates an ORM schema or SQL migration file.

#### **<Callout>**

If you're using Kysely, you can apply the migration directly with `migrate` command below. Use `generate` only if you plan to apply the migration manually.

#### **</Callout>**

```
```bash title="Terminal"
npx @better-auth/cli generate
```
```

**\*\*Migrate\*\***: This command creates the required tables directly in the database. (Available only for the built-in Kysely adapter)

```
```bash title="Terminal"
npx @better-auth/cli migrate
```
```

see the [CLI documentation](/docs/concepts/cli) for more information.

<Callout>

If you instead want to create the schema manually, you can find the core schema required in the [database section](/docs/concepts/database#core-schema).

</Callout>

</Step>

<Step>

#### Authentication Methods

Configure the authentication methods you want to use. Better Auth comes with built-in support for email/password, and social sign-on providers.

```
```ts title="auth.ts"
import { betterAuth } from "better-auth"

export const auth = betterAuth({
  //...other options
  emailAndPassword: { // [!code highlight]
    enabled: true // [!code highlight]
  }, // [!code highlight]
  socialProviders: { // [!code highlight]
    github: { // [!code highlight]
      clientId: process.env.GITHUB_CLIENT_ID as string, // [!code highlight]
      clientSecret: process.env.GITHUB_CLIENT_SECRET as string, // [!code highlight]
    }, // [!code highlight]
  }, // [!code highlight]
});
```
```

<Callout type="info">

You can use even more authentication methods like [passkey](/docs/plugins/passkey), [username](/docs/plugins/username), [magic link](/docs/plugins/magic-link) and more through plugins.

</Callout>

</Step>

<Step>

#### Mount Handler

To handle API requests, you need to set up a route handler on your server.

Create a new file or route in your framework's designated catch-all route handler. This route should handle requests for the path `/api/auth/*` (unless you've configured a different base path).

<Callout>

Better Auth supports any backend framework with standard Request and Response objects and offers helper functions for popular frameworks.

</Callout>

<Tabs items=["next-js", "nuxt", "svelte-kit", "remix", "solid-start", "hono", "express", "elysia", "tanstack-start", "expo"]  
defaultValue="react">

<Tab value="next-js">

```
```ts title="/app/api/auth/[...all]/route.ts"
```

```

import { auth } from "@lib/auth"; // path to your auth file
import { toNextJsHandler } from "better-auth/next-js";

export const { POST, GET } = toNextJsHandler(auth);
...
</Tab>

<Tab value="nuxt">
  `` ts title="/server/api/auth/[...all].ts"
  import { auth } from "~/utils/auth"; // path to your auth file

  export default defineEventHandler((event) => {
    return auth.handler(toWebRequest(event));
  });
  ...
</Tab>

<Tab value="svelte-kit">
  `` ts title="hooks.server.ts"
  import { auth } from "$lib/auth"; // path to your auth file
  import { svelteKitHandler } from "better-auth/svelte-kit";

  export async function handle({ event, resolve }) {
    return svelteKitHandler({ event, resolve, auth });
  }
  ...
</Tab>

<Tab value="remix">
  `` ts title="/app/routes/api.auth.$ts"
  import { auth } from '~/lib/auth.server' // Adjust the path as necessary
  import type { LoaderFunctionArgs, ActionFunctionArgs } from "@remix-run/node"

  export async function loader({ request }: LoaderFunctionArgs) {
    return auth.handler(request)
  }

  export async function action({ request }: ActionFunctionArgs) {
    return auth.handler(request)
  }
  ...
</Tab>

<Tab value="solid-start">
  `` ts title="/routes/api/auth/*all.ts"
  import { auth } from "~/lib/auth"; // path to your auth file
  import { toSolidStartHandler } from "better-auth/solid-start";

  export const { GET, POST } = toSolidStartHandler(auth);
  ...
</Tab>

<Tab value="hono">
  `` ts title="src/index.ts"
  import { Hono } from "hono";
  import { auth } from "./auth"; // path to your auth file
  import { serve } from "@hono/node-server";
  import { cors } from "hono/cors";

  const app = new Hono();

  app.on(["POST", "GET"], "/api/auth/**", (c) => auth.handler(c.req.raw));

  serve(app);
  ...

```

&lt;/Tab&gt;

&lt;Tab value="express"&gt;

&lt;Callout type="warn"&gt;

ExpressJS v5 introduced breaking changes to route path matching by switching to `path-to-regexp@6`. Wildcard routes like `\*` should now be written using the new named syntax, e.g. `{\*any}`, to properly capture catch-all patterns. This ensures compatibility and predictable behavior across routing scenarios.

See the [Express v5 migration guide](https://expressjs.com/en/guide/migrating-5.html) for details.

As a result, the implementation in ExpressJS v5 should look like this:

```
```ts
app.all('/api/auth/{*any}', toNodeHandler(auth));
```
```

\*The name any is arbitrary and can be replaced with any identifier you prefer.\*

&lt;/Callout&gt;

```
```ts title="server.ts"
```

```
import express from "express";
import { toNodeHandler } from "better-auth/node";
import { auth } from "./auth";
```

```
const app = express();
const port = 8000;
```

```
app.all("/api/auth/*", toNodeHandler(auth));
```

```
// Mount express json middleware after Better Auth handler
// or only apply it to routes that don't interact with Better Auth
app.use(express.json());
```

```
app.listen(port, () => {
  console.log(`Better Auth app listening on port ${port}`);
});
```
```

This will also work for any other node server framework like express, fastify, hapi, etc., but may require some modifications. See [fastify guide](/docs/integrations/fastify). Note that CommonJS (cjs) isn't supported.

&lt;/Tab&gt;

&lt;Tab value="astro"&gt;

```
```ts title="/pages/api/auth/[...all].ts"
import type { APIRoute } from "astro";
import { auth } from "@auth"; // path to your auth file
```

```
export const GET: APIRoute = async (ctx) => {
  return auth.handler(ctx.request);
};
```

```
export const POST: APIRoute = async (ctx) => {
  return auth.handler(ctx.request);
};
```
```

&lt;/Tab&gt;

&lt;Tab value="elysia"&gt;

```
```ts
import { Elysia, Context } from "elysia";
import { auth } from "./auth";
```

```
const betterAuthView = (context: Context) => {
  const BETTER_AUTH_ACCEPT_METHODS = ["POST", "GET"]
  // validate request method
  if(BETTER_AUTH_ACCEPT_METHODS.includes(context.request.method)) {
```

```

    return auth.handler(context.request);
  } else {
    context.error(405)
  }
}

const app = new Elysia().all("/api/auth/*", betterAuthView).listen(3000);

console.log(
  `🦊 Elysia is running at ${app.server?.hostname}:${app.server?.port}`
);
...
</Tab>

<Tab value="tanstack-start">
  `` `ts title="src/routes/api/auth/$.ts"
  import { auth } from '~/lib/server/auth'
  import { createServerFileRoute } from '@tanstack/react-start/server'

  export const ServerRoute = createServerFileRoute('/api/auth/$').methods({
    GET: ({ request }) => {
      return auth.handler(request)
    },
    POST: ({ request }) => {
      return auth.handler(request)
    },
  });
  ...
</Tab>

<Tab value="expo">
  `` `ts title="app/api/auth/[...all]+api.ts"
  import { auth } from '@lib/server/auth'; // path to your auth file

  const handler = auth.handler;
  export { handler as GET, handler as POST };
  ...
</Tab>
</Tabs>
</Step>

```

### <Step> #### Create Client Instance

The client-side library helps you interact with the auth server. Better Auth comes with a client for all the popular web frameworks, including vanilla JavaScript.

1. Import `createAuthClient` from the package for your framework (e.g., "better-auth/react" for React).
2. Call the function to create your client.
3. Pass the base URL of your auth server. (If the auth server is running on the same domain as your client, you can skip this step.)

```

<Callout type="info">
  If you're using a different base path other than `/api/auth` make sure to pass the whole URL including the path. (e.g.
  `http://localhost:3000/custom-path/auth`)
</Callout>

<Tabs
  items={["react", "vue", "svelte", "solid",
"vanilla"]}
  defaultValue="react"
>
  <Tab value="vanilla">
    `` `ts title="lib/auth-client.ts"
    import { createAuthClient } from "better-auth/client"

```



```
export const authClient = createAuthClient({
  /** The base URL of the server (optional if you're using the same domain) */ // [!code highlight]
  baseURL: "http://localhost:3000" // [!code highlight]
})
...
</Tab>
```

```
<Tab value="react" title="lib/auth-client.ts">
  `` `ts title="lib/auth-client.ts"
  import { createAuthClient } from "better-auth/react"
  export const authClient = createAuthClient({
    /** The base URL of the server (optional if you're using the same domain) */ // [!code highlight]
    baseURL: "http://localhost:3000" // [!code highlight]
  })
  ...
</Tab>
```

```
<Tab value="vue" title="lib/auth-client.ts">
  `` `ts title="lib/auth-client.ts"
  import { createAuthClient } from "better-auth/vue"
  export const authClient = createAuthClient({
    /** The base URL of the server (optional if you're using the same domain) */ // [!code highlight]
    baseURL: "http://localhost:3000" // [!code highlight]
  })
  ...
</Tab>
```

```
<Tab value="svelte" title="lib/auth-client.ts">
  `` `ts title="lib/auth-client.ts"
  import { createAuthClient } from "better-auth/svelte"
  export const authClient = createAuthClient({
    /** The base URL of the server (optional if you're using the same domain) */ // [!code highlight]
    baseURL: "http://localhost:3000" // [!code highlight]
  })
  ...
</Tab>
```

```
<Tab value="solid" title="lib/auth-client.ts">
  `` `ts title="lib/auth-client.ts"
  import { createAuthClient } from "better-auth/solid"
  export const authClient = createAuthClient({
    /** The base URL of the server (optional if you're using the same domain) */ // [!code highlight]
    baseURL: "http://localhost:3000" // [!code highlight]
  })
  ...
</Tab>
</Tabs>
```

```
<Callout type="info">
  Tip: You can also export specific methods if you prefer:
</Callout>
```

```
`` `ts
export const { signIn, signUp, useSession } = createAuthClient()
...
</Step>
```

```
<Step>
#### 🎉 That's it!
```

That's it! You're now ready to use better-auth in your application. Continue to [\[basic usage\]\(/docs/basic-usage\)](/docs/basic-usage) to learn how to use the auth instance to sign in users.

```
</Step>
</Steps>
```

```
file: ./content/docs/introduction.mdx
meta: {
  "title": "Introduction",
  "description": "Introduction to Better Auth."
}
```

Better Auth is a framework-agnostic authentication and authorization framework for TypeScript. It provides a comprehensive set of features out of the box and includes a plugin ecosystem that simplifies adding advanced functionalities. Whether you need 2FA, multi-tenancy, multi-session support, or even enterprise features like SSO, it lets you focus on building your application instead of reinventing the wheel.

### ## Why Better Auth?

\*Authentication in the TypeScript ecosystem has long been a half-solved problem. Other open-source libraries often require a lot of additional code for anything beyond basic authentication features. Rather than just pushing third-party services as the solution, I believe we can do better as a community—hence, Better Auth.\*

### ## Features

Better Auth aims to be the most comprehensive auth library. It provides a wide range of features out of the box and allows you to extend it with plugins. Here are some of the features:

<Features />

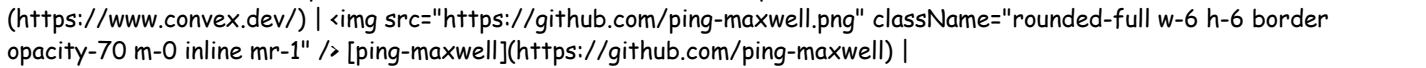
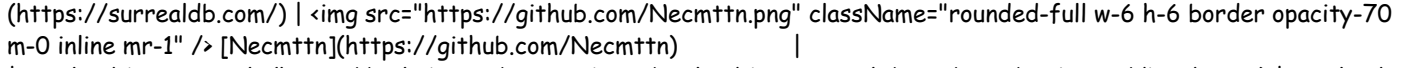
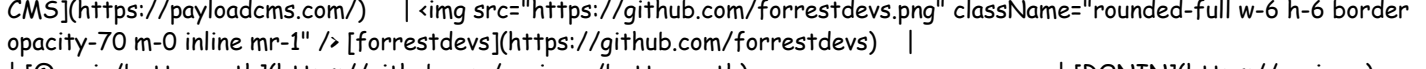
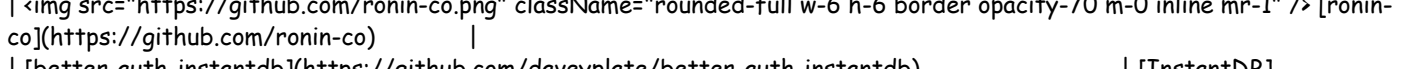
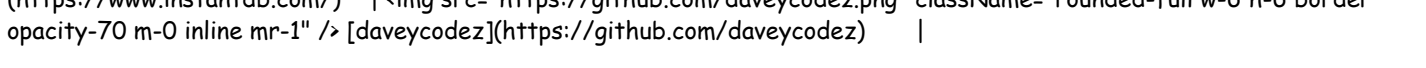
...and much more!

### ## LLMs.txt

Better Auth provides an LLMs.txt file that helps AI models understand how to interact with your authentication system. You can find it at [<https://better-auth.com/llms.txt>](<https://better-auth.com/llms.txt>).

```
file: ./content/docs/adapters/community-adapters.mdx
meta: {
  "title": "Community Adapters",
  "description": "Integrate Better Auth with community made database adapters."
}
```

This page showcases a list of recommended community made database adapters.  
We encourage you to create any missing database adapters and maybe get added to the list!

Adapter	Database Dialect	Author
<hr/>		
[convex-better-auth]( <a href="https://www.better-auth-kit.com/docs/adapters/convex">https://www.better-auth-kit.com/docs/adapters/convex</a> )		[Convex Database] ( <a href="https://www.convex.dev/">https://www.convex.dev/</a> )
		
[surrealdb-better-auth]( <a href="https://github.com/Necmttn/surrealdb-better-auth">https://github.com/Necmttn/surrealdb-better-auth</a> )		[Surreal Database] ( <a href="https://surrealdb.com/">https://surrealdb.com/</a> )
		
[payload-better-auth]( <a href="https://github.com/ForrestDevs/payload-better-auth/tree/main/packages/db-adapter">https://github.com/ForrestDevs/payload-better-auth/tree/main/packages/db-adapter</a> )		[Payload CMS] ( <a href="https://payloadcms.com/">https://payloadcms.com/</a> )
		
[@ronin/better-auth]( <a href="https://github.com/ronin-co/better-auth">https://github.com/ronin-co/better-auth</a> )		[RONIN] ( <a href="https://ronin.co">https://ronin.co</a> )
		
[better-auth-instantdb]( <a href="https://github.com/daveyplate/better-auth-instantdb">https://github.com/daveyplate/better-auth-instantdb</a> )		[InstantDB] ( <a href="https://www.instantdb.com/">https://www.instantdb.com/</a> )
		

```
file: ./content/docs/adapters/drizzle.mdx
```

```
meta: {
  "title": "Drizzle ORM Adapter",
  "description": "Integrate Better Auth with Drizzle ORM."
}
```

Drizzle ORM is a powerful and flexible ORM for Node.js and TypeScript. It provides a simple and intuitive API for working with databases, and supports a wide range of databases including MySQL, PostgreSQL, SQLite, and more. Read more here: [Drizzle ORM](https://orm.drizzle.team/).

### ## Example Usage

Make sure you have Drizzle installed and configured. Then, you can use the Drizzle adapter to connect to your database.

```
```ts title="auth.ts"
import { betterAuth } from "better-auth";
import { drizzleAdapter } from "better-auth/adapters/drizzle";
import { db } from "./database.ts";

export const auth = betterAuth({
  database: drizzleAdapter(db, {
    // [!code highlight]
    provider: "sqlite", // or "pg" or "mysql" // [!code highlight]
  }), // [!code highlight]
  //... the rest of your config
});
```
```

### ## Schema generation & migration

The [Better Auth CLI](/docs/concepts/cli) allows you to generate or migrate your database schema based on your Better Auth configuration and plugins.

To generate the schema required by Better Auth, run the following command:

```
```bash title="Schema Generation"
npx @better-auth/cli@latest generate
```
```

To generate and apply the migration, run the following commands:

```
```bash title="Schema Migration"
npx drizzle-kit generate # generate the migration file
npx drizzle-kit migrate # apply the migration
```
```

### ## Additional Information

The Drizzle adapter expects the schema you define to match the table names. For example, if your Drizzle schema maps the `user` table to `users`, you need to manually pass the schema and map it to the user table.

```
```ts
import { betterAuth } from "better-auth";
import { db } from "./drizzle";
import { drizzleAdapter } from "better-auth/adapters/drizzle";
import { schema } from "./schema";

export const auth = betterAuth({
  database: drizzleAdapter(db, {
    provider: "sqlite", // or "pg" or "mysql"
    schema: {
      ...schema,
      user: schema.users,
    },
  }),
});
```
```

```
});
```,
```

If all your tables are using plural form, you can just pass the `usePlural` option:

```
```ts
export const auth = betterAuth({
  database: drizzleAdapter(db, {
    ...
    usePlural: true,
  }),
});
```,
```

If you're looking for performance improvements or tips, take a look at our guide to [performance optimizations](/docs/guides/optimizing-for-performance).

```
file: ./content/docs/adapters/mongo.mdx
meta: {
  "title": "MongoDB Adapter",
  "description": "Integrate Better Auth with MongoDB."
}
```

MongoDB is a popular NoSQL database that is widely used for building scalable and flexible applications. It provides a flexible schema that allows for easy data modeling and querying. Read more here: [\[MongoDB\]\(https://www.mongodb.com/\)](https://www.mongodb.com/).

### ## Example Usage

Make sure you have MongoDB installed and configured. Then, you can use the mongodb adapter.

```
```ts title="auth.ts"
import { betterAuth } from "better-auth";
import { MongoClient } from "mongodb";
import { mongodbAdapter } from "better-auth/adapters/mongodb";

const client = new MongoClient("mongodb://localhost:27017/database");
const db = client.db();

export const auth = betterAuth({
  database: mongodbAdapter(db),
});
```,
```

### ## Schema generation & migration

For MongoDB, we don't need to generate or migrate the schema.

```
file: ./content/docs/adapters/mssql.mdx
meta: {
  "title": "MS SQL",
  "description": "Integrate Better Auth with MS SQL."
}
```

Microsoft SQL Server is a relational database management system developed by Microsoft, designed for enterprise-level data storage, management, and analytics with robust security and scalability features. Read more [\[here\]\(https://en.wikipedia.org/wiki/Microsoft\\_SQL\\_Server\)](https://en.wikipedia.org/wiki/Microsoft_SQL_Server).

### ## Example Usage

Make sure you have MS SQL installed and configured. Then, you can connect it straight into Better Auth.

```

` `` ts title="auth.ts"
import { betterAuth } from "better-auth";
import { MssqlDialect } from "kysely";
import * as Tedious from 'tedious'
import * as Tarn from 'tarn'

const dialect = new MssqlDialect({
  tarn: {
    ...Tarn,
    options: {
      min: 0,
      max: 10,
    },
  },
  tedious: {
    ...Tedious,
    connectionFactory: () => new Tedious.Connection({
      authentication: {
        options: {
          password: 'password',
          userName: 'username',
        },
        type: 'default',
      },
      options: {
        database: 'some_db',
        port: 1433,
        trustServerCertificate: true,
      },
      server: 'localhost',
    }),
  },
})

export const auth = betterAuth({
  database: {
    dialect,
    type: "mssql"
  }
});

```

```

` ``

```

<Callout>

For more information, read Kysely's documentation to the [MssqlDialect](<https://kysely-org.github.io/kysely-apidoc/classes/MssqlDialect.html>).

</Callout>

### ## Schema generation & migration

The [Better Auth CLI](/docs/concepts/cli) allows you to generate or migrate your database schema based on your Better Auth configuration and plugins.

<table>

<tr className="border-b">

<th>

<p className="font-bold text-[16px] mb-1">MS SQL Schema Generation</p>

</th>

<th>

<p className="font-bold text-[16px] mb-1">MS SQL Schema Migration</p>

</th>

</tr>

```
<tr className="h-10">
  <td>✔ Supported</td>
  <td>✔ Supported</td>
</tr>
</table>
```

```
```bash title="Schema Generation"
npx @better-auth/cli@latest generate
```
```

```
```bash title="Schema Migration"
npx @better-auth/cli@latest migrate
```
```

### ## Additional Information

MS SQL is supported under the hood via the [Kysely](https://kysely.dev/) adapter, any database supported by Kysely would also be supported. ([Read more here](/docs/adapters/other-relational-databases))

If you're looking for performance improvements or tips, take a look at our guide to [performance optimizations](/docs/guides/optimizing-for-performance).

```
file: ./content/docs/adapters/mysql.mdx
meta: {
  "title": "MySQL",
  "description": "Integrate Better Auth with MySQL."
}
```

MySQL is a popular open-source relational database management system (RDBMS) that is widely used for building web applications and other types of software. It provides a flexible and scalable database solution that allows for efficient storage and retrieval of data.

Read more here: [MySQL](https://www.mysql.com/).

### ## Example Usage

Make sure you have MySQL installed and configured. Then, you can connect it straight into Better Auth.

```
```ts title="auth.ts"
import { betterAuth } from "better-auth";
import { createPool } from "mysql2/promise";
```

```
export const auth = betterAuth({
  database: createPool({
    host: "localhost",
    user: "root",
    password: "password",
    database: "database",
  }),
});
```
```

#### <Callout>

For more information, read Kysely's documentation to the [MySQLDialect](https://kysely-org.github.io/kysely-apidoc/classes/MysqlDialect.html).

#### </Callout>

### ## Schema generation & migration

The [Better Auth CLI](/docs/concepts/cli) allows you to generate or migrate your database schema based on your Better Auth configuration and plugins.

```
<table>
```

```

<tr className="border-b">
  <th>
    <p className="font-bold text-[16px] mb-1">MySQL Schema Generation</p>
  </th>

  <th>
    <p className="font-bold text-[16px] mb-1">MySQL Schema Migration</p>
  </th>
</tr>

<tr className="h-10">
  <td>✔ Supported</td>
  <td>✔ Supported</td>
</tr>
</table>

```

```

```bash title="Schema Generation"
npx @better-auth/cli@latest generate
```

```

```

```bash title="Schema Migration"
npx @better-auth/cli@latest migrate
```

```

### ## Additional Information

MySQL is supported under the hood via the [Kysely](https://kysely.dev/) adapter, any database supported by Kysely would also be supported. ([Read more here](/docs/adapters/other-relational-databases))

If you're looking for performance improvements or tips, take a look at our guide to [performance optimizations](/docs/guides/optimizing-for-performance).

```

file: ./content/docs/adapters/other-relational-databases.mdx
meta: {
  "title": "Other Relational Databases",
  "description": "Integrate Better Auth with other relational databases."
}

```

Better Auth supports a wide range of database dialects out of the box thanks to [Kysely](https://kysely.dev/).

Any dialect supported by Kysely can be utilized with Better Auth, including capabilities for generating and migrating database schemas through the [CLI](/docs/concepts/cli).

### ## Core Dialects

- \* [MySQL](/docs/adapters/mysql)
- \* [SQLite](/docs/adapters/sqlite)
- \* [PostgreSQL](/docs/adapters/postgresql)
- \* [MS SQL](/docs/adapters/mssql)

### ## Kysely Organization Dialects

- \* [Postgres.js](https://github.com/kysely-org/kysely-postgres-js)
- \* [SingleStore Data API](https://github.com/kysely-org/kysely-singlestore)

### ## Kysely Community dialects

- \* [PlanetScale Serverless Driver](https://github.com/depot/kysely-planetscale)
- \* [Cloudflare D1](https://github.com/aidenwallis/kysely-d1)
- \* [AWS RDS Data API](https://github.com/serverless-stack/kysely-data-api)
- \* [SurrealDB](https://github.com/igalklebanov/kysely-surrealdb)
- \* [Neon](https://github.com/seveibar/kysely-neon)
- \* [Xata](https://github.com/xataio/client-ts/tree/main/packages/plugin-client-kysely)

- \* [AWS S3 Select](https://github.com/igalklebanov/kysely-s3-select)
- \* [libSQL/sqlid](https://github.com/libsql/kysely-libsql)
- \* [Fetch driver](https://github.com/andersgee/kysely-fetch-driver)
- \* [SQLite WASM](https://github.com/DallasHoff/sqllocal)
- \* [Deno SQLite](https://gitlab.com/soapbox-pub/kysely-deno-sqlite)
- \* [TiDB Cloud Serverless Driver](https://github.com/tidbcloud/kysely)
- \* [Capacitor SQLite Kysely](https://github.com/DawidWetzler/capacitor-sqlite-kysely)
- \* [BigQuery](https://github.com/maktouch/kysely-bigquery)
- \* [Clickhouse](https://github.com/founderpathcom/kysely-clickhouse)
- \* [PGLite](https://github.com/czeidler/kysely-pglight-dialect)

<Callout>

You can see the full list of supported Kysely dialects{" "}  
 <Link href="https://kysely.dev/docs/dialects">here</Link>.

</Callout>

file: ./content/docs/adapters/postgresql.mdx

```
meta: {
  "title": "PostgreSQL",
  "description": "Integrate Better Auth with PostgreSQL."
}
```

PostgreSQL is a powerful, open-source relational database management system known for its advanced features, extensibility, and support for complex queries and large datasets.

Read more [here](https://www.postgresql.org/).

## ## Example Usage

Make sure you have PostgreSQL installed and configured.  
 Then, you can connect it straight into Better Auth.

```
```ts title="auth.ts"
import { betterAuth } from "better-auth";
import { Pool } from "pg";

export const auth = betterAuth({
  database: new Pool({
    connectionString: "postgres://user:password@localhost:5432/database",
  }),
});
```
```

<Callout>

For more information, read Kysely's documentation to the  
 [PostgresDialect](https://kysely-org.github.io/kysely-apidoc/classes/PostgresDialect.html).

</Callout>

## ## Schema generation & migration

The [Better Auth CLI](/docs/concepts/cli) allows you to generate or migrate your database schema based on your Better Auth configuration and plugins.

```
<table>
<tr className="border-b">
  <th>
    <p className="font-bold text-[16px] mb-1">PostgreSQL Schema Generation</p>
  </th>

  <th>
    <p className="font-bold text-[16px] mb-1">PostgreSQL Schema Migration</p>
  </th>
</tr>

<tr className="h-10">
```



<td>✔ Supported</td>
<td>✔ Supported</td>

```
```bash title="Schema Generation"
npx @better-auth/cli@latest generate
```
```

```
```bash title="Schema Migration"
npx @better-auth/cli@latest migrate
```
```

### ## Additional Information

PostgreSQL is supported under the hood via the [Kysely](https://kysely.dev/) adapter, any database supported by Kysely would also be supported. (<Link href="/docs/adapters/other-relational-databases">Read more here</Link>)

If you're looking for performance improvements or tips, take a look at our guide to <Link href="/docs/guides/optimizing-for-performance">performance optimizations</Link>.

```
file: ./content/docs/adapters/prisma.mdx
meta: {
  "title": "Prisma",
  "description": "Integrate Better Auth with Prisma."
}
```

Prisma ORM is an open-source database toolkit that simplifies database access and management in applications by providing a type-safe query builder and an intuitive data modeling interface. Read more [here](https://www.prisma.io/).

### ## Example Usage

Make sure you have Prisma installed and configured. Then, you can use the Prisma adapter to connect to your database.

```
```ts title="auth.ts"
import { betterAuth } from "better-auth";
import { prismaAdapter } from "better-auth/adapters/prisma";
import { PrismaClient } from "@prisma/client";

const prisma = new PrismaClient();

export const auth = betterAuth({
  database: prismaAdapter(prisma, {
    provider: "sqlite",
  }),
});
```
```

<Callout type="warning">

If you have configured a custom output directory in your `schema.prisma` file (e.g., `output = "../src/generated/prisma"`), make sure to import the Prisma client from that location instead of `@prisma/client`. Learn more about custom output directories in the [Prisma documentation](https://www.prisma.io/docs/guides/nextjs#21-install-prisma-orm-and-create-your-first-models).

</Callout>

### ## Schema generation & migration

The [Better Auth CLI](/docs/concepts/cli) allows you to generate or migrate your database schema based on your Better Auth configuration and plugins.

|                           |
|---------------------------|
| <tr className="border-b"> |
|---------------------------|

```

<th>
  <p className="font-bold text-[16px] mb-1">Prisma Schema Generation</p>
</th>

<th>
  <p className="font-bold text-[16px] mb-1">Prisma Schema Migration</p>
</th>
</tr>

<tr className="h-10">
  <td>✅ Supported</td>
  <td>❌ Not Supported</td>
</tr>
</table>

```

```

```bash title="Schema Generation"
npx @better-auth/cli@latest generate
```

```

### ## Additional Information

If you're looking for performance improvements or tips, take a look at our guide to [performance optimizations](/docs/guides/optimizing-for-performance).

```

file: ./content/docs/adapters/sqlite.mdx
meta: {
  "title": "SQLite",
  "description": "Integrate Better Auth with SQLite."
}

```

SQLite is a lightweight, serverless, self-contained SQL database engine that is widely used for local data storage in applications.

Read more [here.](https://www.sqlite.org/)

### ## Example Usage

Make sure you have SQLite installed and configured. Then, you can connect it straight into Better Auth.

```

```ts title="auth.ts"
import { betterAuth } from "better-auth";
import Database from "better-sqlite3";

export const auth = betterAuth({
  database: new Database("database.sqlite"),
});
```

```

#### <Callout>

For more information, read Kysely's documentation to the [\[SqliteDialect\]\(https://kysely-org.github.io/kysely-apidoc/classes/SqliteDialect.html\)](https://kysely-org.github.io/kysely-apidoc/classes/SqliteDialect.html).

### ## Schema generation & migration

The [\[Better Auth CLI\]\(/docs/concepts/cli\)](/docs/concepts/cli) allows you to generate or migrate your database schema based on your Better Auth configuration and plugins.

```

<table>
  <tr className="border-b">
    <th>
      <p className="font-bold text-[16px] mb-1">SQLite Schema Generation</p>
    </th>

```

```

<th>
  <p className="font-bold text-[16px] mb-1">SQLite Schema Migration</p>
</th>
</tr>

<tr className="h-10">
  <td>✔ Supported</td>
  <td>✔ Supported</td>
</tr>
</table>

```

```

```bash title="Schema Generation"
npx @better-auth/cli@latest generate
```

```

```

```bash title="Schema Migration"
npx @better-auth/cli@latest migrate
```

```

### ## Additional Information

SQLite is supported under the hood via the [Kysely](https://kysely.dev/) adapter, any database supported by Kysely would also be supported. ([Read more here](/docs/adapters/other-relational-databases))

If you're looking for performance improvements or tips, take a look at our guide to [performance optimizations](/docs/guides/optimizing-for-performance).

```

file: ./content/docs/authentication/apple.mdx
meta: {
  "title": "Apple",
  "description": "Apple provider setup and usage."
}

```

```

<Steps>
  <Step>
    #### Get your OAuth credentials

```

To use Apple sign in, you need a client ID and client secret. You can get them from the [Apple Developer Portal] (https://developer.apple.com/account/resources/authkeys/list).

You will need an active **Apple Developer account** to access the developer portal and generate these credentials.

Follow these steps to set up your App ID, Service ID, and generate the key needed for your client secret:

#### 1. **Navigate to Certificates, Identifiers & Profiles:**

In the Apple Developer Portal, go to the "Certificates, Identifiers & Profiles" section.

#### 2. **Create an App ID:**

- \* Go to the `Identifiers` tab.

- \* Click the `+` icon next to Identifiers.

- \* Select `App IDs`, then click `Continue`.

- \* Select `App` as the type, then click `Continue`.

- \* **Description:** Enter a name for your app (e.g., "My Awesome App"). This name may be displayed to users when they sign in.

- \* **Bundle ID:** Set a bundle ID. The recommended format is a reverse domain name (e.g., `com.yourcompany.yourapp`). Using a suffix like `.ai` (for app identifier) can help with organization but is not required (e.g., `com.yourcompany.yourapp.ai`).

- \* Scroll down to **Capabilities**. Select the checkbox for `Sign In with Apple`.

- \* Click `Continue`, then `Register`.

#### 3. **Create a Service ID:**

- \* Go back to the `Identifiers` tab.

- \* Click the `+` icon.

- \* Select `Service IDs`, then click `Continue`.

- \* **Description:** Enter a description for this service (e.g., your app name again).
- \* **Identifier:** Set a unique identifier for the service. Use a reverse domain format, distinct from your App ID (e.g., `com.yourcompany.yourapp.si`, where `.si` indicates service identifier - this is for your organization and not required).
- This Service ID will be your `clientId`.**
  - \* Click `Continue`, then `Register`.

#### 4. **Configure the Service ID:**

- \* Find the Service ID you just created in the `Identifiers` list and click on it.
- \* Check the `Sign In with Apple` capability, then click `Configure`.
- \* Under **Primary App ID**, select the App ID you created earlier (e.g., `com.yourcompany.yourapp.ai`).
- \* Under **Domains and Subdomains**, list all the root domains you will use for Sign In with Apple (e.g., `example.com`, `anotherdomain.com`).
- \* Under **Return URLs**, enter the callback URL. `https://yourdomain.com/api/auth/callback/apple`. Add all necessary return URLs.
- \* Click `Next`, then `Done`.
- \* Click `Continue`, then `Save`.

#### 5. **Create a Client Secret Key:**

- \* Go to the `Keys` tab.
- \* Click the `+` icon to create a new key.
- \* **Key Name:** Enter a name for the key (e.g., "Sign In with Apple Key").
- \* Scroll down and select the checkbox for `Sign In with Apple`.
- \* Click the `Configure` button next to `Sign In with Apple`.
- \* Select the **Primary App ID** you created earlier.
- \* Click `Save`, then `Continue`, then `Register`.
- \* **Download the Key:** Immediately download the `.p8` key file. **This file is only available for download once.** Note the Key ID (available on the Keys page after creation) and your Team ID (available in your Apple Developer Account settings).

#### 6. **Generate the Client Secret (JWT):**

Apple requires a JSON Web Token (JWT) to be generated dynamically using the downloaded `.p8` key, the Key ID, and your Team ID. This JWT serves as your `clientSecret`.

You can use the guide below from Apple's documentation to understand how to generate this client secret:

```
<Link href="https://developer.apple.com/documentation/accountorganizationaldatasharing/creating-a-client-secret">
  Creating a client secret
</Link>
</Step>
```

```
<Step>
#### Configure the provider
```

To configure the provider, you need to add it to the `socialProviders` option of the auth instance.

You also need to add `https://appleid.apple.com` to the `trustedOrigins` array in your auth instance configuration to allow communication with Apple's authentication servers.

```
` ` ` ts title="auth.ts"
import { betterAuth } from "better-auth"

export const auth = betterAuth({
  socialProviders: {
    apple: { // [!code highlight]
      clientId: process.env.APPLE_CLIENT_ID as string, // [!code highlight]
      clientSecret: process.env.APPLE_CLIENT_SECRET as string, // [!code highlight]
      // Optional
      appBundleIdentifier: process.env.APPLE_APP_BUNDLE_IDENTIFIER as string, // [!code highlight]
    }, // [!code highlight]
  },
  // Add appleid.apple.com to trustedOrigins for Sign In with Apple flows
  trustedOrigins: ["https://appleid.apple.com"], // [!code highlight]
})
` ` `
```

On native iOS, it doesn't use the service ID but the app ID (bundle ID) as client ID, so if using the service ID as `clientId` in `signIn.social` with `idToken`, it throws an error: `JWTClaimValidationFailed: unexpected "aud" claim value`. So you need to provide the `appBundleIdentifier` when you want to sign in with Apple using the ID Token.

</Step>  
</Steps>

## ## Usage

### #### Sign In with Apple

To sign in with Apple, you can use the `signIn.social` function provided by the client. The `signIn` function takes an object with the following properties:

\* `provider`: The provider to use. It should be set to `apple`.

```
```ts title="auth-client.ts" /
import { createAuthClient } from "better-auth/client"
const authClient = createAuthClient()

const signIn = async () => {
  const data = await authClient.signIn.social({
    provider: "apple"
  })
}
```
```

### #### Sign In with Apple With ID Token

To sign in with Apple using the ID Token, you can use the `signIn.social` function to pass the ID Token.

This is useful when you have the ID Token from Apple on the client-side and want to use it to sign in on the server.

<Callout>  
If ID token is provided no redirection will happen, and the user will be signed in directly.  
</Callout>

```
```ts title="auth-client.ts"
await authClient.signIn.social({
  provider: "apple",
  idToken: {
    token: // Apple ID Token,
    nonce: // Nonce (optional)
    accessToken: // Access Token (optional)
  }
})
```
```

```
file: ./content/docs/authentication/discord.mdx
meta: {
  "title": "Discord",
  "description": "Discord provider setup and usage."
}
```

<Steps>  
<Step>  
#### Get your Discord credentials

To use Discord sign in, you need a client ID and client secret. You can get them from the [Discord Developer Portal] (<https://discord.com/developers/applications>).

Make sure to set the redirect URL to `http://localhost:3000/api/auth/callback/discord` for local development. For production, you should set it to the URL of your application. If you change the base path of the auth routes, you should update the redirect URL accordingly.

</Step>

<Step>  
 #### Configure the provider

To configure the provider, you need to import the provider and pass it to the `socialProviders` option of the auth instance.

```
```ts title="auth.ts"
import { betterAuth } from "better-auth"

export const auth = betterAuth({
  socialProviders: {
    discord: { // [!code highlight]
      clientId: process.env.DISCORD_CLIENT_ID as string, // [!code highlight]
      clientSecret: process.env.DISCORD_CLIENT_SECRET as string, // [!code highlight]
    }, // [!code highlight]
  },
})
```
```

</Step>

<Step>  
 #### Sign In with Discord

To sign in with Discord, you can use the `signIn.social` function provided by the client. The `signIn` function takes an object with the following properties:

\* `provider`: The provider to use. It should be set to `discord`.

```
```ts title="auth-client.ts"
import { createAuthClient } from "better-auth/client"
const authClient = createAuthClient()

const signIn = async () => {
  const data = await authClient.signIn.social({
    provider: "discord"
  })
}
```
```

</Step>  
 </Steps>

```
file: ./content/docs/authentication/dropbox.mdx
meta: {
  "title": "Dropbox",
  "description": "Dropbox provider setup and usage."
}
```

<Steps>  
 <Step>  
 #### Get your Dropbox credentials

To use Dropbox sign in, you need a client ID and client secret. You can get them from the [Dropbox Developer Portal] (<https://www.dropbox.com/developers>). You can Allow "Implicit Grant & PKCE" for the application in the App Console.

Make sure to set the redirect URL to `http://localhost:3000/api/auth/callback/dropbox` for local development. For production, you should set it to the URL of your application. If you change the base path of the auth routes, you should update the redirect URL accordingly.

</Step>

If you need deeper dive into Dropbox Authentication, you can check out the [official documentation] (<https://developers.dropbox.com/oauth-guide>).

<Step>

### Configure the provider

To configure the provider, you need to import the provider and pass it to the `socialProviders` option of the auth instance.

```
```ts title="auth.ts"
import { betterAuth } from "better-auth"

export const auth = betterAuth({
  socialProviders: {
    dropbox: { // [!code highlight]
      clientId: process.env.DROPBOX_CLIENT_ID as string, // [!code highlight]
      clientSecret: process.env.DROPBOX_CLIENT_SECRET as string, // [!code highlight]
    }, // [!code highlight]
  },
})
```
```

</Step>

<Step>

### Sign In with Dropbox

To sign in with Dropbox, you can use the `signIn.social` function provided by the client. The `signIn` function takes an object with the following properties:

\* `provider`: The provider to use. It should be set to `dropbox`.

```
```ts title="auth-client.ts"
import { createAuthClient } from "better-auth/client"
const authClient = createAuthClient()

const signIn = async () => {
  const data = await authClient.signIn.social({
    provider: "dropbox"
  })
}
```
```

</Step>

</Steps>

file: ./content/docs/authentication/email-password.mdx

```
meta: {
  "title": "Email & Password",
  "description": "Implementing email and password authentication with Better Auth."
}
```

Email and password authentication is a common method used by many applications. Better Auth provides a built-in email and password authenticator that you can easily integrate into your project.

<Callout type="info">

If you prefer username-based authentication, check out the{" "

<Link href="/docs/plugins/username">username plugin</Link>. It extends the email and password authenticator with username support.

</Callout>

## Enable Email and Password

To enable email and password authentication, you need to set the `emailAndPassword.enabled` option to `true` in the `auth` configuration.

```
```ts title="auth.ts"
import { betterAuth } from "better-auth";

export const auth = betterAuth({
```

```
emailAndPassword: { // [!code highlight]
  enabled: true, // [!code highlight]
}, // [!code highlight]
});
```,
```

```
<Callout type="info">
  If it's not enabled, it'll not allow you to sign in or sign up with email and
  password.
</Callout>
```

## ## Usage

### ### Sign Up

To sign a user up, you can use the `signUp.email` function provided by the client. The `signUp` function takes an object with the following properties:

- \* `email`: The email address of the user.
- \* `password`: The password of the user. It should be at least 8 characters long and max 128 by default.
- \* `name`: The name of the user.
- \* `image`: The image of the user. (optional)
- \* `callbackURL`: The URL to redirect to after the user verifies their email. (optional)

```
```ts title="auth-client.ts"
const { data, error } = await authClient.signUp.email({
  email: "test@example.com",
  password: "password1234",
  name: "test",
  image: "https://example.com/image.png",
});
```,
```

### ### Sign In

To sign a user in, you can use the `signIn.email` function provided by the client. The `signIn` function takes an object with the following properties:

- \* `email`: The email address of the user.
- \* `password`: The password of the user.
- \* `rememberMe`: If false, the user will be signed out when the browser is closed. (optional) (default: true)
- \* `callbackURL`: The URL to redirect to after the user signs in. (optional)

```
```ts title="auth-client.ts"
const { data, error } = await authClient.signIn.email({
  email: "test@example.com",
  password: "password1234",
});
```,
```

### ### Sign Out

To sign a user out, you can use the `signOut` function provided by the client.

```
```ts title="auth-client.ts"
await authClient.signOut();
```,
```

you can pass `fetchOptions` to redirect onSuccess

```
```ts title="auth-client.ts"
await authClient.signOut({
  fetchOptions: {
    onSuccess: () => {
      router.push("/login"); // redirect to login page
    }
  }
});
```



```

    },
  },
});
```

```

### #### Email Verification

To enable email verification, you need to pass a function that sends a verification email with a link. The `sendVerificationEmail` function takes a data object with the following properties:

- \* `user`: The user object.
- \* `url`: The URL to send to the user which contains the token.
- \* `token`: A verification token used to complete the email verification.

and a `request` object as the second parameter.

```

```ts title="auth.ts"
import { betterAuth } from "better-auth";
import { sendEmail } from "../email"; // your email sending function

export const auth = betterAuth({
  emailVerification: {
    sendVerificationEmail: async ({ user, url, token }, request) => {
      await sendEmail({
        to: user.email,
        subject: "Verify your email address",
        text: `Click the link to verify your email: ${url}`,
      });
    },
  },
});
```

```

On the client side you can use `sendVerificationEmail` function to send verification link to user. This will trigger the `sendVerificationEmail` function you provided in the `auth` configuration.

Once the user clicks on the link in the email, if the token is valid, the user will be redirected to the URL provided in the `callbackURL` parameter. If the token is invalid, the user will be redirected to the URL provided in the `callbackURL` parameter with an error message in the query string `?error=invalid\_token`.

### #### Require Email Verification

If you enable require email verification, users must verify their email before they can log in. And every time a user tries to sign in, `sendVerificationEmail` is called.

<Callout>

This only works if you have `sendVerificationEmail` implemented and if the user is trying to sign in with email and password.

</Callout>

```

```ts title="auth.ts"
export const auth = betterAuth({
  emailAndPassword: {
    requireEmailVerification: true,
  },
});
```

```

If a user tries to sign in without verifying their email, you can handle the error and show a message to the user.

```

```ts title="auth-client.ts"
await authClient.signIn.email(
  {
    email: "email@example.com",
    password: "password",
  }
);
```

```

```

    },
    {
      onError: (ctx) => {
        // Handle the error
        if (ctx.error.status === 403) {
          alert("Please verify your email address");
        }
        //you can also show the original error message
        alert(ctx.error.message);
      },
    }
  );
}

```

#### #### Triggering manually Email Verification

You can trigger the email verification manually by calling the `sendVerificationEmail` function.

```

```ts
await authClient.sendVerificationEmail({
  email: "user@email.com",
  callbackURL: "/", // The redirect URL after verification
});
```

```

#### ### Request Password Reset

To allow users to reset a password first you need to provide `sendResetPassword` function to the email and password authenticator. The `sendResetPassword` function takes a data object with the following properties:

- \* `user`: The user object.
- \* `url`: The URL to send to the user which contains the token.
- \* `token`: A verification token used to complete the password reset.

and a `request` object as the second parameter.

```

```ts title="auth.ts"
import { betterAuth } from "better-auth";
import { sendEmail } from "./email"; // your email sending function

export const auth = betterAuth({
  emailAndPassword: {
    enabled: true,
    sendResetPassword: async ({user, url, token}, request) => {
      await sendEmail({
        to: user.email,
        subject: "Reset your password",
        text: `Click the link to reset your password: ${url}`,
      });
    },
  },
});
```

```

Once you configured your server you can call `requestPasswordReset` function to send reset password link to user. If the user exists, it will trigger the `sendResetPassword` function you provided in the auth config.

It takes an object with the following properties:

- \* `email`: The email address of the user.
- \* `redirectTo`: The URL to redirect to after the user clicks on the link in the email. If the token is valid, the user will be redirected to this URL with the token in the query string. If the token is invalid, the user will be redirected to this URL with an error message in the query string `?error=invalid\_token`.

```

```ts title="auth-client.ts"

```

```
const { data, error } = await authClient.requestPasswordReset({
  email: "test@example.com",
  redirectTo: "/reset-password",
});
```,
```

When a user clicks on the link in the email, they will be redirected to the reset password page. You can add the reset password page to your app. Then you can use `resetPassword` function to reset the password. It takes an object with the following properties:

\* `newPassword`: The new password of the user.

```
```ts title="auth-client.ts"
const token = new URLSearchParams(window.location.search).get("token");
if (!token) {
  // Handle the error
}
const { data, error } = await authClient.resetPassword({
  newPassword: "password1234",
  token,
});
```,
```

### ### Update password

<Callout type="warn">

This only works on server-side, and the following code may change over time.

</Callout>

To set a password, you must hash it first:

```
```ts
const ctx = await auth.$context;
const hash = await ctx.password.hash("your-new-password");
```,
```

Then, to set the password:

```
```ts
await ctx.internalAdapter.updatePassword("userId", hash) //(you can also use your orm directly)
```,
```

### ### Configuration

#### **\*\*Password\*\***

Better Auth stores passwords inside the `account` table with `providerId` set to `credential`.

**\*\*Password Hashing\*\*:** Better Auth uses `scrypt` to hash passwords. The `scrypt` algorithm is designed to be slow and memory-intensive to make it difficult for attackers to brute force passwords. OWASP recommends using `scrypt` if `argon2id` is not available. We decided to use `scrypt` because it's natively supported by Node.js.

You can pass custom password hashing algorithm by setting `passwordHasher` option in the `auth` configuration.

```
```ts title="auth.ts"
import { betterAuth } from "better-auth"
import { scrypt } from "scrypt"

export const auth = betterAuth({
  //...rest of the options
  emailAndPassword: {
    password: {
      hash: // your custom password hashing function
      verify: // your custom password verification function
    }
  }
})
```

```

    }
  })
  ``

```

```

<TypeTable
  type={{
    enabled: {
      description: "Enable email and password authentication.",
      type: "boolean",
      default: "false",
    },
    disableSignUp: {
      description: "Disable email and password sign up.",
      type: "boolean",
      default: "false"
    },
    minPasswordLength: {
      description: "The minimum length of a password.",
      type: "number",
      default: 8,
    },
    maxPasswordLength: {
      description: "The maximum length of a password.",
      type: "number",
      default: 128,
    },
    sendResetPassword: {
      description:
        "Sends a password reset email. It takes a function that takes two parameters: token and user.",
      type: "function",
    },
    resetPasswordTokenExpiresIn: {
      description:
        "Number of seconds the reset password token is valid for.",
      type: "number",
      default: 3600
    },
    password: {
      description: "Password configuration.",
      type: "object",
      properties: {
        hash: {
          description: "custom password hashing function",
          type: "function",
        },
        verify: {
          description: "custom password verification function",
          type: "function",
        },
      },
    },
  }}
/>

```

```

file: ./content/docs/authentication/facebook.mdx
meta: {
  "title": "Facebook",
  "description": "Facebook provider setup and usage."
}

```

```

<Steps>
  <Step>
    ### Get your Facebook credentials

```

To use Facebook sign in, you need a client ID and client Secret. You can get them from the [Facebook Developer Portal] (<https://developers.facebook.com/>).

Select your app, navigate to **App Settings > Basic**, locate the following:

**App ID**: This is your `clientId`

**App Secret**: This is your `clientSecret`.

<Callout type="warn">

Avoid exposing the `clientSecret` in client-side code (e.g., frontend apps) because it's sensitive information.

</Callout>

Make sure to set the redirect URL to `http://localhost:3000/api/auth/callback/facebook` for local development. For production, you should set it to the URL of your application. If you change the base path of the auth routes, you should update the redirect URL accordingly.

</Step>

<Step>

#### Configure the provider

To configure the provider, you need to import the provider and pass it to the `socialProviders` option of the auth instance.

```
```ts title="auth.ts"
```

```
import { betterAuth } from "better-auth"
```

```
export const auth = betterAuth({
```

```
  socialProviders: {
```

```
    facebook: { // [!code highlight]
```

```
      clientId: process.env.FACEBOOK_CLIENT_ID as string, // [!code highlight]
```

```
      clientSecret: process.env.FACEBOOK_CLIENT_SECRET as string, // [!code highlight]
```

```
    }, // [!code highlight]
```

```
  },
```

```
})
```

```
```
```

<Callout>

BetterAuth also supports Facebook Login for Business, all you need

to do is provide the `configId` as listed in **Facebook Login For Business > Configurations** alongside your `clientId` and `clientSecret`. Note that the app must be a Business app and, since BetterAuth expects to have an email address and account id, the configuration must be of the "User access token" type. "System-user access token" is not supported.

</Callout>

</Step>

<Step>

#### Sign In with Facebook

To sign in with Facebook, you can use the `signIn.social` function provided by the client. The `signIn` function takes an object with the following properties:

\* `provider`: The provider to use. It should be set to `facebook`.

```
```ts title="auth-client.ts"
```

```
import { createAuthClient } from "better-auth/auth-client"
```

```
const authClient = createAuthClient()
```

```
const signIn = async () => {
```

```
  const data = await authClient.signIn.social({
```

```
    provider: "facebook"
```

```
  })
```

```
}
```

```
```
```

</Step>

</Steps>

## Additional Configuration

### #### Scopes

By default, Facebook provides basic user information. If you need additional permissions, you can specify scopes in your auth configuration:

```
```ts title="auth.ts"
export const auth = betterAuth({
  socialProviders: {
    facebook: {
      clientId: process.env.FACEBOOK_CLIENT_ID as string,
      clientSecret: process.env.FACEBOOK_CLIENT_SECRET as string,
      scopes: ["email", "public_profile", "user_friends"], // Overwrites permissions
      fields: ["user_friends"], // Extending list of fields
    },
  },
})
```
```

Additional options:

- \* `scopes`: Access basic account information (overwrites).
- \* Default: `["email", "public\_profile"]`
- \* `fields`: Extend list of fields to retrieve from the Facebook user profile (assignment).
- \* Default: `["id", "name", "email", "picture"]`

### #### Sign In with Facebook With ID or Access Token

To sign in with Facebook using the ID Token, you can use the `signIn.social` function to pass the ID Token.

This is useful when you have the ID Token from Facebook on the client-side and want to use it to sign in on the server.

<Callout>

If ID token is provided no redirection will happen, and the user will be signed in directly.

</Callout>

For limited login, you need to pass `idToken.token`, for only `accessToken` you need to pass `idToken.accessToken` and `idToken.token` together because of (#1183)\[<https://github.com/better-auth/better-auth/issues/1183>]  
(<https://github.com/better-auth/better-auth/issues/1183>).

```
```ts title="auth-client.ts"
const data = await authClient.signIn.social({
  provider: "facebook",
  idToken: { // [!code highlight]
    ...(platform === 'ios' ? // [!code highlight]
      { token: idToken } // [!code highlight]
      : { token: accessToken, accessToken: accessToken }), // [!code highlight]
  },
})
```
```

For a complete list of available permissions, refer to the [Permissions Reference] (<https://developers.facebook.com/docs/permissions>).

```
file: ./content/docs/authentication/github.mdx
meta: {
  "title": "GitHub",
  "description": "GitHub provider setup and usage."
}
```

<Steps>

<Step>

#### Get your GitHub credentials

To use GitHub sign in, you need a client ID and client secret. You can get them from the [GitHub Developer Portal] (<https://github.com/settings/developers>).

Make sure to set the redirect URL to `http://localhost:3000/api/auth/callback/github` for local development. For production, you should set it to the URL of your application. If you change the base path of the auth routes, you should update the redirect URL accordingly.

Important: You **MUST** include the user.email scope in your Github app. See details below.

</Step>

<Step>

#### Configure the provider

To configure the provider, you need to import the provider and pass it to the `socialProviders` option of the auth instance.

```
```ts title="auth.ts"
import { betterAuth } from "better-auth"

export const auth = betterAuth({
  socialProviders: {
    github: { // [!code highlight]
      clientId: process.env.GITHUB_CLIENT_ID as string, // [!code highlight]
      clientSecret: process.env.GITHUB_CLIENT_SECRET as string, // [!code highlight]
    }, // [!code highlight]
  },
})
```
```

</Step>

<Step>

#### Sign In with GitHub

To sign in with GitHub, you can use the `signIn.social` function provided by the client. The `signIn` function takes an object with the following properties:

\* `provider`: The provider to use. It should be set to `github`.

```
```ts title="auth-client.ts"
import { createAuthClient } from "better-auth/client"
const authClient = createAuthClient()

const signIn = async () => {
  const data = await authClient.signIn.social({
    provider: "github"
  })
}
```
```

</Step>

</Steps>

## Usage

#### Setting up your Github app

Github has two types of apps: Github apps and OAuth apps.

For OAuth apps, you don't have to do anything special (just follow the steps above). For Github apps, you DO have to add one more thing, which is enable it to read the user's email:

1. After creating your app, go to **\*Permissions and Events\*** > **\*Account Permissions\*** > **\*Email Addresses\*** and select "Read-Only"
2. Save changes.

That's all! Now you can copy the Client ID and Client Secret of your app!

<Callout>

If you get "email\\_not\\_found" error, it's because you selected a Github app & did not configure this part!

</Callout>

file: ./content/docs/authentication/gitlab.mdx

```
meta: {
  "title": "GitLab",
  "description": "GitLab provider setup and usage."
}
```

<Steps>

<Step>

#### Get your GitLab credentials

To use GitLab sign in, you need a client ID and client secret. [GitLab OAuth documentation] (<https://docs.gitlab.com/ee/api/oauth2.html>).

Make sure to set the redirect URL to `http://localhost:3000/api/auth/callback/gitlab` for local development. For production, you should set it to the URL of your application. If you change the base path of the auth routes, you should update the redirect URL accordingly.

</Step>

<Step>

#### Configure the provider

To configure the provider, you need to import the provider and pass it to the `socialProviders` option of the auth instance.

```
```ts title="auth.ts"
import { betterAuth } from "better-auth"

export const auth = betterAuth({
  socialProviders: {
    gitlab: { // [!code highlight]
      clientId: process.env.GITLAB_CLIENT_ID as string, // [!code highlight]
      clientSecret: process.env.GITLAB_CLIENT_SECRET as string, // [!code highlight]
      issuer: process.env.GITLAB_ISSUER as string, // [!code highlight]
    }, // [!code highlight]
  },
})
```
```

</Step>

<Step>

#### Sign In with GitLab

To sign in with GitLab, you can use the `signIn.social` function provided by the client. The `signIn` function takes an object with the following properties:

\* `provider`: The provider to use. It should be set to `gitlab`.

```
```ts title="auth-client.ts"
import { createAuthClient } from "better-auth/client"
const authClient = createAuthClient()

const signIn = async () => {
  const data = await authClient.signIn.social({
    provider: "gitlab"
  })
}
```
```

</Step>



</Steps>

```
file: ./content/docs/authentication/google.mdx
meta: {
  "title": "Google",
  "description": "Google provider setup and usage."
}
```

<Steps>

<Step>

#### Get your Google credentials

To use Google as a social provider, you need to get your Google credentials. You can get them by creating a new project in the [Google Cloud Console](https://console.cloud.google.com/apis/dashboard).

In the Google Cloud Console > Credentials > Authorized redirect URIs, make sure to set the redirect URL to `http://localhost:3000/api/auth/callback/google` for local development. For production, make sure to set the redirect URL as your application domain, e.g. `https://example.com/api/auth/callback/google`. If you change the base path of the auth routes, you should update the redirect URL accordingly.

</Step>

<Step>

#### Configure the provider

To configure the provider, you need to pass the `clientId` and `clientSecret` to `socialProviders.google` in your auth configuration.

```
```ts title="auth.ts"
import { betterAuth } from "better-auth"

export const auth = betterAuth({
  socialProviders: {
    google: { // [!code highlight]
      clientId: process.env.GOOGLE_CLIENT_ID as string, // [!code highlight]
      clientSecret: process.env.GOOGLE_CLIENT_SECRET as string, // [!code highlight]
    }, // [!code highlight]
  },
})
```
```

</Step>

</Steps>

## Usage

#### Sign In with Google

To sign in with Google, you can use the `signIn.social` function provided by the client. The `signIn` function takes an object with the following properties:

\* `provider`: The provider to use. It should be set to `google`.

```
```ts title="auth-client.ts" /
import { createAuthClient } from "better-auth/client"
const authClient = createAuthClient()

const signIn = async () => {
  const data = await authClient.signIn.social({
    provider: "google"
  })
}
```
```

#### Sign In with Google With ID Token

To sign in with Google using the ID Token, you can use the `signIn.social` function to pass the ID Token.

This is useful when you have the ID Token from Google on the client-side and want to use it to sign in on the server.

<Callout>

If ID token is provided no redirection will happen, and the user will be signed in directly.

</Callout>

```
```ts title="auth-client.ts"
const data = await authClient.signIn.social({
  provider: "google",
  idToken: {
    token: // Google ID Token,
    accessToken: // Google Access Token
  }
})
```
```

<Callout>

If you want to use google one tap, you can use the [One Tap Plugin](/docs/plugins/one-tap) guide.

</Callout>

### ### Always ask to select an account

If you want to always ask the user to select an account, you pass the `prompt` parameter to the provider, setting it to `select\_account`.

```
```ts
socialProviders: {
  google: {
    prompt: "select_account", // [!code highlight]
    clientId: process.env.GOOGLE_CLIENT_ID as string,
    clientSecret: process.env.GOOGLE_CLIENT_SECRET as string,
  },
}
```
```

### ### Requesting Additional Google Scopes

If your application needs additional Google scopes after the user has already signed up (e.g., for Google Drive, Gmail, or other Google services), you can request them using the `linkSocial` method with the same Google provider.

```
```ts title="auth-client.ts"
const requestGoogleDriveAccess = async () => {
  await authClient.linkSocial({
    provider: "google",
    scopes: ["https://www.googleapis.com/auth/drive.file"],
  });
};

// Example usage in a React component
return <button onClick={requestGoogleDriveAccess}>Add Google Drive Permissions</button>;
```
```

This will trigger a new OAuth flow that requests the additional scopes. After completion, your account will have the new scope in the database, and the access token will give you access to the requested Google APIs.

<Callout>

Ensure you're using Better Auth version 1.2.7 or later to avoid "Social account already linked" errors when requesting additional scopes from the same provider.

</Callout>

```
file: ./content/docs/authentication/kick.mdx
meta: {
```

```
"title": "Kick",
"description": "Kick provider setup and usage."
}
```

<Steps>

<Step>

#### Get your Kick Credentials

To use Kick sign in, you need a client ID and client secret. You can get them from the [Kick Developer Portal] (<https://kick.com/settings/developer>).

Make sure to set the redirect URL to `http://localhost:3000/api/auth/callback/kick` for local development. For production, you should set it to the URL of your application. If you change the base path of the auth routes, you should update the redirect URL accordingly.

</Step>

<Step>

#### Configure the provider

To configure the provider, you need to import the provider and pass it to the `socialProviders` option of the auth instance.

```
` ` ` ts title="auth.ts"
import { betterAuth } from "better-auth"

export const auth = betterAuth({
  socialProviders: {
    kick: { // [!code highlight]
      clientId: process.env.KICK_CLIENT_ID as string, // [!code highlight]
      clientSecret: process.env.KICK_CLIENT_SECRET as string, // [!code highlight]
    }, // [!code highlight]
  }
})
` ` `
```

</Step>

<Step>

#### Sign In with Kick

To sign in with Kick, you can use the `signIn.social` function provided by the client. The `signIn` function takes an object with the following properties:

\* `provider`: The provider to use. It should be set to `kick`.

```
` ` ` ts title="auth-client.ts"
import { createAuthClient } from "better-auth/client"
const authClient = createAuthClient()

const signIn = async () => {
  const data = await authClient.signIn.social({
    provider: "kick"
  })
}
` ` `
```

</Step>

</Steps>

file: ./content/docs/authentication/linkedin.mdx

```
meta: {
  "title": "LinkedIn",
  "description": "LinkedIn Provider"
}
```

<Steps>

<Step>  
 #### Get your LinkedIn credentials

To use LinkedIn sign in, you need a client ID and client secret. You can get them from the [LinkedIn Developer Portal] (<https://www.linkedin.com/developers/>).

Make sure to set the redirect URL to `http://localhost:3000/api/auth/callback/linkedin` for local development. For production, you should set it to the URL of your application. If you change the base path of the auth routes, you should update the redirect URL accordingly.

</Step>

<Callout type="info">

In the LinkedIn portal under products you need the **\*\*Sign In with LinkedIn using OpenID Connect\*\*** product.

</Callout>

There are some different Guides here:

[Authorization Code Flow (3-legged OAuth) (Outdated)](<https://learn.microsoft.com/en-us/linkedin/shared/authentication/authorization-code-flow>)

[Sign In with LinkedIn using OpenID Connect](<https://learn.microsoft.com/en-us/linkedin/consumer/integrations/self-serve/sign-in-with-linkedin-v2?context=linkedin%2Fconsumer%2Fcontext>)

<Step>  
 #### Configure the provider

To configure the provider, you need to import the provider and pass it to the `socialProviders` option of the auth instance.

```
```ts title="auth.ts"
import { betterAuth } from "better-auth"

export const auth = betterAuth({
  socialProviders: {
    linkedin: { // [!code highlight]
      clientId: process.env.LINKEDIN_CLIENT_ID as string, // [!code highlight]
      clientSecret: process.env.LINKEDIN_CLIENT_SECRET as string, // [!code highlight]
    }, // [!code highlight]
  },
})
```
```

</Step>

<Step>  
 #### Sign In with LinkedIn

To sign in with LinkedIn, you can use the `signIn.social` function provided by the client. The `signIn` function takes an object with the following properties:

\* `provider`: The provider to use. It should be set to `linkedin`.

```
```ts title="auth-client.ts"
import { createAuthClient } from "better-auth/client"
const authClient = createAuthClient()

const signIn = async () => {
  const data = await authClient.signIn.social({
    provider: "linkedin"
  })
}
```
```

</Step>

</Steps>

file: ./content/docs/authentication/microsoft.mdx  
 meta: {

```
"title": "Microsoft",
"description": "Microsoft provider setup and usage."
}
```

Enabling OAuth with Microsoft Azure Entra ID (formerly Active Directory) allows your users to sign in and sign up to your application with their Microsoft account.

<Steps>

<Step>

#### Get your Microsoft credentials

To use Microsoft as a social provider, you need to get your Microsoft credentials. Which involves generating your own Client ID and Client Secret using your Microsoft Entra ID dashboard account.

Make sure to set the redirect URL to `http://localhost:3000/api/auth/callback/microsoft` for local development. For production, you should change it to the URL of your application. If you change the base path of the auth routes, you should update the redirect URL accordingly.

see the [Microsoft Entra ID documentation](https://docs.microsoft.com/en-us/azure/active-directory/develop/quickstart-register-app) for more information.

</Step>

<Step>

#### Configure the provider

To configure the provider, you need to pass the `clientId` and `clientSecret` to `socialProviders.microsoft` in your auth configuration.

```
` `` ts title="auth.ts"
import { betterAuth } from "better-auth"

export const auth = betterAuth({
  socialProviders: {
    microsoft: { // [!code highlight]
      clientId: process.env.MICROSOFT_CLIENT_ID as string, // [!code highlight]
      clientSecret: process.env.MICROSOFT_CLIENT_SECRET as string, // [!code highlight]
      // Optional
      tenantId: 'common', // [!code highlight]
      prompt: "select_account", // Forces account selection // [!code highlight]
    }, // [!code highlight]
  },
})
` ``

</Step>
</Steps>
```

## Sign In with Microsoft

To sign in with Microsoft, you can use the `signIn.social` function provided by the client. The `signIn` function takes an object with the following properties:

\* `provider`: The provider to use. It should be set to `microsoft`.

```
` `` ts title="auth-client.ts" /
import { createAuthClient } from "better-auth/client";

const authClient = createAuthClient();

const signIn = async () => {
  const data = await authClient.signIn.social({
    provider: "microsoft",
    callbackURL: "/dashboard", // The URL to redirect to after the sign in
  });
};
` ``
```

```
file: ./content/docs/authentication/other-social-providers.mdx
meta: {
  "title": "Other Social Providers",
  "description": "Other social providers setup and usage."
}
```

Better Auth provides out of the box support for the [Generic OAuth Plugin](/docs/plugins/generic-oauth) which allows you to use any social provider that implements the OAuth2 protocol or OpenID Connect (OIDC) flows.

To use a provider that is not supported out of the box, you can use the [Generic OAuth Plugin](/docs/plugins/generic-oauth).

## ## Installation

### <Steps>

#### <Step>

#### Add the plugin to your auth config

To use the *Generic OAuth* plugin, add it to your auth config.

```
```ts title="auth.ts"
import { betterAuth } from "better-auth"
import { genericOAuth } from "better-auth/plugins" // [!code highlight]

export const auth = betterAuth({
  // ... other config options
  plugins: [
    genericOAuth({ // [!code highlight]
      config: [ // [!code highlight]
        { // [!code highlight]
          providerId: "provider-id", // [!code highlight]
          clientId: "test-client-id", // [!code highlight]
          clientSecret: "test-client-secret", // [!code highlight]
          discoveryUrl: "https://auth.example.com/.well-known/openid-configuration", // [!code highlight]
          // ... other config options // [!code highlight]
        }, // [!code highlight]
        // Add more providers as needed // [!code highlight]
      ] // [!code highlight]
    }) // [!code highlight]
  ]
})
```
```

#### </Step>

#### <Step>

#### Add the client plugin

Include the *Generic OAuth* client plugin in your authentication client instance.

```
```ts title="auth-client.ts"
import { createAuthClient } from "better-auth/client"
import { genericOAuthClient } from "better-auth/client/plugins"

const authClient = createAuthClient({
  plugins: [
    genericOAuthClient()
  ]
})
```
```

#### </Step>

### </Steps>

### <Callout>

Read more about installation and usage of the *Generic OAuth* plugin

```
[here](/docs/plugins/generic-oauth#usage).
</Callout>
```

### ## Example usage

#### #### Slack Example

```
```ts title="auth.ts"
import { betterAuth } from "better-auth";
import { genericOAuth } from "better-auth/plugins";

export const auth = betterAuth({
  // ... other config options
  plugins: [
    genericOAuth({
      config: [
        {
          providerId: "slack",
          clientId: process.env.SLACK_CLIENT_ID as string,
          clientSecret: process.env.SLACK_CLIENT_SECRET as string,
          authorizationUrl: "https://slack.com/oauth/v2/authorize",
          tokenUrl: "https://slack.com/api/oauth.v2.access",
          scopes: ["users:read", "users:read.email"], // and more...
        },
      ],
    }),
  ],
});
```

```ts title="sign-in.ts"
const response = await authClient.signIn.oauth2({
  providerId: "slack",
  callbackURL: "/dashboard", // the path to redirect to after the user is authenticated
});
```
```

#### #### Instagram Example

```
```ts title="auth.ts"
import { betterAuth } from "better-auth";
import { genericOAuth } from "better-auth/plugins";

export const auth = betterAuth({
  // ... other config options
  plugins: [
    genericOAuth({
      config: [
        {
          providerId: "instagram",
          clientId: process.env.INSTAGRAM_CLIENT_ID as string,
          clientSecret: process.env.INSTAGRAM_CLIENT_SECRET as string,
          authorizationUrl: "https://api.instagram.com/oauth/authorize",
          tokenUrl: "https://api.instagram.com/oauth/access_token",
          scopes: ["user_profile", "user_media"],
        },
      ],
    }),
  ],
});
```

```ts title="sign-in.ts"
const response = await authClient.signIn.oauth2({
  providerId: "instagram",
});
```
```

```

    callbackURL: "/dashboard", // the path to redirect to after the user is authenticated
  });
  ``

```

### ### Coinbase Example

```

` `` ts title="auth.ts"
import { betterAuth } from "better-auth";
import { genericOAuth } from "better-auth/plugins";

export const auth = betterAuth({
  // ... other config options
  plugins: [
    genericOAuth({
      config: [
        {
          providerId: "coinbase",
          clientId: process.env.COINBASE_CLIENT_ID as string,
          clientSecret: process.env.COINBASE_CLIENT_SECRET as string,
          authorizationUrl: "https://www.coinbase.com/oauth/authorize",
          tokenUrl: "https://api.coinbase.com/oauth/token",
          scopes: ["wallet:user:read"], // and more...
        },
      ],
    }),
  ],
});
` ``

` `` ts title="sign-in.ts"
const response = await authClient.signIn.oauth2({
  providerId: "coinbase",
  callbackURL: "/dashboard", // the path to redirect to after the user is authenticated
});
` ``

```

```

file: ./content/docs/authentication/reddit.mdx
meta: {
  "title": "Reddit",
  "description": "Reddit provider setup and usage."
}

```

<Steps>

<Step>

### Get your Reddit Credentials

To use Reddit sign in, you need a client ID and client secret. You can get them from the [Reddit Developer Portal] (<https://www.reddit.com/prefs/apps>).

1. Click "Create App" or "Create Another App"
2. Select "web app" as the application type
3. Set the redirect URL to `http://localhost:3000/api/auth/callback/reddit` for local development
4. For production, set it to your application's domain (e.g. `https://example.com/api/auth/callback/reddit`)
5. After creating the app, you'll get the client ID (under the app name) and client secret

If you change the base path of the auth routes, make sure to update the redirect URL accordingly.

</Step>

<Step>

### Configure the provider

To configure the provider, you need to import the provider and pass it to the `socialProviders` option of the auth instance.



```

```ts title="auth.ts"
import { betterAuth } from "better-auth"

export const auth = betterAuth({
  socialProviders: {
    reddit: {
      clientId: process.env.REDDIT_CLIENT_ID as string,
      clientSecret: process.env.REDDIT_CLIENT_SECRET as string,
    },
  },
})
```
</Step>

```

```

<Step>
#### Sign In with Reddit

```

To sign in with Reddit, you can use the `signIn.social` function provided by the client. The `signIn` function takes an object with the following properties:

\* `provider`: The provider to use. It should be set to `reddit`.

```

```ts title="auth-client.ts"
import { createAuthClient } from "better-auth/client"
const authClient = createAuthClient()

const signIn = async () => {
  const data = await authClient.signIn.social({
    provider: "reddit"
  })
}
```
</Step>
</Steps>

```

## ## Additional Configuration

### #### Scopes

By default, Reddit provides basic user information. If you need additional permissions, you can specify scopes in your auth configuration:

```

```ts title="auth.ts"
export const auth = betterAuth({
  socialProviders: {
    reddit: {
      clientId: process.env.REDDIT_CLIENT_ID as string,
      clientSecret: process.env.REDDIT_CLIENT_SECRET as string,
      duration: "permanent",
      scope: ["read", "submit"] // Add required scopes
    },
  },
})
```

```

Common Reddit scopes include:

- \* `identity`: Access basic account information
- \* `read`: Access posts and comments
- \* `submit`: Submit posts and comments
- \* `subscribe`: Manage subreddit subscriptions
- \* `history`: Access voting history

For a complete list of available scopes, refer to the [Reddit OAuth2 documentation](https://www.reddit.com/dev/api/oauth).

```
file: ./content/docs/authentication/roblox.mdx
meta: {
  "title": "Roblox",
  "description": "Roblox provider setup and usage."
}
```

<Steps>

<Step>

#### Get your Roblox Credentials

Get your Roblox credentials from the [Roblox Creator Hub](https://create.roblox.com/dashboard/credentials?activeTab=OAuthTab).

Make sure to set the redirect URL to `http://localhost:3000/api/auth/callback/roblox` for local development. For production, you should set it to the URL of your application. If you change the base path of the auth routes, you should update the redirect URL accordingly.

<Callout type="info">

The Roblox API does not provide email addresses. As a workaround, the user's `email` field uses the `preferred\_username` value instead.

</Callout>

</Step>

<Step>

#### Configure the provider

To configure the provider, you need to import the provider and pass it to the `socialProviders` option of the auth instance.

```
```ts title="auth.ts"
```

```
import { betterAuth } from "better-auth"
```

```
export const auth = betterAuth({
```

```
  socialProviders: {
```

```
    roblox: { // [!code highlight]
```

```
      clientId: process.env.ROBLOX_CLIENT_ID as string, // [!code highlight]
```

```
      clientSecret: process.env.ROBLOX_CLIENT_SECRET as string, // [!code highlight]
```

```
    }, // [!code highlight]
```

```
  },
```

```
})
```

```
```
```

</Step>

<Step>

#### Sign In with Roblox

To sign in with Roblox, you can use the `signIn.social` function provided by the client. The `signIn` function takes an object with the following properties:

\* `provider`: The provider to use. It should be set to `roblox`.

```
```ts title="auth-client.ts"
```

```
import { createAuthClient } from "better-auth/client"
```

```
const authClient = createAuthClient()
```

```
const signIn = async () => {
```

```
  const data = await authClient.signIn.social({
```

```
    provider: "roblox"
```

```
  })
```

```
}
```

```
```
```

</Step>

</Steps>

```
file: ./content/docs/authentication/spotify.mdx
meta: {
  "title": "Spotify",
  "description": "Spotify provider setup and usage."
}
```

<Steps>

<Step>

#### Get your Spotify Credentials

To use Spotify sign in, you need a client ID and client secret. You can get them from the [Spotify Developer Portal] (<https://developer.spotify.com/dashboard/applications>).

Make sure to set the redirect URL to `http://localhost:3000/api/auth/callback/spotify` for local development. For production, you should set it to the URL of your application. If you change the base path of the auth routes, you should update the redirect URL accordingly.

</Step>

<Step>

#### Configure the provider

To configure the provider, you need to import the provider and pass it to the `socialProviders` option of the auth instance.

```
```ts title="auth.ts"
import { betterAuth } from "better-auth"

export const auth = betterAuth({
  socialProviders: {
    spotify: { // [!code highlight]
      clientId: process.env.SPOTIFY_CLIENT_ID as string, // [!code highlight]
      clientSecret: process.env.SPOTIFY_CLIENT_SECRET as string, // [!code highlight]
    }, // [!code highlight]
  },
})
```
```

</Step>

<Step>

#### Sign In with Spotify

To sign in with Spotify, you can use the `signIn.social` function provided by the client. The `signIn` function takes an object with the following properties:

\* `provider`: The provider to use. It should be set to `spotify`.

```
```ts title="auth-client.ts" /
import { createAuthClient } from "better-auth/client"
const authClient = createAuthClient()

const signIn = async () => {
  const data = await authClient.signIn.social({
    provider: "spotify"
  })
}
```
```

</Step>

</Steps>

```
file: ./content/docs/authentication/tiktok.mdx
meta: {
  "title": "TikTok",
```

```
"description": "TikTok provider setup and usage."
}
```

<Steps>

<Step>

#### Get your TikTok Credentials

To integrate with TikTok, you need to obtain API credentials by creating an application in the [TikTok Developer Portal] (<https://developers.tiktok.com/apps>).

Follow these steps:

1. Create an account on the TikTok Developer Portal
2. Create a new application
3. Set up a sandbox environment for testing
4. Configure your redirect URL (must be HTTPS)
5. Note your Client ID, Client Secret and Client Key

<Callout type="info">

\* The TikTok API does not work with localhost. You need to use a public domain for the redirect URL and HTTPS for local testing. You can use [NGROK](<https://ngrok.com/>) or another similar tool for this.

\* For testing, you will need to use the [Sandbox mode](<https://developers.tiktok.com/blog/introducing-sandbox>), which you can enable in the TikTok Developer Portal.

\* The default scope is `user.info.profile`. For additional scopes, refer to the [Available Scopes] (<https://developers.tiktok.com/doc/tiktok-api-scopes/>) documentation.

</Callout>

Make sure to set the redirect URL to a valid HTTPS domain for local development. For production, you should set it to the URL of your application. If you change the base path of the auth routes, you should update the redirect URL accordingly.

<Callout type="info">

\* The TikTok API does not provide email addresses. As a workaround, this implementation uses the user's `username` value for the `email` field, which is why it requires the `user.info.profile` scope instead of just `user.info.basic`.

\* For production use, you will need to request approval from TikTok for the scopes you intend to use.

</Callout>

</Step>

<Step>

#### Configure the provider

To configure the provider, you need to import the provider and pass it to the `socialProviders` option of the auth instance.

```
` ` ` ts title="auth.ts"
```

```
import { betterAuth } from "better-auth"
```

```
export const auth = betterAuth({
```

```
  socialProviders: {
```

```
    tiktok: { // [!code highlight]
```

```
      clientId: process.env.TIKTOK_CLIENT_ID as string, // [!code highlight]
```

```
      clientSecret: process.env.TIKTOK_CLIENT_SECRET as string, // [!code highlight]
```

```
      clientKey: process.env.TIKTOK_CLIENT_KEY as string, // [!code highlight]
```

```
    }, // [!code highlight]
```

```
  },
```

```
})
```

```
` ` `
```

</Step>

<Step>

#### Sign In with TikTok

To sign in with TikTok, you can use the `signIn.social` function provided by the client. The `signIn` function takes an object with the following properties:

\* `provider`: The provider to use. It should be set to `tiktok`.

```

` `` ts title="auth-client.ts"
import { createAuthClient } from "better-auth/client"
const authClient = createAuthClient()

const signIn = async () => {
  const data = await authClient.signIn.social({
    provider: "tiktok"
  })
}
` ``
</Step>
</Steps>

```

```

file: ./content/docs/authentication/twitch.mdx
meta: {
  "title": "Twitch",
  "description": "Twitch provider setup and usage."
}

```

```

<Steps>
<Step>
  #### Get your Twitch Credentials

```

To use Twitch sign in, you need a client ID and client secret. You can get them from the [Twitch Developer Portal] (<https://dev.twitch.tv/console/apps>).

Make sure to set the redirect URL to `http://localhost:3000/api/auth/callback/twitch` for local development. For production, you should set it to the URL of your application. If you change the base path of the auth routes, you should update the redirect URL accordingly.

```

</Step>

```

```

<Step>
  #### Configure the provider

```

To configure the provider, you need to import the provider and pass it to the `socialProviders` option of the auth instance.

```

` `` ts title="auth.ts"
import { betterAuth } from "better-auth"

export const auth = betterAuth({
  socialProviders: {
    twitch: { // [!code highlight]
      clientId: process.env.TWITCH_CLIENT_ID as string, // [!code highlight]
      clientSecret: process.env.TWITCH_CLIENT_SECRET as string, // [!code highlight]
    }, // [!code highlight]
  }
})
` ``
</Step>

```

```

<Step>
  #### Sign In with Twitch

```

To sign in with Twitch, you can use the `signIn.social` function provided by the client. The `signIn` function takes an object with the following properties:

\* `provider`: The provider to use. It should be set to `twitch`.

```

` `` ts title="auth-client.ts"
import { createAuthClient } from "better-auth/client"
const authClient = createAuthClient()

```

```
const signIn = async () => {
  const data = await authClient.signIn.social({
    provider: "twitch"
  })
}
...
</Step>
</Steps>
```

```
file: ./content/docs/authentication/twitter.mdx
meta: {
  "title": "Twitter (X)",
  "description": "Twitter provider setup and usage."
}
```

```
<Steps>
<Step>
  #### Get your Twitter Credentials
```

Get your Twitter credentials from the [Twitter Developer Portal](https://developer.twitter.com/en/portal/dashboard).

Make sure to set the redirect URL to `http://localhost:3000/api/auth/callback/twitter` for local development. For production, you should set it to the URL of your application. If you change the base path of the auth routes, you should update the redirect URL accordingly.

```
<Callout type="info">
  Twitter API v2 now supports email address retrieval. Make sure to request the `user.email` scope when configuring your
  Twitter app to enable this feature.
</Callout>
</Step>
```

```
<Step>
  #### Configure the provider
```

To configure the provider, you need to import the provider and pass it to the `socialProviders` option of the auth instance.

```
```ts title="auth.ts"
import { betterAuth } from "better-auth"

export const auth = betterAuth({
  socialProviders: {
    twitter: { // [!code highlight]
      clientId: process.env.TWITTER_CLIENT_ID as string, // [!code highlight]
      clientSecret: process.env.TWITTER_CLIENT_SECRET as string, // [!code highlight]
    }, // [!code highlight]
  },
})
```
</Step>
```

```
<Step>
  #### Sign In with Twitter
```

To sign in with Twitter, you can use the `signIn.social` function provided by the client. The `signIn` function takes an object with the following properties:

\* `provider`: The provider to use. It should be set to `twitter`.

```
```ts title="auth-client.ts"
import { createAuthClient } from "better-auth/client"
const authClient = createAuthClient()

const signIn = async () => {
```

```

    const data = await authClient.signIn.social({
      provider: "twitter"
    })
  }
  ...
</Step>
</Steps>

```

```

file: ./content/docs/authentication/vk.mdx
meta: {
  "title": "VK",
  "description": "VK ID Provider"
}

```

```

<Steps>
<Step>
  #### Get your VK ID credentials

```

To use VK ID sign in, you need a client ID and client secret. You can get them from the [VK ID Developer Portal] (<https://id.vk.com/about/business/go/docs>).

Make sure to set the redirect URL to `http://localhost:3000/api/auth/callback/vk` for local development. For production, you should set it to the URL of your application. If you change the base path of the auth routes, you should update the redirect URL accordingly.

```

</Step>

```

```

<Step>
  #### Configure the provider

```

To configure the provider, you need to import the provider and pass it to the `socialProviders` option of the auth instance.

```

```ts title="auth.ts"
import { betterAuth } from "better-auth";

export const auth = betterAuth({
  socialProviders: {
    vk: { // [!code highlight]
      clientId: process.env.VK_CLIENT_ID as string, // [!code highlight]
      clientSecret: process.env.VK_CLIENT_SECRET as string, // [!code highlight]
    },
  },
});
```
</Step>

```

```

<Step>
  #### Sign In with VK

```

To sign in with VK, you can use the `signIn.social` function provided by the client. The `signIn` function takes an object with the following properties:

\* `provider`: The provider to use. It should be set to `vk`.

```

```ts title="auth-client.ts"
import { createAuthClient } from "better-auth/client";
const authClient = createAuthClient();

const signIn = async () => {
  const data = await authClient.signIn.social({
    provider: "vk",
  });
};
```

```

```
</Step>
</Steps>
```

```
file: ./content/docs/authentication/zoom.mdx
meta: {
  "title": "Zoom",
  "description": "Zoom provider setup and usage."
}
```

```
<Steps>
<Step>
  ### Create a Zoom App from Marketplace

  1. Visit [Zoom Marketplace](https://marketplace.zoom.us).

  2. Hover on the `Develop` button and select `Build App`

  3. Select `General App` and click `Create`
</Step>
```

```
<Step>
  ### Configure your Zoom App

  Ensure that you are in the `Basic Information` of your app settings.

  1. Under `Select how the app is managed`, choose `User-managed`

  2. Under `App Credentials`, copy your `Client ID` and `Client Secret` and store them in a safe location

  3. Under `OAuth Information` -> `OAuth Redirect URL`, add your Callback URL. For example,

  ```
  http://localhost:3000/api/auth/callback/zoom
  ```
```

```
<Callout>
  For production, you should set it to the URL of your application. If you change the base
  path of the auth routes, you should update the redirect URL accordingly.
</Callout>
```

Skip to the `Scopes` section, then

```
1. Click the `Add Scopes` button
2. Search for `user:read:user` (View a user) and select it
3. Add any other scopes your applications needs and click `Done`
</Step>
```

```
<Step>
  ### Configure the provider
```

To configure the provider, you need to import the provider and pass it to the `socialProviders` option of the auth instance.

```
```ts title="auth.ts"
import { betterAuth } from "better-auth"

export const auth = betterAuth({
  socialProviders: {
    zoom: { // [!code highlight]
      clientId: process.env.ZOOM_CLIENT_ID as string, // [!code highlight]
      clientSecret: process.env.ZOOM_CLIENT_SECRET as string, // [!code highlight]
    }, // [!code highlight]
  },
})
```



```

` ``
</Step>

<Step>
#### Sign In with Zoom

To sign in with Zoom, you can use the `signIn.social` function provided by the client.
You will need to specify `zoom` as the provider.

` `` ts title="auth-client.ts"
import { createAuthClient } from "better-auth/client"
const authClient = createAuthClient()

const signIn = async () => {
  const data = await authClient.signIn.social({
    provider: "zoom"
  })
}
` ``
</Step>
</Steps>

```

```

file: ./content/docs/concepts/api.mdx
meta: {
  "title": "API",
  "description": "Better Auth API."
}

```

When you create a new Better Auth instance, it provides you with an `api` object. This object exposes every endpoint that exist in your Better Auth instance. And you can use this to interact with Better Auth server side.

Any endpoint added to Better Auth, whether from plugins or the core, will be accessible through the `api` object.

### ## Calling API Endpoints on the Server

To call an API endpoint on the server, import your `auth` instance and call the endpoint using the `api` object.

```

` `` ts title="server.ts"
import { betterAuth } from "better-auth";
import { headers } from "next/headers";

export const auth = betterAuth({
  //...
})

// calling get session on the server
await auth.api.getSession({
  headers: await headers() // some endpoint might require headers
})
` ``

```

### ### Body, Headers, Query

Unlike the client, the server needs the values to be passed as an object with the key `body` for the body, `headers` for the headers, and `query` for query parameters.

```

` `` ts title="server.ts"
await auth.api.getSession({
  headers: await headers()
})

await auth.api.signInEmail({
  body: {
    email: "john@doe.com",

```

```

    password: "password"
  },
  headers: await headers() // optional but would be useful to get the user IP, user agent, etc.
})

await auth.api.verifyEmail({
  query: {
    token: "my_token"
  }
})
```,

```

#### <Callout>

Better auth API endpoints are built on top of [better-call](https://github.com/bekacru/better-call), a tiny web framework that lets you call REST API endpoints as if they were regular functions and allows us to easily infer client types from the server.

#### </Callout>

### ### Getting `headers` and `Response` Object

When you invoke an API endpoint on the server, it will return a standard JavaScript object or array directly as it's just a regular function call.

But there are times where you might want to get the `headers` or the `Response` object instead. For example, if you need to get the cookies or the headers.

#### #### Getting `headers`

To get the `headers`, you can pass the `returnHeaders` option to the endpoint.

```

```ts
const { headers, response } = await auth.api.signUpEmail({
  returnHeaders: true,
  body: {
    email: "john@doe.com",
    password: "password",
    name: "John Doe",
  },
});
```,

```

The `headers` will be a `Headers` object. Which you can use to get the cookies or the headers.

```

```ts
const cookies = headers.get("set-cookie");
const headers = headers.get("x-custom-header");
```,

```

#### #### Getting `Response` Object

To get the `Response` object, you can pass the `asResponse` option to the endpoint.

```

```ts title="server.ts"
const response = await auth.api.signInEmail({
  body: {
    email: "",
    password: ""
  },
  asResponse: true
})
```,

```

### ### Error Handling

When you call an API endpoint in the server, it will throw an error if the request fails. You can catch the error and handle it

as you see fit. The error instance is an instance of `APIError`.

```
```ts title="server.ts"
import { APIError } from "better-auth/api";

try {
  await auth.api.signInEmail({
    body: {
      email: "",
      password: ""
    }
  })
} catch (error) {
  if (error instanceof APIError) {
    console.log(error.message, error.status)
  }
}
```
```

```
file: ./content/docs/concepts/cli.mdx
meta: {
  "title": "CLI",
  "description": "Built-in CLI for managing your project."
}
```

Better Auth comes with a built-in CLI to help you manage the database schemas, initialize your project, and generate a secret key for your application.

### ## Generate

The `generate` command creates the schema required by Better Auth. If you're using a database adapter like Prisma or Drizzle, this command will generate the right schema for your ORM. If you're using the built-in Kysely adapter, it will generate an SQL file you can run directly on your database.

```
```bash title="Terminal"
npx @better-auth/cli@latest generate
```
```

### ### Options

\* `--output` - Where to save the generated schema. For Prisma, it will be saved in prisma/schema.prisma. For Drizzle, it goes to schema.ts in your project root. For Kysely, it's an SQL file saved as schema.sql in your project root.  
 \* `--config` - The path to your Better Auth config file. By default, the CLI will search for a auth.ts file in `\*\*./\*\*`, `\*\*./utils\*\*`, `\*\*./lib\*\*`, or any of these directories under `src` directory.  
 \* `--y` - Skip the confirmation prompt and generate the schema directly.

### ## Migrate

The migrate command applies the Better Auth schema directly to your database. This is available if you're using the built-in Kysely adapter. For other adapters, you'll need to apply the schema using your ORM's migration tool.

```
```bash title="Terminal"
npx @better-auth/cli@latest migrate
```
```

### ### Options

\* `--config` - The path to your Better Auth config file. By default, the CLI will search for a auth.ts file in `\*\*./\*\*`, `\*\*./utils\*\*`, `\*\*./lib\*\*`, or any of these directories under `src` directory.  
 \* `--y` - Skip the confirmation prompt and apply the schema directly.

### ## Init

The `init` command allows you to initialize Better Auth in your project.

```
```bash title="Terminal"
npx @better-auth/cli@latest init
```
```

### ### Options

- \* `--name` - The name of your application. (Defaults to your `package.json`'s `name` property.)
- \* `--framework` - The framework your codebase is using. Currently, the only supported framework is `nextjs`.
- \* `--plugins` - The plugins you want to use. You can specify multiple plugins by separating them with a comma.
- \* `--database` - The database you want to use. Currently, the only supported database is `sqlite`.
- \* `--package-manager` - The package manager you want to use. Currently, the only supported package managers are `npm`, `pnpm`, `yarn`, `bun`. (Defaults to the manager you used to initialize the CLI.)

### ## Secret

The CLI also provides a way to generate a secret key for your Better Auth instance.

```
```bash title="Terminal"
npx @better-auth/cli@latest secret
```
```

### ## Common Issues

**\*\*Error: Cannot find module X\*\***

If you see this error, it means the CLI can't resolve imported modules in your Better Auth config file. We're working on a fix for many of these issues, but in the meantime, you can try the following:

- \* Remove any import aliases in your config file and use relative paths instead. After running the CLI, you can revert to using aliases.

```
file: ./content/docs/concepts/client.mdx
meta: {
  "title": "Client",
  "description": "Better Auth client library for authentication."
}
```

Better Auth offers a client library compatible with popular frontend frameworks like React, Vue, Svelte, and more. This client library includes a set of functions for interacting with the Better Auth server. Each framework's client library is built on top of a core client library that is framework-agnostic, so that all methods and hooks are consistently available across all client libraries.

### ## Installation

If you haven't already, install better-auth.

```
<Tabs groupId="package-manager" persist items={}>
  <Tab value="npm">
    ```bash
    npm i better-auth
    ```
  </Tab>

  <Tab value="pnpm">
    ```bash
    pnpm add better-auth
    ```
  </Tab>

  <Tab value="yarn">
    ```bash
    yarn add better-auth
    ```
  </Tab>
```

```
</Tab>
```

```
<Tab value="bun">
  ```bash
  bun add better-auth
  ```
```

```
</Tab>
```

```
</Tabs>
```

### ## Create Client Instance

Import `createAuthClient` from the package for your framework (e.g., "better-auth/react" for React). Call the function to create your client. Pass the base URL of your auth server. If the auth server is running on the same domain as your client, you can skip this step.

```
<Callout type="info">
```

If you're using a different base path other than `/api/auth`, make sure to pass the whole URL, including the path. (e.g., `http://localhost:3000/custom-path/auth`)

```
</Callout>
```

```
<Tabs
  items={["react", "vue", "svelte", "solid",
"vanilla"]}
  defaultValue="react"
>
```

```
<Tab value="vanilla">
  ```ts title="lib/auth-client.ts"
  import { createAuthClient } from "better-auth/client"
  export const authClient = createAuthClient({
    baseUrl: "http://localhost:3000" // The base URL of your auth server // [!code highlight]
  })
  ```
```

```
</Tab>
```

```
<Tab value="react" title="lib/auth-client.ts">
  ```ts title="lib/auth-client.ts"
  import { createAuthClient } from "better-auth/react"
  export const authClient = createAuthClient({
    baseUrl: "http://localhost:3000" // The base URL of your auth server // [!code highlight]
  })
  ```
```

```
</Tab>
```

```
<Tab value="vue" title="lib/auth-client.ts">
  ```ts title="lib/auth-client.ts"
  import { createAuthClient } from "better-auth/vue"
  export const authClient = createAuthClient({
    baseUrl: "http://localhost:3000" // The base URL of your auth server // [!code highlight]
  })
  ```
```

```
</Tab>
```

```
<Tab value="svelte" title="lib/auth-client.ts">
  ```ts title="lib/auth-client.ts"
  import { createAuthClient } from "better-auth/svelte"
  export const authClient = createAuthClient({
    baseUrl: "http://localhost:3000" // The base URL of your auth server // [!code highlight]
  })
  ```
```

```
</Tab>
```

```
<Tab value="solid" title="lib/auth-client.ts">
  ```ts title="lib/auth-client.ts"
  import { createAuthClient } from "better-auth/solid"
  export const authClient = createAuthClient({
```

```

    baseUrl: "http://localhost:3000" // The base URL of your auth server // [!code highlight]
  })
  ...
</Tab>
</Tabs>

```

### ## Usage

Once you've created your client instance, you can use the client to interact with the Better Auth server. The client provides a set of functions by default and they can be extended with plugins.

#### \*\*Example: Sign In\*\*

```

```ts title="auth-client.ts"
import { createAuthClient } from "better-auth/client"
const authClient = createAuthClient()

await authClient.signIn.email({
  email: "test@user.com",
  password: "password1234"
})
```

```

### #### Hooks

On top of normal methods, the client provides hooks to easily access different reactive data. Every hook is available in the root object of the client and they all start with `use`.

#### \*\*Example: useSession\*\*

```

<Tabs items={["React", "Vue", "Svelte", "Solid"]} defaultValue="React">
  <Tab value="React">
    ```tsx title="user.tsx"
    //make sure you're using the react client
    import { createAuthClient } from "better-auth/react"
    const { useSession } = createAuthClient() // [!code highlight]

    export function User() {
      const {
        data: session,
        isPending, //loading state
        error, //error object
        refetch //refetch the session
      } = useSession()
      return (
        //...
      )
    }
    ...
  </Tab>

  <Tab value="Vue">
    ```vue title="user.vue"
    <script lang="ts" setup>
    import { authClient } from '@lib/auth-client'
    const session = authClient.useSession()
    </script>
    <template>
      <div>
        <button v-if="!session.data" @click="()" => authClient.signIn.social({
          provider: 'github'
        })>
          Continue with GitHub
        </button>
      </div>
    </template>
  </Tab>

```

```

    <pre>{{ session.data }}</pre>
    <button v-if="session.data" @click="authClient.signOut()">
      Sign out
    </button>
  </div>
</div>
</template>
...
</Tab>

<Tab value="Svelte">
  ```svelte title="user.svelte"
  <script lang="ts">
    import { client } from "$lib/client";
    const session = client.useSession();
  </script>

  <div
    style="display: flex; flex-direction: column; gap: 10px; border-radius: 10px; border: 1px solid #4B453F; padding: 20px;
margin-top: 10px;"
  >
    <div>
      {#if $session}
        <div>
          <p>
            {$session?.data?.user.name}
          </p>
          <p>
            {$session?.data?.user.email}
          </p>
          <button
            on:click={async () => {
              await authClient.signOut();
            }}
          >
            Signout
          </button>
        </div>
      {:else}
        <button
          on:click={async () => {
            await authClient.signIn.social({
              provider: "github",
            });
          }}
        >
          Continue with GitHub
        </button>
      {/if}
    </div>
    ...
  </Tab>

  <Tab value="Solid">
    ```tsx title="user.tsx"
    import { client } from "~/lib/client";
    import { Show } from 'solid-js';

    export default function Home() {
      const session = client.useSession()
      return (
        <Show
          when={session()}
          fallback={<button onClick={toggle}>Log in</button>}

```

```

    >
    <button onClick={toggle}>Log out</button>
  </Show>
);
}
...
</Tab>
</Tabs>

```

### Fetch Options

The client uses a library called [better fetch](https://better-fetch.vercel.app) to make requests to the server.

Better fetch is a wrapper around the native fetch API that provides a more convenient way to make requests. It's created by the same team behind Better Auth and is designed to work seamlessly with it.

You can pass any default fetch options to the client by passing `fetchOptions` object to the `createAuthClient`.

```

```ts title="auth-client.ts"
import { createAuthClient } from "better-auth/client"

const authClient = createAuthClient({
  fetchOptions: {
    //any better-fetch options
  },
})
```

```

You can also pass fetch options to most of the client functions. Either as the second argument or as a property in the object.

```

```ts title="auth-client.ts"
await authClient.signIn.email({
  email: "email@email.com",
  password: "password1234",
}, {
  onSuccess(ctx) {
    //
  }
})

//or

await authClient.signIn.email({
  email: "email@email.com",
  password: "password1234",
  fetchOptions: {
    onSuccess(ctx) {
      //
    }
  },
})
```

```

### Handling Errors

Most of the client functions return a response object with the following properties:

- \* `data`: The response data.
- \* `error`: The error object if there was an error.

the error object contains the following properties:

- \* `message`: The error message. (e.g., "Invalid email or password")
- \* `status`: The HTTP status code.
- \* `statusText`: The HTTP status text.



```

` `` ts title="auth-client.ts"
const { data, error } = await authClient.signIn.email({
  email: "email@email.com",
  password: "password1234"
})
if (error) {
  //handle error
}
` ``

```

If the actions accepts a `fetchOptions` option, you can pass `onError` callback to handle errors.

```

` `` ts title="auth-client.ts"

await authClient.signIn.email({
  email: "email@email.com",
  password: "password1234",
}, {
  onError(ctx) {
    //handle error
  }
})

//or
await authClient.signIn.email({
  email: "email@email.com",
  password: "password1234",
  fetchOptions: {
    onError(ctx) {
      //handle error
    }
  }
})
` ``

```

Hooks like `useSession` also return an error object if there was an error fetching the session. On top of that, they also return a `isPending` property to indicate if the request is still pending.

```

` `` ts title="auth-client.ts"
const { data, error, isPending } = useSession()
if (error) {
  //handle error
}
` ``

```

#### #### Error Codes

The client instance contains \$ERROR\\_CODES object that contains all the error codes returned by the server. You can use this to handle error translations or custom error messages.

```

` `` ts title="auth-client.ts"
const authClient = createAuthClient();

type ErrorTypes = Partial<
  Record<
    keyof typeof authClient.$ERROR_CODES,
    {
      en: string;
      es: string;
    }
  >
>;

```

```

const errorCodes = {

```

```

    USER_ALREADY_EXISTS: {
      en: "user already registered",
      es: "usuario ya registrada",
    },
  } satisfies ErrorTypes;

const getErrorMessage = (code: string, lang: "en" | "es") => {
  if (code in errorCodes) {
    return errorCodes[code as keyof typeof errorCodes][lang];
  }
  return "";
};

```

```

const { error } = await authClient.signUp.email({
  email: "user@email.com",
  password: "password",
  name: "User",
});
if(error?.code){
  alert(getErrorMessage(error.code, "en"));
}
` ``

```

### ### Plugins

You can extend the client with plugins to add more functionality. Plugins can add new functions to the client or modify existing ones.

#### \*\*Example: Magic Link Plugin\*\*

```

` `` ts title="auth-client.ts"
import { createAuthClient } from "better-auth/client"
import { magicLinkClient } from "better-auth/client/plugins"

const authClient = createAuthClient({
  plugins: [
    magicLinkClient()
  ]
})
` ``

```

once you've added the plugin, you can use the new functions provided by the plugin.

```

` `` ts title="auth-client.ts"
await authClient.signIn.magicLink({
  email: "test@email.com"
})
` ``

```

```

file: ./content/docs/concepts/cookies.mdx
meta: {
  "title": "Cookies",
  "description": "Learn how cookies are used in Better Auth."
}

```

Cookies are used to store data such as session tokens, OAuth state, and more. All cookies are signed using the `secret` key provided in the auth options.

### ### Cookie Prefix

Better Auth cookies will follow `\${prefix}.\${cookie\_name}` format by default. The prefix will be "better-auth" by default. You can change the prefix by setting `cookiePrefix` in the `advanced` object of the auth options.

```
```ts title="auth.ts"
import { betterAuth } from "better-auth"

export const auth = betterAuth({
  advanced: {
    cookiePrefix: "my-app"
  }
})
```
```

### ### Custom Cookies

All cookies are `httpOnly` and `secure` if the server is running in production mode.

If you want to set custom cookie names and attributes, you can do so by setting `cookieOptions` in the `advanced` object of the auth options.

By default, Better Auth uses the following cookies:

- \* `session\_token` to store the session token
- \* `session\_data` to store the session data if cookie cache is enabled
- \* `dont\_remember` to store the `dont\_remember` flag if remember me is disabled

Plugins may also use cookies to store data. For example, the Two Factor Authentication plugin uses the `two\_factor` cookie to store the two-factor authentication state.

```
```ts title="auth.ts"
import { betterAuth } from "better-auth"

export const auth = betterAuth({
  advanced: {
    cookies: {
      session_token: {
        name: "custom_session_token",
        attributes: {
          // Set custom cookie attributes
        }
      },
    },
  },
})
```
```

### ### Cross Subdomain Cookies

Sometimes you may need to share cookies across subdomains. For example, if you have `app.example.com` and `example.com`, and if you authenticate on `example.com`, you may want to access the same session on `app.example.com`.

By default, cookies are not shared between subdomains. However, if you need to access the same session across different subdomains, you can enable cross-subdomain cookies by configuring `crossSubDomainCookies` and `defaultCookieAttributes` in the `advanced` object of the auth options.

**<Callout>**The leading period of the `domain` attribute broadens the cookie's scope beyond your main domain, making it accessible across all subdomains.**</Callout>**

**<Callout type="note">**

If your frontend and backend are on different domains, you must configure CORS on your backend, set `credentials: 'include'` in your frontend requests, and add your frontend domain to `trustedOrigins`. See the example above for cookie attributes and trustedOrigins usage.

**</Callout>**

```
```ts title="auth.ts"
import { betterAuth } from "better-auth"

export const auth = betterAuth({
```

```

    advanced: {
      crossSubDomainCookies: {
        enabled: true,
        domain: ".example.com", // your domain
      },
    },
    trustedOrigins: [
      'https://example.com',
      'https://app1.example.com',
      'https://app2.example.com',
    ],
  })
  ``

```

### #### Secure Cookies

By default, cookies are secure only when the server is running in production mode. You can force cookies to be always secure by setting `useSecureCookies` to `true` in the `advanced` object in the auth options.

```

````ts title="auth.ts"
import { betterAuth } from "better-auth"

export const auth = betterAuth({
  advanced: {
    useSecureCookies: true
  }
})
````

```

```

file: ./content/docs/concepts/database.mdx
meta: {
  "title": "Database",
  "description": "Learn how to use a database with Better Auth."
}

```

### ## Adapters

Better Auth requires a database connection to store data. It comes with a query builder called [Kysely](https://kysely.dev/) to manage and query your database. The database will be used to store data such as users, sessions, and more. Plugins can also define their own database tables to store data.

You can pass a database connection to Better Auth by passing a supported database instance, a dialect instance or a Kysely instance in the database options.

You can learn more about supported Kysely dialects in the [Other relational databases](/docs/adapters/other-relational-databases) documentation. Or if you're using an ORM, you can find our supported ORM adapters in that same category on the documentation sidebar.

### ## CLI

Better Auth comes with a CLI tool to manage database migrations and generate schema.

### #### Running Migrations

The cli checks your database and prompts you to add missing tables or update existing ones with new columns. This is only supported for the built-in Kysely adapter. For other adapters, you can use the `generate` command to create the schema and handle the migration through your ORM.

```

````bash
npx @better-auth/cli migrate
````

```

### #### Generating Schema

Better Auth also provides a `generate` command to generate the schema required by Better Auth. The `generate` command creates the schema required by Better Auth. If you're using a database adapter like Prisma or Drizzle, this command will generate the right schema for your ORM. If you're using the built-in Kysely adapter, it will generate an SQL file you can run directly on your database.

```
```bash
npx @better-auth/cli generate
```
```

See the [CLI](/docs/concepts/cli) documentation for more information on the CLI.

<Callout>

If you prefer adding tables manually, you can do that as well. The core schema required by Better Auth is described below and you can find additional schema required by plugins in the plugin documentation.

</Callout>

## ## Secondary Storage

Secondary storage in Better Auth allows you to use key-value stores for managing session data, rate limiting counters, etc. This can be useful when you want to offload the storage of this intensive records to a high performance storage or even RAM.

### ### Implementation

To use secondary storage, implement the `SecondaryStorage` interface:

```
```typescript
interface SecondaryStorage {
  get: (key: string) => Promise<string | null>;
  set: (key: string, value: string, ttl?: number) => Promise<void>;
  delete: (key: string) => Promise<void>;
}
```
```

Then, provide your implementation to the `betterAuth` function:

```
```typescript
betterAuth({
  // ... other options
  secondaryStorage: {
    // Your implementation here
  },
});
```
```

### \*\*Example: Redis Implementation\*\*

Here's a basic example using Redis:

```
```typescript
import { createClient } from "redis";
import { betterAuth } from "better-auth";

const redis = createClient();
await redis.connect();

export const auth = betterAuth({
  // ... other options
  secondaryStorage: {
    get: async (key) => {
      const value = await redis.get(key);
      return value ? value : null;
    },
    set: async (key, value, ttl) => {

```

```

    if (ttl) await redis.set(key, value, { EX: ttl });
    // or for ioredis:
    // if (ttl) await redis.set(key, value, 'EX', ttl)
    else await redis.set(key, value);
  },
  delete: async (key) => {
    await redis.del(key);
  }
});
```

```

This implementation allows Better Auth to use Redis for storing session data and rate limiting counters. You can also add prefixes to the keys names.

### ## Core Schema

Better Auth requires the following tables to be present in the database. The types are in `typescript` format. You can use corresponding types in your database.

#### #### User

Table Name: `user`

```

<DatabaseTable
  fields=[
    {
      name: "id",
      type: "string",
      description: "Unique identifier for each user",
      isPrimaryKey: true,
    },
    {
      name: "name",
      type: "string",
      description: "User's chosen display name",
    },
    {
      name: "email",
      type: "string",
      description: "User's email address for communication and login",
    },
    {
      name: "emailVerified",
      type: "boolean",
      description: "Whether the user's email is verified",
    },
    {
      name: "image",
      type: "string",
      description: "User's image url",
      isOptional: true,
    },
    {
      name: "createdAt",
      type: "Date",
      description: "Timestamp of when the user account was created",
    },
    {
      name: "updatedAt",
      type: "Date",
      description: "Timestamp of the last update to the user's information",
    },
  ],
/>

```

### #### Session

Table Name: `session`

```
<DatabaseTable
  fields=[
    {
      name: "id",
      type: "string",
      description: "Unique identifier for each session",
      isPrimaryKey: true,
    },
    {
      name: "userId",
      type: "string",
      description: "The ID of the user",
      isForeignKey: true,
    },
    {
      name: "token",
      type: "string",
      description: "The unique session token",
      isUnique: true,
    },
    {
      name: "expiresAt",
      type: "Date",
      description: "The time when the session expires",
    },
    {
      name: "ipAddress",
      type: "string",
      description: "The IP address of the device",
      isOptional: true,
    },
    {
      name: "userAgent",
      type: "string",
      description: "The user agent information of the device",
      isOptional: true,
    },
    {
      name: "createdAt",
      type: "Date",
      description: "Timestamp of when the session was created",
    },
    {
      name: "updatedAt",
      type: "Date",
      description: "Timestamp of when the session was updated",
    },
  ]
/>
```

### #### Account

Table Name: `account`

```
<DatabaseTable
  fields=[
    {
      name: "id",
      type: "string",
      description: "Unique identifier for each account",
```

```
    isPrimaryKey: true,
  },
  {
    name: "userId",
    type: "string",
    description: "The ID of the user",
    isForeignKey: true,
  },
  {
    name: "accountId",
    type: "string",
    description:
      "The ID of the account as provided by the SSO or equal to userId for credential accounts",
  },
  {
    name: "providerId",
    type: "string",
    description: "The ID of the provider",
  },
  {
    name: "accessToken",
    type: "string",
    description: "The access token of the account. Returned by the provider",
    isOptional: true,
  },
  {
    name: "refreshToken",
    type: "string",
    description: "The refresh token of the account. Returned by the provider",
    isOptional: true,
  },
  {
    name: "accessTokenExpiresAt",
    type: "Date",
    description: "The time when the access token expires",
    isOptional: true,
  },
  {
    name: "refreshTokenExpiresAt",
    type: "Date",
    description: "The time when the refresh token expires",
    isOptional: true,
  },
  {
    name: "scope",
    type: "string",
    description: "The scope of the account. Returned by the provider",
    isOptional: true,
  },
  {
    name: "idToken",
    type: "string",
    description: "The ID token returned from the provider",
    isOptional: true,
  },
  {
    name: "password",
    type: "string",
    description:
      "The password of the account. Mainly used for email and password authentication",
    isOptional: true,
  },
  {
    name: "createdAt",
    type: "Date",
```



```

    description: "Timestamp of when the account was created",
  },
  {
    name: "updatedAt",
    type: "Date",
    description: "Timestamp of when the account was updated",
  },
]
/>

```

### ### Verification

Table Name: `verification`

```

<DatabaseTable
  fields=[
    {
      name: "id",
      type: "string",
      description: "Unique identifier for each verification",
      isPrimaryKey: true,
    },
    {
      name: "identifier",
      type: "string",
      description: "The identifier for the verification request",
    },
    {
      name: "value",
      type: "string",
      description: "The value to be verified",
    },
    {
      name: "expiresAt",
      type: "Date",
      description: "The time when the verification request expires",
    },
    {
      name: "createdAt",
      type: "Date",
      description: "Timestamp of when the verification request was created",
    },
    {
      name: "updatedAt",
      type: "Date",
      description: "Timestamp of when the verification request was updated",
    },
  ]
/>

```

### ## Custom Tables

Better Auth allows you to customize the table names and column names for the core schema. You can also extend the core schema by adding additional fields to the user and session tables.

#### ### Custom Table Names

You can customize the table names and column names for the core schema by using the `modelName` and `fields` properties in your auth config:

```

```ts title="auth.ts"
export const auth = betterAuth({
  user: {
    modelName: "users",
    fields: {

```

```

      name: "full_name",
      email: "email_address",
    },
  },
  session: {
    modelName: "user_sessions",
    fields: {
      userId: "user_id",
    },
  },
});
```

```

<Callout>

Type inference in your code will still use the original field names (e.g., `user.name`, not `user.full\_name`).

</Callout>

To customize table names and column name for plugins, you can use the `schema` property in the plugin config:

```

```ts title="auth.ts"
import { betterAuth } from "better-auth";
import { twoFactor } from "better-auth/plugins";

export const auth = betterAuth({
  plugins: [
    twoFactor({
      schema: {
        user: {
          fields: {
            twoFactorEnabled: "two_factor_enabled",
            secret: "two_factor_secret",
          },
        },
      },
    }),
  ],
});
```

```

### ### Extending Core Schema

Better Auth provides a type-safe way to extend the `user` and `session` schemas. You can add custom fields to your auth config, and the CLI will automatically update the database schema. These additional fields will be properly inferred in functions like `useSession`, `signup.email`, and other endpoints that work with user or session objects.

To add custom fields, use the `additionalFields` property in the `user` or `session` object of your auth config. The `additionalFields` object uses field names as keys, with each value being a `FieldAttributes` object containing:

- \* `type`: The data type of the field (e.g., "string", "number", "boolean").
- \* `required`: A boolean indicating if the field is mandatory.
- \* `defaultValue`: The default value for the field (note: this only applies in the JavaScript layer; in the database, the field will be optional).
- \* `input`: This determines whether a value can be provided when creating a new record (default: `true`). If there are additional fields, like `role`, that should not be provided by the user during signup, you can set this to `false`.

Here's an example of how to extend the user schema with additional fields:

```

```ts title="auth.ts"
import { betterAuth } from "better-auth";

export const auth = betterAuth({
  user: {
    additionalFields: {
      role: {

```

```

    type: "string",
    required: false,
    defaultValue: "user",
    input: false, // don't allow user to set role
  },
  lang: {
    type: "string",
    required: false,
    defaultValue: "en",
  },
},
});
```,

```

Now you can access the additional fields in your application logic.

```

```ts
//on signup
const res = await auth.api.signUpEmail({
  email: "test@example.com",
  password: "password",
  name: "John Doe",
  lang: "fr",
});

```

```

//user object
res.user.role; // > "admin"
res.user.lang; // > "fr"
```

```

<Callout>  
 See the  
[\[TypeScript\]\(/docs/concepts/typescript#inferring-additional-fields-on-client\)](#)  
 documentation for more information on how to infer additional fields on the  
 client side.  
 </Callout>

If you're using social / OAuth providers, you may want to provide `mapProfileToUser` to map the profile data to the user object. So, you can populate additional fields from the provider's profile.

**\*\*Example: Mapping Profile to User For `firstName` and `lastName`\*\***

```

```ts title="auth.ts"
import { betterAuth } from "better-auth";

export const auth = betterAuth({
  socialProviders: {
    github: {
      clientId: "YOUR_GITHUB_CLIENT_ID",
      clientSecret: "YOUR_GITHUB_CLIENT_SECRET",
      mapProfileToUser: (profile) => {
        return {
          firstName: profile.name.split(" ")[0],
          lastName: profile.name.split(" ")[1],
        };
      },
    },
  },
  google: {
    clientId: "YOUR_GOOGLE_CLIENT_ID",
    clientSecret: "YOUR_GOOGLE_CLIENT_SECRET",
    mapProfileToUser: (profile) => {
      return {
        firstName: profile.given_name,
        lastName: profile.family_name,
      };
    },
  },
});

```

```

    };
  },
},
});
```

```

### ### ID Generation

Better Auth by default will generate unique IDs for users, sessions, and other entities. If you want to customize how IDs are generated, you can configure this in the `advanced.database.generateId` option in your auth config.

You can also disable ID generation by setting the `advanced.database.generateId` option to `false`. This will assume your database will generate the ID automatically.

#### **\*\*Example: Automatic Database IDs\*\***

```

```ts title="auth.ts"
import { betterAuth } from "better-auth";
import { db } from "./db";

export const auth = betterAuth({
  database: {
    db: db,
  },
  advanced: {
    database: {
      generateId: false,
    },
  },
});
```

```

### ### Database Hooks

Database hooks allow you to define custom logic that can be executed during the lifecycle of core database operations in Better Auth. You can create hooks for the following models: **\*\*user\*\***, **\*\*session\*\***, and **\*\*account\*\***.

There are two types of hooks you can define:

#### #### 1. Before Hook

**\*\*Purpose\*\***: This hook is called before the respective entity (user, session, or account) is created or updated.

**\*\*Behavior\*\***: If the hook returns `false`, the operation will be aborted. And If it returns a data object, it'll replace the original payload.

#### #### 2. After Hook

**\*\*Purpose\*\***: This hook is called after the respective entity is created or updated.

**\*\*Behavior\*\***: You can perform additional actions or modifications after the entity has been successfully created or updated.

#### **\*\*Example Usage\*\***

```

```typescript title="auth.ts"
import { betterAuth } from "better-auth";

export const auth = betterAuth({
  databaseHooks: {
    user: {
      create: {
        before: async (user, ctx) => {
          // Modify the user object before it is created
          return {
            data: {

```

```

    ...user,
    firstName: user.name.split(" ")[0],
    lastName: user.name.split(" ")[1],
  },
};
},
after: async (user) => {
  //perform additional actions, like creating a stripe customer
},
},
},
});
```

```

#### #### Throwing Errors

If you want to stop the database hook from proceeding, you can throw errors using the `APIError` class imported from `better-auth/api`.

```

```typescript title="auth.ts"
import { betterAuth } from "better-auth";
import { APIError } from "better-auth/api";

export const auth = betterAuth({
  databaseHooks: {
    user: {
      create: {
        before: async (user, ctx) => {
          if (user.isAgreedToTerms === false) {
            // Your special condition.
            // Send the API error.
            throw new APIError("BAD_REQUEST", {
              message: "User must agree to the TOS before signing up.",
            });
          }
          return {
            data: user,
          };
        },
      },
    },
  },
});
```

```

Much like standard hooks, database hooks also provide a `ctx` object that offers a variety of useful properties. Learn more in the [Hooks Documentation](/docs/concepts/hooks#ctx).

#### ## Plugins Schema

Plugins can define their own tables in the database to store additional data. They can also add columns to the core tables to store additional data. For example, the two factor authentication plugin adds the following columns to the `user` table:

- \* `twoFactorEnabled`: Whether two factor authentication is enabled for the user.
- \* `twoFactorSecret`: The secret key used to generate TOTP codes.
- \* `twoFactorBackupCodes`: Encrypted backup codes for account recovery.

To add new tables and columns to your database, you have two options:

- `CLI`: Use the migrate or generate command. These commands will scan your database and guide you through adding any missing tables or columns.
- `Manual Method`: Follow the instructions in the plugin documentation to manually add tables and columns.

Both methods ensure your database schema stays up to date with your plugins' requirements.

```
file: ./content/docs/concepts/email.mdx
meta: {
  "title": "Email",
  "description": "Learn how to use email with Better Auth."
}
```

Email is a key part of Better Auth, required for all users regardless of their authentication method. Better Auth provides email and password authentication out of the box, and a lot of utilities to help you manage email verification, password reset, and more.

## ## Email Verification

Email verification is a security feature that ensures users provide a valid email address. It helps prevent spam and abuse by confirming that the email address belongs to the user. In this guide, you'll get a walk through of how to implement token based email verification in your app.

To use otp based email verification, check out the [\[OTP Verification\]\(/docs/plugins/email-otp\)](/docs/plugins/email-otp) guide.

## ### Adding Email Verification to Your App

To enable email verification, you need to pass a function that sends a verification email with a link.

**\*\*\*sendVerificationEmail\*\*:** This function is triggered when email verification starts. It accepts a data object with the following properties:

- \* `user`: The user object containing the email address.
- \* `url`: The verification URL the user must click to verify their email.
- \* `token`: The verification token used to complete the email verification to be used when implementing a custom verification URL.

and a `request` object as the second parameter.

```
```ts title="auth.ts"
import { betterAuth } from 'better-auth';
import { sendEmail } from './email'; // your email sending function

export const auth = betterAuth({
  emailVerification: {
    sendVerificationEmail: async ({ user, url, token }, request) => {
      await sendEmail({
        to: user.email,
        subject: 'Verify your email address',
        text: `Click the link to verify your email: ${url}`
      })
    }
  }
})
```
```

## ### Triggering Email Verification

You can initiate email verification in several ways:

### #### 1. During Sign-up

To automatically send a verification email at signup, set `emailVerification.sendOnSignUp` to `true`.

```
```ts title="auth.ts"
import { betterAuth } from 'better-auth';

export const auth = betterAuth({
  emailVerification: {
    sendOnSignUp: true
  }
})
```
```

```
```
```

This sends a verification email when a user signs up. For social logins, email verification status is read from the SSO.

<Callout>

With `sendOnSignUp` enabled, when the user logs in with an SSO that does not claim the email as verified, Better Auth will dispatch a verification email, but the verification is not required to login even when `requireEmailVerification` is enabled.

</Callout>

## #### 2. Require Email Verification

If you enable require email verification, users must verify their email before they can log in. And every time a user tries to sign in, `sendVerificationEmail` is called.

<Callout>

This only works if you have `sendVerificationEmail` implemented and if the user is trying to sign in with email and password.

</Callout>

```
```ts title="auth.ts"
export const auth = betterAuth({
  emailAndPassword: {
    requireEmailVerification: true
  }
})
```
```

if a user tries to sign in without verifying their email, you can handle the error and show a message to the user.

```
```ts title="auth-client.ts"
await authClient.signIn.email({
  email: "email@example.com",
  password: "password"
},{
  onError: (ctx) => {
    // Handle the error
    if(ctx.error.status === 403) {
      alert("Please verify your email address")
    }
    //you can also show the original error message
    alert(ctx.error.message)
  }
})
```
```

## #### 3. Manually

You can also manually trigger email verification by calling `sendVerificationEmail`.

```
```ts
await authClient.sendVerificationEmail({
  email: "user@email.com",
  callbackURL: "/" // The redirect URL after verification
})
```
```

## ### Verifying the Email

If the user clicks the provided verification URL, their email is automatically verified, and they are redirected to the `callbackURL`.

For manual verification, you can send the user a custom link with the `token` and call the `verifyEmail` function.

```
```ts
```

```
await authClient.verifyEmail({
  query: {
    token: "" // Pass the token here
  }
})
```,
```

### ### Auto SignIn After Verification

To sign in the user automatically after they successfully verify their email, set the `autoSignInAfterVerification` option to `true`:

```
```ts
const auth = betterAuth({
  //...your other options
  emailVerification: {
    autoSignInAfterVerification: true
  }
})
```,
```

### ## Password Reset Email

Password reset allows users to reset their password if they forget it. Better Auth provides a simple way to implement password reset functionality.

You can enable password reset by passing a function that sends a password reset email with a link.

```
```ts title="auth.ts"
import { betterAuth } from 'better-auth';
import { sendEmail } from './email'; // your email sending function

export const auth = betterAuth({
  emailAndPassword: {
    enabled: true,
    sendResetPassword: async ({ user, url, token }, request) => {
      await sendEmail({
        to: user.email,
        subject: 'Reset your password',
        text: `Click the link to reset your password: ${url}`
      })
    }
  }
})
```,
```

Check out the [Email and Password](/docs/authentication/email-password#forget-password) guide for more details on how to implement password reset in your app.

Also you can check out the [Otp verification](/docs/plugins/email-otp#reset-password) guide for how to implement password reset with OTP in your app.

```
file: ./content/docs/concepts/hooks.mdx
meta: {
  "title": "Hooks",
  "description": "Better Auth Hooks let you customize BetterAuth's behavior"
}
```

Hooks in Better Auth let you "hook into" the lifecycle and execute custom logic. They provide a way to customize Better Auth's behavior without writing a full plugin.

<Callout>

We highly recommend using hooks if you need to make custom adjustments to an endpoint rather than making another endpoint outside of Better Auth.

</Callout>



## ## Before Hooks

**\*\*Before hooks\*\*** run *\*before\** an endpoint is executed. Use them to modify requests, pre validate data, or return early.

### #### Example: Enforce Email Domain Restriction

This hook ensures that users can only sign up if their email ends with `@example.com`:

```

` `` ts title="auth.ts"
import { betterAuth } from "better-auth";
import { createAuthMiddleware, APIError } from "better-auth/api";

export const auth = betterAuth({
  hooks: {
    before: createAuthMiddleware(async (ctx) => {
      if (ctx.path !== "/sign-up/email") {
        return;
      }
      if (!ctx.body?.email.endsWith("@example.com")) {
        throw new APIError("BAD_REQUEST", {
          message: "Email must end with @example.com",
        });
      }
    },
  ),
},
});
` ``

```

### #### Example: Modify Request Context

To adjust the request context before proceeding:

```

` `` ts title="auth.ts"
import { betterAuth } from "better-auth";
import { createAuthMiddleware } from "better-auth/api";

export const auth = betterAuth({
  hooks: {
    before: createAuthMiddleware(async (ctx) => {
      if (ctx.path === "/sign-up/email") {
        return {
          context: {
            ...ctx,
            body: {
              ...ctx.body,
              name: "John Doe",
            },
          },
        };
      }
    },
  ),
},
});
` ``

```

## ## After Hooks

**\*\*After hooks\*\*** run *\*after\** an endpoint is executed. Use them to modify responses.

### #### Example: Send a notification to your channel when a new user is registered

```

` `` ts title="auth.ts"
import { betterAuth } from "better-auth";
import { createAuthMiddleware } from "better-auth/api";

```

```
import { sendMessage } from "@lib/notification"

export const auth = betterAuth({
  hooks: {
    after: createAuthMiddleware(async (ctx) => {
      if (ctx.path.startsWith("/sign-up")) {
        const newSession = ctx.context.newSession;
        if (newSession) {
          sendMessage({
            type: "user-register",
            name: newSession.user.name,
          })
        }
      }
    }),
  },
});
``
```

### ### Ctx

When you call `createAuthMiddleware` a `ctx` object is passed that provides a lot of useful properties. Including:

- \* **Path:** `ctx.path` to get the current endpoint path.
- \* **Body:** `ctx.body` for parsed request body (available for POST requests).
- \* **Headers:** `ctx.headers` to access request headers.
- \* **Request:** `ctx.request` to access the request object (may not exist in server-only endpoints).
- \* **Query Parameters:** `ctx.query` to access query parameters.
- \* **Context:** `ctx.context` auth related context, useful for accessing new session, auth cookies configuration, password hashing, config...

and more.

### #### Request Response

This utilities allows you to get request information and to send response from a hook.

### #### JSON Responses

Use `ctx.json` to send JSON responses:

```
``ts
const hook = createAuthMiddleware(async (ctx) => {
  return ctx.json({
    message: "Hello World",
  });
});
``
```

### #### Redirects

Use `ctx.redirect` to redirect users:

```
``ts
import { createAuthMiddleware } from "better-auth/api";

const hook = createAuthMiddleware(async (ctx) => {
  throw ctx.redirect("/sign-up/name");
});
``
```

### #### Cookies

- \* Set cookies: `ctx.setCookies` or `ctx.setSignedCookie`.
- \* Get cookies: `ctx.getCookies` or `ctx.getSignedCookies`.

Example:

```
```ts
import { createAuthMiddleware } from "better-auth/api";

const hook = createAuthMiddleware(async (ctx) => {
  ctx.setCookies("my-cookie", "value");
  await ctx.setSignedCookie("my-signed-cookie", "value", ctx.context.secret, {
    maxAge: 1000,
  });

  const cookie = ctx.getCookies("my-cookie");
  const signedCookie = await ctx.getSignedCookies("my-signed-cookie");
});
```
```

#### ##### Errors

Throw errors with `APIError` for a specific status code and message:

```
```ts
import { createAuthMiddleware, APIError } from "better-auth/api";

const hook = createAuthMiddleware(async (ctx) => {
  throw new APIError("BAD_REQUEST", {
    message: "Invalid request",
  });
});
```
```

#### ### Context

The `ctx` object contains another `context` object inside that's meant to hold contexts related to auth. Including a newly created session on after hook, cookies configuration, password hasher and so on.

#### ##### New Session

The newly created session after an endpoint is run. This only exist in after hook.

```
```ts title="auth.ts"
createAuthMiddleware(async (ctx) => {
  const newSession = ctx.context.newSession
});
```
```

#### ##### Returned

The returned value from the hook is passed to the next hook in the chain.

```
```ts title="auth.ts"
createAuthMiddleware(async (ctx) => {
  const returned = ctx.context.returned; //this could be a successful response or an APIError
});
```
```

#### ##### Response Headers

The response headers added by endpoints and hooks that run before this hook.

```
```ts title="auth.ts"
createAuthMiddleware(async (ctx) => {
  const responseHeaders = ctx.context.responseHeaders;
});
```
```

#### #### Predefined Auth Cookies

Access BetterAuth's predefined cookie properties:

```
```ts title="auth.ts"
createAuthMiddleware(async (ctx) => {
  const cookieName = ctx.context.authCookies.sessionToken.name;
});
```
```

#### #### Secret

You can access the `secret` for your auth instance on `ctx.context.secret`

#### #### Password

The password object provider `hash` and `verify`

- \* `ctx.context.password.hash`: let's you hash a given password.
- \* `ctx.context.password.verify`: let's you verify given `password` and a `hash`.

#### #### Adapter

Adapter exposes the adapter methods used by Better Auth. Including `findOne`, `findMany`, `create`, `delete`, `update` and `updateMany`. You generally should use your actually `db` instance from your orm rather than this adapter.

#### #### Internal Adapter

These are calls to your db that perform specific actions. `createUser`, `createSession`, `updateSession` ...

This may be useful to use instead of using your db directly to get access to `databaseHooks`, proper `secondaryStorage` support and so on. If you're make a query similar to what exist in this internal adapter actions it's worth a look.

#### #### generateId

You can use `ctx.context.generateId` to generate Id for various reasons.

#### ## Reusable Hooks

If you need to reuse a hook across multiple endpoints, consider creating a plugin. Learn more in the [Plugins Documentation] (/docs/concepts/plugins).

```
file: ./content/docs/concepts/oauth.mdx
meta: {
  "title": "OAuth",
  "description": "How Better Auth handles OAuth"
}
```

Better Auth comes with built-in support for OAuth 2.0 and OpenID Connect. This allows you to authenticate users via popular OAuth providers like Google, Facebook, GitHub, and more.

If your desired provider isn't directly supported, you can use the [Generic OAuth Plugin] (/docs/plugins/generic-oauth) for custom integrations.

#### ## Configuring Social Providers

To enable a social provider, you need to provide `clientId` and `clientSecret` for the provider.

Here's an example of how to configure Google as a provider:

```
```ts title="auth.ts"
import { betterAuth } from "better-auth";
```

```
export const auth = betterAuth({
  // Other configurations...
  socialProviders: {
    google: {
      clientId: "YOUR_GOOGLE_CLIENT_ID",
      clientSecret: "YOUR_GOOGLE_CLIENT_SECRET",
    },
  },
});
````
```

### ## Usage

#### #### Sign In

To sign in with a social provider, you can use the `signIn.social` function with the `authClient` or `auth.api` for server-side usage.

```
````ts
await authClient.signIn.social({
  provider: "google", // or any other provider id
})
````
```

server-side usage:

```
````ts
await auth.api.signInSocial({
  body: {
    provider: "google", // or any other provider id
  },
});
````
```

#### #### Link account

To link an account to a social provider, you can use the `linkAccount` function with the `authClient` or `auth.api` for server-side usage.

```
````ts
await authClient.linkSocial({
  provider: "google", // or any other provider id
})
````
```

server-side usage:

```
````ts
await auth.api.linkSocialAccount({
  body: {
    provider: "google", // or any other provider id
  },
  headers: // pass headers with authenticated token
});
````
```

#### #### Get Access Token

To get the access token for a social provider, you can use the `getAccessToken` function with the `authClient` or `auth.api` for server-side usage. When you use this endpoint, if the access token is expired, it will be refreshed.

```
````ts
const { accessToken } = await authClient.getAccessToken({
  providerId: "google", // or any other provider id
  accountId: "accountId", // optional, if you want to get the access token for a specific account
});
````
```

```
})
```

```

server-side usage:

```
```ts
await auth.api.getAccessToken({
  body: {
    providerId: "google", // or any other provider id
    accountId: "accountId", // optional, if you want to get the access token for a specific account
    userId: "userId", // optional, if you don't provide headers with authenticated token
  },
  headers: // pass headers with authenticated token
});
```

```

### ### Get Account Info Provided by the provider

To get provider specific account info you can use the `accountInfo` function with the `authClient` or `auth.api` for server-side usage.

```
```ts
const info = await authClient.accountInfo({
  accountId: "accountId", // here you pass in the provider given account id, the provider is automatically detected from the account id
})
```

```

server-side usage:

```
```ts
await auth.api.accountInfo({
  body: { accountId: "accountId" },
  headers: // pass headers with authenticated token
});
```

```

### ### Requesting Additional Scopes

Sometimes your application may need additional OAuth scopes after the user has already signed up (e.g., for accessing GitHub repositories or Google Drive). Users may not want to grant extensive permissions initially, preferring to start with minimal permissions and grant additional access as needed.

You can request additional scopes by using the `linkSocial` method with the same provider. This will trigger a new OAuth flow that requests the additional scopes while maintaining the existing account connection.

```
```ts
const requestAdditionalScopes = async () => {
  await authClient.linkSocial({
    provider: "google",
    scopes: ["https://www.googleapis.com/auth/drive.file"],
  });
};
```

```

<Callout>

Make sure you're running Better Auth version 1.2.7 or later. Earlier versions (like 1.2.2) may show a "Social account already linked" error when trying to link with an existing provider for additional scopes.

</Callout>

### ### Other Provider Configurations

**\*\*scope\*\*** The scope of the access request. For example, `email` or `profile`.

**\*\*redirectURI\*\*** Custom redirect URI for the provider. By default, it uses `/api/auth/callback/\${providerName}`.

**\*\*disableImplicitSignUp:\*\*** Disables implicit sign-up. In order to sign up a user, `requestSignUp` needs to be set to `true` when signing in.

**\*\*disableSignUp:\*\*** Disables sign-up for new users.

**\*\*disableIdTokenSignIn:\*\*** Disables the use of the ID token for sign-in. By default, it's enabled for some providers like Google and Apple.

**\*\*verifyIdToken\*\*** A custom function to verify the ID token.

**\*\*getUserInfo\*\*** A custom function to fetch user information from the provider. Given the tokens returned from the provider, this function should return the user's information.

**\*\*overrideUserInfoOnSignIn\*\*:** A boolean value that determines whether to override the user information in the database when signing in. By default, it is set to `false`, meaning that the user information will not be overridden during sign-in. If you want to update the user information every time they sign in, set this to `true`.

**\*\*refreshAccessToken\*\*:** A custom function to refresh the token. This feature is only supported for built-in social providers (Google, Facebook, GitHub, etc.) and is not currently supported for custom OAuth providers configured through the Generic OAuth Plugin. For built-in providers, you can provide a custom function to refresh the token if needed.

**\*\*mapProfileToUser\*\*** A custom function to map the user profile returned from the provider to the user object in your database.

Useful, if you have additional fields in your user object you want to populate from the provider's profile. Or if you want to change how by default the user object is mapped.

```
```ts title="auth.ts"
import { betterAuth } from "better-auth";

export const auth = betterAuth({
  // Other configurations...
  socialProviders: {
    google: {
      clientId: "YOUR_GOOGLE_CLIENT_ID",
      clientSecret: "YOUR_GOOGLE_CLIENT_SECRET",
      mapProfileToUser: (profile) => {
        return {
          firstName: profile.given_name,
          lastName: profile.family_name,
        };
      },
    },
  },
});
```
```

## ## How OAuth Works in Better Auth

Here's what happens when a user selects a provider to authenticate with:

- \*\*Configuration Check:\*\*** Ensure the necessary provider details (e.g., client ID, secret) are configured.
- \*\*State Generation:\*\*** Generate and save a state token in your database for CSRF protection.
- \*\*PKCE Support:\*\*** If applicable, create a PKCE code challenge and verifier for secure exchanges.
- \*\*Authorization URL Construction:\*\*** Build the provider's authorization URL with parameters like client ID, redirect URI, state, etc. The callback URL usually follows the pattern `/api/auth/callback/\${providerName}`.
- \*\*User Redirection:\*\***
  - \* If redirection is enabled, users are redirected to the provider's login page.
  - \* If redirection is disabled, the authorization URL is returned for the client to handle the redirection.

## ### Post-Login Flow

After the user completes the login process, the provider redirects them back to the callback URL with a code and state. Better Auth handles the rest:

1. **Token Exchange:** The code is exchanged for an access token and user information.
2. **User Handling:**
  - \* If the user doesn't exist, a new account is created.
  - \* If the user exists, they are logged in.
  - \* If the user has multiple accounts across providers, Better Auth links them based on your configuration. Learn more about [account linking](#) (`/docs/concepts/users-accounts#account-linking`).
3. **Session Creation:** A new session is created for the user.
4. **Redirect:** Users are redirected to the specified URL provided during the initial request or ``/``.

If any error occurs during the process, Better Auth handles it and redirects the user to the error URL (if provided) or the callbackURL. And it includes the error message in the query string `?error=...`.

```
file: ./content/docs/concepts/plugins.mdx
meta: {
  "title": "Plugins",
  "description": "Learn how to use plugins with Better Auth."
}
```

Plugins are a key part of Better Auth, they let you extend the base functionalities. You can use them to add new authentication methods, features, or customize behaviors.

Better Auth offers comes with many built-in plugins ready to use. Check the plugins section for details. You can also create your own plugins.

### ## Using a Plugin

Plugins can be a server-side plugin, a client-side plugin, or both.

To add a plugin on the server, include it in the `plugins`` array in your auth configuration. The plugin will initialize with the provided options.

```
`` `ts title="server.ts"
import { betterAuth } from "better-auth";

export const auth = betterAuth({
  plugins: [
    // Add your plugins here
  ]
});
`` `
```

Client plugins are added when creating the client. Most plugin require both server and client plugins to work correctly. The Better Auth auth client on the frontend uses the `createAuthClient`` function provided by ``better-auth/client``.

```
`` `ts title="auth-client.ts"
import { createAuthClient } from "better-auth/client";

const authClient = createAuthClient({
  plugins: [
    // Add your client plugins here
  ]
});
`` `
```

We recommend keeping the auth-client and your normal auth instance in separate files.

```
<Files>
<Folder name="auth" defaultOpen>
  <File name="server.ts" />

  <File name="auth-client.ts" />
</Folder>
</Files>
```



## ## Creating a Plugin

To get started, you'll need a server plugin.

Server plugins are the backbone of all plugins, and client plugins are there to provide an interface with frontend APIs to easily work with your server plugins.

<Callout type="info">

If your server plugins has endpoints that needs to be called from the client, you'll also need to create a client plugin.

</Callout>

### ### What can a plugin do?

- \* Create custom `endpoint`s to perform any action you want.
- \* Extend database tables with custom `schemas`.
- \* Use a `middleware` to target a group of routes using it's route matcher, and run only when those routes are called through a request.
- \* Use `hooks` to target a specific route or request. And if you want to run the hook even if the endpoint is called directly.
- \* Use `onRequest` or `onResponse` if you want to do something that affects all requests or responses.
- \* Create custom `rate-limit` rule.

## ## Create a Server plugin

To create a server plugin you need to pass an object that satisfies the `BetterAuthPlugin` interface.

The only required property is `id`, which is a unique identifier for the plugin.

Both server and client plugins can use the same `id`.

```

` `` ts title="plugin.ts"
import type { BetterAuthPlugin } from "better-auth";

export const myPlugin = ()=>{
  return {
    id: "my-plugin",
  } satisfies BetterAuthPlugin
}
` ``

```

<Callout>

You don't have to make the plugin a function, but it's recommended to do so. This way you can pass options to the plugin and it's consistent with the built-in plugins.

</Callout>

### ### Endpoints

To add endpoints to the server, you can pass `endpoints` which requires an object with the key being any `string` and the value being an `AuthEndpoint`.

To create an Auth Endpoint you'll need to import `createAuthEndpoint` from `better-auth`.

Better Auth uses wraps around another library called <Link href="https://github.com/bekacru/better-call">Better Call</Link> to create endpoints. Better call is a simple ts web framework made by the same team behind Better Auth.

```

` `` ts title="plugin.ts"
import { createAuthEndpoint } from "better-auth/api";

const myPlugin = ()=> {
  return {
    id: "my-plugin",
    endpoints: {
      getHelloWorld: createAuthEndpoint("/my-plugin/hello-world", {
        method: "GET",
      }, async(ctx) => {
        return ctx.json({
          message: "Hello World"
        })
      })
    }
  }
}

```

```

    })
  })
}
} satisfies BetterAuthPlugin
}
` ``

```

Create Auth endpoints wraps around `createEndpoint` from Better Call. Inside the `ctx` object, it'll provide another object called `context` that give you access better-auth specific contexts including `options`, `db`, `baseURL` and more.

### **\*\*Context Object\*\***

- \* `appName`: The name of the application. Defaults to "Better Auth".
- \* `options`: The options passed to the Better Auth instance.
- \* `tables`: Core tables definition. It is an object which has the table name as the key and the schema definition as the value.
- \* `baseURL`: the baseURL of the auth server. This includes the path. For example, if the server is running on `http://localhost:3000`, the baseURL will be `http://localhost:3000/api/auth` by default unless changed by the user.
- \* `session`: The session configuration. Includes `updateAge` and `expiresIn` values.
- \* `secret`: The secret key used for various purposes. This is defined by the user.
- \* `authCookie`: The default cookie configuration for core auth cookies.
- \* `logger`: The logger instance used by Better Auth.
- \* `db`: The Kysely instance used by Better Auth to interact with the database.
- \* `adapter`: This is the same as db but it give you `orm` like functions to interact with the database. (we recommend using this over `db` unless you need raw sql queries or for performance reasons)
- \* `internalAdapter`: These are internal db calls that are used by Better Auth. For example, you can use these calls to create a session instead of using `adapter` directly. `internalAdapter.createSession(userId)`
- \* `createAuthCookie`: This is a helper function that let's you get a cookie `name` and `options` for either to `set` or `get` cookies. It implements things like `\_\_secure` prefix and `\_\_host` prefix for cookies based on

For other properties, you can check the [Better Call](https://github.com/bekacru/better-call) documentation and the [source code](https://github.com/better-auth/better-auth/blob/main/packages/better-auth/src/init.ts).

### **\*\*Rules for Endpoints\*\***

- \* Makes sure you use kebab-case for the endpoint path
- \* Make sure to only use `POST` or `GET` methods for the endpoints.
- \* Any function that modifies a data should be a `POST` method.
- \* Any function that fetches data should be a `GET` method.
- \* Make sure to use the `createAuthEndpoint` function to create API endpoints.
- \* Make sure your paths are unique to avoid conflicts with other plugins. If you're using a common path, add the plugin name as a prefix to the path. (`/my-plugin/hello-world` instead of `/hello-world`.)

### **### Schema**

You can define a database schema for your plugin by passing a `schema` object. The schema object should have the table name as the key and the schema definition as the value.

```

` `` ts title="plugin.ts"
import { BetterAuthPlugin } from "better-auth/plugins";

const myPlugin = () => {
  return {
    id: "my-plugin",
    schema: {
      myTable: {
        fields: {
          name: {
            type: "string"
          }
        },
      },
      modelName: "myTable" // optional if you want to use a different name than the key
    }
  }
}

```

```
    } satisfies BetterAuthPlugin
  }
  ...

```

### **\*\*Fields\*\***

By default better-auth will create an `id` field for each table. You can add additional fields to the table by adding them to the `fields` object.

The key is the column name and the value is the column definition. The column definition can have the following properties:

`type`: The type of the field. It can be `string`, `number`, `boolean`, `date`.

`required`: if the field should be required on a new record. (default: `false`)

`unique`: if the field should be unique. (default: `false`)

`reference`: if the field is a reference to another table. (default: `null`) It takes an object with the following properties:

\* `model`: The table name to reference.

\* `field`: The field name to reference.

\* `onDelete`: The action to take when the referenced record is deleted. (default: `null`)

### **\*\*Other Schema Properties\*\***

`disableMigration`: if the table should not be migrated. (default: `false`)

```
`` ts title="plugin.ts"
const myPlugin = (opts: PluginOptions)=>{
  return {
    id: "my-plugin",
    schema: {
      rateLimit: {
        fields: {
          key: {
            type: "string",
          },
        },
        disableMigration: opts.storage.provider !== "database", // [!code highlight]
      },
    },
  } satisfies BetterAuthPlugin
}
...

```

if you add additional fields to a `user` or `session` table, the types will be inferred automatically on `getSession` and `signUpEmail` calls.

```
`` ts title="plugin.ts"

const myPlugin = ()=>{
  return {
    id: "my-plugin",
    schema: {
      user: {
        fields: {
          age: {
            type: "number",
          },
        },
      },
    },
  },
} satisfies BetterAuthPlugin
}
...

```

This will add an `age` field to the `user` table and all `user` returning endpoints will include the `age` field and it'll be inferred properly by typescript.

<Callout type="warn">

Don't store sensitive information in `user` or `session` table. Crate a new table if you need to store sensitive information.

### ### Hooks

Hooks are used to run code before or after an action is performed, either from a client or directly on the server. You can add hooks to the server by passing a `hooks` object, which should contain `before` and `after` properties.

```
```ts title="plugin.ts"
import { createAuthMiddleware } from "better-auth/plugins";

const myPlugin = ()=>{
  return {
    id: "my-plugin",
    hooks: {
      before: [{
        matcher: (context)=>{
          return context.headers.get("x-my-header") === "my-value"
        },
        handler: createAuthMiddleware(async(ctx)=>{
          //do something before the request
          return {
            context: ctx // if you want to modify the context
          }
        })
      }],
      after: [{
        matcher: (context)=>{
          return context.path === "/sign-up/email"
        },
        handler: async(ctx)=>{
          return ctx.json({
            message: "Hello World"
          }) // if you want to modify the response
        }
      ]
    }
  }
} satisfies BetterAuthPlugin
```
```

### ### Middleware

You can add middleware to the server by passing a `middlewares` array. This array should contain middleware objects, each with a `path` and a `middleware` property. Unlike hooks, middleware only runs on `api` requests from a client. If the endpoint is invoked directly, the middleware will not run.

The `path` can be either a string or a path matcher, using the same path-matching system as `better-call`.

If you throw an `APIError` from the middleware or returned a `Response` object, the request will be stopped and the response will be sent to the client.

```
```ts title="plugin.ts"
const myPlugin = ()=>{
  return {
    id: "my-plugin",
    middlewares: [
      {
        path: "/my-plugin/hello-world",
        middleware: createAuthMiddleware(async(ctx)=>{

```

```

        //do something
      })
    }
  ]
} satisfies BetterAuthPlugin
}
`

```

### #### On Request & On Response

Additional to middlewares, you can also hook into right before a request is made and right after a response is returned. This is mostly useful if you want to do something that affects all requests or responses.

#### ##### On Request

The ``onRequest`` function is called right before the request is made. It takes two parameters: the ``request`` and the ``context`` object.

Here's how it works:

- \*\*\*Continue as Normal\*\*:** If you don't return anything, the request will proceed as usual.
- \*\*\*Interrupt the Request\*\*:** To stop the request and send a response, return an object with a ``response`` property that contains a ``Response`` object.
- \*\*\*Modify the Request\*\*:** You can also return a modified ``request`` object to change the request before it's sent.

```

` ` ` ts title="plugin.ts"
const myPlugin = () => {
  return {
    id: "my-plugin",
    onRequest: async (request, context) => {
      //do something
    },
  } satisfies BetterAuthPlugin
}
`

```

#### ##### On Response

The ``onResponse`` function is executed immediately after a response is returned. It takes two parameters: the ``response`` and the ``context`` object.

Here's how to use it:

- \*\*\*Modify the Response\*\*:** You can return a modified response object to change the response before it is sent to the client.
- \*\*\*Continue Normally\*\*:** If you don't return anything, the response will be sent as is.

```

` ` ` ts title="plugin.ts"
const myPlugin = () => {
  return {
    id: "my-plugin",
    onResponse: async (response, context) => {
      //do something
    },
  } satisfies BetterAuthPlugin
}
`

```

### #### Rate Limit

You can define custom rate limit rules for your plugin by passing a ``rateLimit`` array. The rate limit array should contain an array of rate limit objects.

```

` ` ` ts title="plugin.ts"
const myPlugin = () => {

```

```

return {
  id: "my-plugin",
  rateLimit: [
    {
      pathMatcher: (path)=>{
        return path === "/my-plugin/hello-world"
      },
      limit: 10,
      window: 60,
    }
  ]
} satisfies BetterAuthPlugin
}
`

```

### Server-plugin helper functions

Some additional helper functions for creating server plugins.

#### ##### `getSessionFromCtx`

Allows you to get the client's session data by passing the auth middleware's `context`.

```

` ` ` ts title="plugin.ts"
import { createAuthMiddleware } from "better-auth/plugins";

const myPlugin = {
  id: "my-plugin",
  hooks: {
    before: [{
      matcher: (context)=>{
        return context.headers.get("x-my-header") === "my-value"
      },
      handler: createAuthMiddleware(async (ctx) => {
        const session = await getSessionFromCtx(ctx);
        //do something with the client's session.

        return {
          context: ctx
        }
      })
    ]},
  }
} satisfies BetterAuthPlugin
`

```

#### ##### `sessionMiddleware`

A middleware that checks if the client has a valid session. If the client has a valid session, it'll add the session data to the context object.

```

` ` ` ts title="plugin.ts"
import { createAuthMiddleware } from "better-auth/plugins";
import { sessionMiddleware } from "better-auth/api";

const myPlugin = ()=>{
  return {
    id: "my-plugin",
    endpoints: {
      getHelloWorld: createAuthEndpoint("/my-plugin/hello-world", {
        method: "GET",
        use: [sessionMiddleware], // [!code highlight]
      }, async(ctx) => {
        const session = ctx.context.session;
        return ctx.json({

```

```

        message: "Hello World"
      })
    })
  }
} satisfies BetterAuthPlugin
}
`

```

### ### Creating a client plugin

If your endpoints needs to be called from the client, you'll need to also create a client plugin. Better Auth clients can infer the endpoints from the server plugins. You can also add additional client side logic.

```

` ` ` ts title="client-plugin.ts"
import type { BetterAuthClientPlugin } from "better-auth";

export const myPluginClient = ()=>{
  return {
    id: "my-plugin",
  } satisfies BetterAuthClientPlugin
}
` ` `

```

### ### Endpoint Interface

Endpoints are inferred from the server plugin by adding a ``$InferServerPlugin`` key to the client plugin.

The client infers the ``path`` as an object and converts kebab-case to camelCase. For example, ``/my-plugin/hello-world`` becomes ``myPlugin.helloWorld``.

```

` ` ` ts title="client-plugin.ts"
import type { BetterAuthClientPlugin } from "better-auth/client";
import type { myPlugin } from "../plugin";

const myPluginClient = () => {
  return {
    id: "my-plugin",
    $InferServerPlugin: {} as ReturnType<typeof myPlugin>,
  } satisfies BetterAuthClientPlugin
}
` ` `

```

### ### Get actions

If you need to add additional methods or what not to the client you can use the ``getActions`` function. This function is called with the ``fetch`` function from the client.

Better Auth uses `<Link href="https://better-fetch.vercel.app"> Better fetch </Link>` to make requests. Better fetch is a simple fetch wrapper made by the same author of Better Auth.

```

` ` ` ts title="client-plugin.ts"
import type { BetterAuthClientPlugin } from "better-auth/client";
import type { myPlugin } from "../plugin";
import type { BetterFetchOption } from "@better-fetch/fetch";

const myPluginClient = {
  id: "my-plugin",
  $InferServerPlugin: {} as ReturnType<typeof myPlugin>,
  getActions: ($fetch)=>{
    return {
      myCustomAction: async (data: {
        foo: string,
      }, fetchOptions?: BetterFetchOption)=>{
        const res = $fetch("/custom/action", {
          method: "POST",

```

```

    body: {
      foo: data.foo
    },
    ...fetchOptions
  })
  return res
}
}
}
} satisfies BetterAuthClientPlugin
``

```

<Callout>

As a general guideline, ensure that each function accepts only one argument, with an optional second argument for `fetchOptions` to allow users to pass additional options to the fetch call. The function should return an object containing data and error keys.

If your use case involves actions beyond API calls, feel free to deviate from this rule.

</Callout>

### ### Get Atoms

This is only useful if you want to provide ``hooks`` like ``useSession``.

Get atoms is called with the ``fetch`` function from better fetch and it should return an object with the atoms. The atoms should be created using [nanostores](https://github.com/nanostores/nanostores). The atoms will be resolved by each framework ``useStore`` hook provided by nanostores.

```

`` ts title="client-plugin.ts"
import { atom } from "nanostores";
import type { BetterAuthClientPlugin } from "better-auth/client";

const myPluginClient = {
  id: "my-plugin",
  $InferServerPlugin: {} as ReturnType<typeof myPlugin>,
  getAtoms: ($fetch)=>{
    const myAtom = atom<null>()
    return {
      myAtom
    }
  }
} satisfies BetterAuthClientPlugin
``

```

See built-in plugins for examples of how to use atoms properly.

### ### Path methods

by default, inferred paths use ``GET`` method if they don't require a body and ``POST`` if they do. You can override this by passing a ``pathMethods`` object. The key should be the path and the value should be the method ("POST" | "GET").

```

`` ts title="client-plugin.ts"
import type { BetterAuthClientPlugin } from "better-auth/client";
import type { myPlugin } from "../plugin";

const myPluginClient = {
  id: "my-plugin",
  $InferServerPlugin: {} as ReturnType<typeof myPlugin>,
  pathMethods: {
    "/my-plugin/hello-world": "POST"
  }
} satisfies BetterAuthClientPlugin
``

```

### ### Fetch plugins



If you need to use better fetch plugins you can pass them to the `fetchPlugins` array. You can read more about better fetch plugins in the [better fetch documentation](https://better-fetch.vercel.app/docs/plugins).

### Atom Listeners

This is only useful if you want to provide `hooks` like `useSession` and you want to listen to atoms and re-evaluate them when they change.

You can see how this is used in the built-in plugins.

```
file: ./content/docs/concepts/rate-limit.mdx
```

```
meta: {
  "title": "Rate Limit",
  "description": "How to limit the number of requests a user can make to the server in a given time period."
}
```

Better Auth includes a built-in rate limiter to help manage traffic and prevent abuse. By default, in production mode, the rate limiter is set to:

- \* Window: 60 seconds
- \* Max Requests: 100 requests

```
<Callout type="warning">
```

```
  Server-side requests made using `auth.api` aren't affected by rate limiting. Rate limits only apply to client-initiated requests.
```

```
</Callout>
```

You can easily customize these settings by passing the rateLimit object to the betterAuth function.

```
```ts title="auth.ts"
import { betterAuth } from "better-auth";

export const auth = betterAuth({
  rateLimit: {
    window: 10, // time window in seconds
    max: 100, // max requests in the window
  },
});
```
```

Rate limiting is disabled in development mode by default. In order to enable it, set `enabled` to `true`:

```
```ts title="auth.ts"
export const auth = betterAuth({
  rateLimit: {
    enabled: true,
    //...other options
  },
});
```
```

In addition to the default settings, Better Auth provides custom rules for specific paths. For example:

\* `/sign-in/email` : Is limited to 3 requests within 10 seconds.

In addition, plugins also define custom rules for specific paths. For example, `twoFactor` plugin has custom rules:

\* `/two-factor/verify` : Is limited to 3 requests within 10 seconds.

These custom rules ensure that sensitive operations are protected with stricter limits.

### Configuring Rate Limit

### Rate Limit Window

```
```ts title="auth.ts"
import { betterAuth } from "better-auth";

export const auth = betterAuth({
  //...other options
  rateLimit: {
    window: 60, // time window in seconds
    max: 100, // max requests in the window
  },
})
```
```

You can also pass custom rules for specific paths.

```
```ts title="auth.ts"
import { betterAuth } from "better-auth";

export const auth = betterAuth({
  //...other options
  rateLimit: {
    window: 60, // time window in seconds
    max: 100, // max requests in the window
    customRules: {
      "/sign-in/email": {
        window: 10,
        max: 3,
      },
      "/two-factor/*": async (request) => {
        // custom function to return rate limit window and max
        return {
          window: 10,
          max: 3,
        }
      }
    },
  },
})
```
```

### Storage

By default, rate limit data is stored in memory, which may not be suitable for many use cases, particularly in serverless environments. To address this, you can use a database, secondary storage, or custom storage for storing rate limit data.

#### \*\*Using Database\*\*

```
```ts title="auth.ts"
import { betterAuth } from "better-auth";

export const auth = betterAuth({
  //...other options
  rateLimit: {
    storage: "database",
    modelName: "rateLimit", //optional by default "rateLimit" is used
  },
})
```
```

Make sure to run `migrate` to create the rate limit table in your database.

```
```bash
npx @better-auth/cli migrate
```
```

**\*\*Using Secondary Storage\*\***

If a [Secondary Storage](/docs/concepts/database#secondary-storage) has been configured you can use that to store rate limit data.

```
```ts title="auth.ts"
import { betterAuth } from "better-auth";

export const auth = betterAuth({
  //...other options
  rateLimit: {
    storage: "secondary-storage"
  },
})
```
```

**\*\*Custom Storage\*\***

If none of the above solutions suits your use case you can implement a `customStorage`.

```
```ts title="auth.ts"
import { betterAuth } from "better-auth";

export const auth = betterAuth({
  //...other options
  rateLimit: {
    customStorage: {
      get: async (key) => {
        // get rate limit data
      },
      set: async (key, value) => {
        // set rate limit data
      },
    },
  },
})
```
```

**## Handling Rate Limit Errors**

When a request exceeds the rate limit, Better Auth returns the following header:

\* `X-Retry-After`: The number of seconds until the user can make another request.

To handle rate limit errors on the client side, you can manage them either globally or on a per-request basis. Since Better Auth clients wrap over Better Fetch, you can pass `fetchOptions` to handle rate limit errors

**\*\*Global Handling\*\***

```
```ts title="auth-client.ts"
import { createAuthClient } from "better-auth/client";

export const authClient = createAuthClient({
  fetchOptions: {
    onError: async (context) => {
      const { response } = context;
      if (response.status === 429) {
        const retryAfter = response.headers.get("X-Retry-After");
        console.log(`Rate limit exceeded. Retry after ${retryAfter} seconds`);
      }
    },
  },
})
```
```

**\*\*Per Request Handling\*\***

```

` `` `ts title="auth-client.ts"
import { authClient } from "../auth-client";

await authClient.signIn.email({
  fetchOptions: {
    onError: async (context) => {
      const { response } = context;
      if (response.status === 429) {
        const retryAfter = response.headers.get("X-Retry-After");
        console.log(`Rate limit exceeded. Retry after ${retryAfter} seconds`);
      }
    },
  },
})
` `` `

```

**#### Schema**

If you are using a database to store rate limit data you need this schema:

Table Name: `rateLimit`

```

<DatabaseTable
  fields=[
    {
      name: "id",
      type: "string",
      description: "Database ID",
      isPrimaryKey: true
    },
    {
      name: "key",
      type: "string",
      description: "Unique identifier for each rate limit key",
    },
    {
      name: "count",
      type: "integer",
      description: "Time window in seconds"
    },
    {
      name: "lastRequest",
      type: "bigint",
      description: "Max requests in the window"
    }
  ]
/>

```

```

file: ./content/docs/concepts/session-management.mdx
meta: {
  "title": "Session Management",
  "description": "Better Auth session management."
}

```

Better Auth manages session using a traditional cookie-based session management. The session is stored in a cookie and is sent to the server on every request. The server then verifies the session and returns the user data if the session is valid.

**## Session table**

The session table stores the session data. The session table has the following fields:

\* `id` : The session token. Which is also used as the session cookie.

- \* `userId`: The user ID of the user.
- \* `expiresAt`: The expiration date of the session.
- \* `ipAddress`: The IP address of the user.
- \* `userAgent`: The user agent of the user. It stores the user agent header from the request.

### ## Session Expiration

The session expires after 7 days by default. But whenever the session is used and the `updateAge` is reached, the session expiration is updated to the current time plus the `expiresIn` value.

You can change both the `expiresIn` and `updateAge` values by passing the `session` object to the `auth` configuration.

```
```ts title="auth.ts"
import { betterAuth } from "better-auth"

export const auth = betterAuth({
  //... other config options
  session: {
    expiresIn: 60 * 60 * 24 * 7, // 7 days
    updateAge: 60 * 60 * 24 // 1 day (every 1 day the session expiration is updated)
  }
})
```
```

### ### Disable Session Refresh

You can disable session refresh so that the session is not updated regardless of the `updateAge` option.

```
```ts title="auth.ts"
import { betterAuth } from "better-auth"

export const auth = betterAuth({
  //... other config options
  session: {
    disableSessionRefresh: true
  }
})
```
```

### ## Session Freshness

Some endpoints in Better Auth require the session to be **fresh**. A session is considered fresh if its `createdAt` is within the `freshAge` limit. By default, the `freshAge` is set to **1 day** (60 \* 60 \* 24).

You can customize the `freshAge` value by passing a `session` object in the `auth` configuration:

```
```ts title="auth.ts"
import { betterAuth } from "better-auth"

export const auth = betterAuth({
  //... other config options
  session: {
    freshAge: 60 * 5 // 5 minutes (the session is fresh if created within the last 5 minutes)
  }
})
```
```

To **disable the freshness check**, set `freshAge` to `0`:

```
```ts title="auth.ts"
import { betterAuth } from "better-auth"

export const auth = betterAuth({
  //... other config options
  session: {
    freshAge: 0
  }
})
```
```

```

    freshAge: 0 // Disable freshness check
  }
})
```

```

## ## Session Management

Better Auth provides a set of functions to manage sessions.

### ### Get Session

The `getSession` function retrieves the current active session.

```

```ts client="client.ts"
import { authClient } from "@lib/client"

const { data: session } = await authClient.getSession()
```

```

To learn how to customize the session response check the [Customizing Session Response](#customizing-session-response) section.

### ### Use Session

The `useSession` action provides a reactive way to access the current session.

```

```ts client="client.ts"
import { authClient } from "@lib/client"

const { data: session } = authClient.useSession()
```

```

### ### List Sessions

The `listSessions` function returns a list of sessions that are active for the user.

```

```ts title="auth-client.ts"
import { authClient } from "@lib/client"

const sessions = await authClient.listSessions()
```

```

### ### Revoke Session

When a user signs out of a device, the session is automatically ended. However, you can also end a session manually from any device the user is signed into.

To end a session, use the `revokeSession` function. Just pass the session token as a parameter.

```

```ts title="auth-client.ts"
await authClient.revokeSession({
  token: "session-token"
})
```

```

### ### Revoke Other Sessions

To revoke all other sessions except the current session, you can use the `revokeOtherSessions` function.

```

```ts title="auth-client.ts"
await authClient.revokeOtherSessions()
```

```

### ### Revoke All Sessions

To revoke all sessions, you can use the `revokeSessions` function.

```
```ts title="auth-client.ts"
await authClient.revokeSessions()
```
```

### ### Revoking Sessions on Password Change

You can revoke all sessions when the user changes their password by passing `revokeOtherSessions` as true on `changePassword` function.

```
```ts title="auth.ts"
await authClient.changePassword({
  newPassword: newPassword,
  currentPassword: currentPassword,
  revokeOtherSessions: true,
})
```
```

## ## Session Caching

### ### Cookie Cache

Calling your database every time `useSession` or `getSession` invoked isn't ideal, especially if sessions don't change frequently. Cookie caching handles this by storing session data in a short-lived, signed cookie—similar to how JWT access tokens are used with refresh tokens.

When cookie caching is enabled, the server can check session validity from the cookie itself instead of hitting the database each time. The cookie is signed to prevent tampering, and a short `maxAge` ensures that the session data gets refreshed regularly. If a session is revoked or expires, the cookie will be invalidated automatically.

To turn on cookie caching, just set `session.cookieCache` in your auth config:

```
```ts title="auth.ts"
import { betterAuth } from "better-auth"

export const auth = betterAuth({
  session: {
    cookieCache: {
      enabled: true,
      maxAge: 5 * 60 // Cache duration in seconds
    }
  }
});
```
```

If you want to disable returning from the cookie cache when fetching the session, you can pass `disableCookieCache:true` this will force the server to fetch the session from the database and also refresh the cookie cache.

```
```ts title="auth-client.ts"
const session = await authClient.getSession({ query: {
  disableCookieCache: true
}})
```
```

or on the server

```
```ts title="server.ts"
await auth.api.getSession({
  query: {
    disableCookieCache: true,
  },
  headers: req.headers, // pass the headers
});
```
```

## ## Customizing Session Response

When you call `getSession`` or `useSession``, the session data is returned as a ``user`` and ``session`` object. You can customize this response using the `customSession`` plugin.

```
```ts title="auth.ts"
import { customSession } from "better-auth/plugins";

export const auth = betterAuth({
  plugins: [
    customSession(async ({ user, session }) => {
      const roles = findUserRoles(session.session.userId);
      return {
        roles,
        user: {
          ...user,
          newField: "newField",
        },
        session
      };
    })
  ],
});
```
```

This will add `roles`` and `user.newField`` to the session response.

### \*\*Infer on the Client\*\*

```
```ts title="auth-client.ts"
import { customSessionClient } from "better-auth/client/plugins";
import type { auth } from "@lib/auth"; // Import the auth instance as a type

const authClient = createAuthClient({
  plugins: [customSessionClient<typeof auth>()],
});

const { data } = authClient.useSession();
const { data: sessionData } = await authClient.getSession();
// data.roles
// data.user.newField
```
```

### \*\*Some Caveats\*\*:

\* The passed `session`` object to the callback does not infer fields added by plugins.

However, as a workaround, you can pull up your auth options and pass it to the plugin to infer the fields.

```
```ts
import { betterAuth, BetterAuthOptions } from "better-auth";

const options = {
  //...config options
  plugins: [
    //...plugins
  ]
} satisfies BetterAuthOptions;

export const auth = betterAuth({
  ...options,
  plugins: [
    ...(options.plugins ?? []),
    customSession(async ({ user, session }, ctx) => {
```



```

    // now both user and session will infer the fields added by plugins and your custom fields
    return {
      user,
      session
    }
  }, options), // pass options here // [!code highlight]
]
}))
```

```

\* If you cannot use the `auth` instance as a type, inference will not work on the client.

\* Session caching, including secondary storage or cookie cache, does not include custom fields. Each time the session is fetched, your custom session function will be called.

```

file: ./content/docs/concepts/typescript.mdx
meta: {
  "title": "TypeScript",
  "description": "Better Auth TypeScript integration."
}

```

Better Auth is designed to be type-safe. Both the client and server are built with TypeScript, allowing you to easily infer types.

## ## TypeScript Config

### ### Strict Mode

Better Auth is designed to work with TypeScript's strict mode. We recommend enabling strict mode in your TypeScript config file:

```

```json title="tsconfig.json"
{
  "compilerOptions": {
    "strict": true
  }
}
```

```

if you can't set `strict` to `true`, you can enable `strictNullChecks`:

```

```json title="tsconfig.json"
{
  "compilerOptions": {
    "strictNullChecks": true,
  }
}
```

```

## ## Inferring Types

Both the client SDK and the server offer types that can be inferred using the `\$Infer` property. Plugins can extend base types like `User` and `Session`, and you can use `\$Infer` to infer these types. Additionally, plugins can provide extra types that can also be inferred through `\$Infer`.

```

```ts title="auth-client.ts"
import { createAuthClient } from "better-auth/client"

const authClient = createAuthClient()

export type Session = typeof authClient.$Infer.Session
```

```

The `Session` type includes both `session` and `user` properties. The user property represents the user object type, and the `session` property represents the `session` object type.

You can also infer types on the server side.

```
```ts
title="auth.ts"
import { betterAuth } from "better-auth"
import Database from "better-sqlite3"

export const auth = betterAuth({
  database: new Database("database.db")
})

type Session = typeof auth.$Infer.Session
```
```

### ### Additional Fields

Better Auth allows you to add additional fields to the user and session objects. All additional fields are properly inferred and available on the server and client side.

```
```ts
import { betterAuth } from "better-auth"
import Database from "better-sqlite3"

export const auth = betterAuth({
  database: new Database("database.db"),
  user: {
    additionalFields: {
      role: {
        type: "string"
      }
    }
  }
})

type Session = typeof auth.$Infer.Session
```
```

In the example above, we added a `role` field to the user object. This field is now available on the `Session` type.

### ### Inferring Additional Fields on Client

To make sure proper type inference for additional fields on the client side, you need to inform the client about these fields. There are two approaches to achieve this, depending on your project structure:

#### 1. For Monorepo or Single-Project Setups

If your server and client code reside in the same project, you can use the `inferAdditionalFields` plugin to automatically infer the additional fields from your server configuration.

```
```ts
import { inferAdditionalFields } from "better-auth/client/plugins";
import { createAuthClient } from "better-auth/react";
import type { auth } from "../auth";

export const authClient = createAuthClient({
  plugins: [inferAdditionalFields<typeof auth>()],
});
```
```

#### 2. For Separate Client-Server Projects

If your client and server are in separate projects, you'll need to manually specify the additional fields when creating the auth client.

```

```ts
import type { auth } from "./auth";
import { inferAdditionalFields } from "better-auth/client/plugins";

export const authClient = createAuthClient({
  plugins: [inferAdditionalFields({
    user: {
      role: {
        type: "string"
      }
    }
  })],
});
```

```

```

file: ./content/docs/concepts/users-accounts.mdx
meta: {
  "title": "User & Accounts",
  "description": "User and account management."
}

```

Beyond authenticating users, Better Auth also provides a set of methods to manage users. This includes, updating user information, changing passwords, and more.

The user table stores the authentication data of the user [\[Click here to view the schema\]\(/docs/concepts/database#user\)](/docs/concepts/database#user).

The user table can be extended using [\[additional fields\]\(/docs/concepts/database#extending-core-schema\)](/docs/concepts/database#extending-core-schema) or by plugins to store additional data.

## ## Update User

### ### Update User Information

To update user information, you can use the `updateUser` function provided by the client. The `updateUser` function takes an object with the following properties:

```

```ts
await authClient.updateUser({
  image: "https://example.com/image.jpg",
  name: "John Doe",
})
```

```

### ### Change Email

To allow users to change their email, first enable the `changeEmail` feature, which is disabled by default. Set `changeEmail.enabled` to `true`:

```

```ts
export const auth = betterAuth({
  user: {
    changeEmail: {
      enabled: true,
    }
  }
})
```

```

For users with a verified email, provide the `sendChangeEmailVerification` function. This function triggers when a user changes their email, sending a verification email with a URL and token. If the current email isn't verified, the change happens immediately without verification.

```

```ts
export const auth = betterAuth({

```

```

user: {
  changeEmail: {
    enabled: true,
    sendChangeEmailVerification: async ({ user, newEmail, url, token }, request) => {
      await sendEmail({
        to: user.email, // verification email must be sent to the current user email to approve the change
        subject: 'Approve email change',
        text: `Click the link to approve the change: ${url}`
      })
    }
  }
}
})
```

```

Once enabled, use the `changeEmail` function on the client to update a user's email. The user must verify their current email before changing it.

```

```ts
await authClient.changeEmail({
  newEmail: "new-email@email.com",
  callbackURL: "/dashboard", //to redirect after verification
});
```

```

After verification, the new email is updated in the user table, and a confirmation is sent to the new address.

<Callout type="warn">

If the current email is unverified, the new email is updated without the verification step.

</Callout>

### ### Change Password

Password of a user isn't stored in the user table. Instead, it's stored in the account table. To change the password of a user, you can use the `changePassword` function provided by the client. The `changePassword` function takes an object with the following properties:

```

```ts
await authClient.changePassword({
  newPassword: "newPassword123",
  currentPassword: "oldPassword123",
  revokeOtherSessions: true, // revoke all other sessions the user is signed into
});
```

```

### ### Set Password

If a user was registered using OAuth or other providers, they won't have a password or a credential account. In this case, you can use the `setPassword` action to set a password for the user. For security reasons, this function can only be called from the server. We recommend having users go through a 'forgot password' flow to set a password for their account.

```

```ts
await auth.api.setPassword({
  body: { newPassword: "password" },
  headers: // headers containing the user's session token
});
```

```

### ## Delete User

Better Auth provides a utility to hard delete a user from your database. It's disabled by default, but you can enable it easily by passing `enabled:true`

```

```ts
export const auth = betterAuth({

```

```
//...other config
user: {
  deleteUser: { // [!code highlight]
    enabled: true // [!code highlight]
  } // [!code highlight]
}
})
```,
```

Once enabled, you can call `authClient.deleteUser`` to permanently delete user data from your database.

### ### Adding Verification Before Deletion

For added security, you'll likely want to confirm the user's intent before deleting their account. A common approach is to send a verification email. Better Auth provides a `sendDeleteAccountVerification`` utility for this purpose. This is especially needed if you have OAuth setup and want them to be able to delete their account without forcing them to login again for a fresh session.

Here's how you can set it up:

```
```ts
export const auth = betterAuth({
  user: {
    deleteUser: {
      enabled: true,
      sendDeleteAccountVerification: async (
        {
          user, // The user object
          url, // The auto-generated URL for deletion
          token // The verification token (can be used to generate custom URL)
        },
        request // The original request object (optional)
      ) => {
        // Your email sending logic here
        // Example: sendEmail(data.user.email, "Verify Deletion", data.url);
      },
    },
  },
});
```,
```

### \*\*How callback verification works:\*\*

**\*\*\*Callback URL\*\*:** The URL provided in `sendDeleteAccountVerification`` is a pre-generated link that deletes the user data when accessed.

```
```ts title="delete-user.ts"
await authClient.deleteUser({
  callbackURL: "/goodbye" // you can provide a callback URL to redirect after deletion
});
```,
```

**\*\*\*Authentication Check\*\*:** The user must be signed in to the account they're attempting to delete. If they aren't signed in, the deletion process will fail.

If you have sent a custom URL, you can use the `deleteUser`` method with the token to delete the user.

```
```ts title="delete-user.ts"
await authClient.deleteUser({
  token
});
```,
```

### ### Authentication Requirements

To delete a user, the user must meet one of the following requirements:

### 1. A valid password

if the user has a password, they can delete their account by providing the password.

```
```ts title="delete-user.ts"
await authClient.deleteUser({
  password: "password"
});
```
```

### 2. Fresh session

The user must have a `fresh` session token, meaning the user must have signed in recently. This is checked if the password is not provided.

<Callout type="warn">

By default `session.freshAge` is set to `60 \* 60 \* 24` (1 day). You can change this value by passing the `session` object to the `auth` configuration. If it is set to `0`, the freshness check is disabled. It is recommended not to disable this check if you are not using email verification for deleting the account.

</Callout>

```
```ts title="delete-user.ts"
await authClient.deleteUser();
```
```

### 3. Enabled email verification (needed for OAuth users)

As OAuth users don't have a password, we need to send a verification email to confirm the user's intent to delete their account. If you have already added the `sendDeleteAccountVerification` callback, you can just call the `deleteUser` method without providing any other information.

Note that this would fail if they have a password. In that case, you need to provide the password to delete the account.

```
```ts title="delete-user.ts"
await authClient.deleteUser({});
```
```

### 4. If you have a custom delete account page and sent that url via the `sendDeleteAccountVerification` callback. Then you need to call the `deleteUser` method with the token to complete the deletion.

```
```ts title="delete-user.ts"
await authClient.deleteUser({
  token
});
```
```

### ### Callbacks

**\*\*beforeDelete\*\***: This callback is called before the user is deleted. You can use this callback to perform any cleanup or additional checks before deleting the user.

```
```ts title="auth.ts"
export const auth = betterAuth({
  user: {
    deleteUser: {
      enabled: true,
      beforeDelete: async (user) => {
        // Perform any cleanup or additional checks here
      },
    },
  },
});
```
```

you can also throw `APIError` to interrupt the deletion process.

```
```ts title="auth.ts"
import { betterAuth } from "better-auth";
import { APIError } from "better-auth/api";

export const auth = betterAuth({
  user: {
    deleteUser: {
      enabled: true,
      beforeDelete: async (user, request) => {
        if (user.email.includes("admin")) {
          throw new APIError("BAD_REQUEST", {
            message: "Admin accounts can't be deleted",
          });
        }
      },
    },
  },
});
```
```

**\*\*afterDelete\*\***: This callback is called after the user is deleted. You can use this callback to perform any cleanup or additional actions after the user is deleted.

```
```ts title="auth.ts"
export const auth = betterAuth({
  user: {
    deleteUser: {
      enabled: true,
      afterDelete: async (user, request) => {
        // Perform any cleanup or additional actions here
      },
    },
  },
});
```
```

### ### Accounts

Better Auth supports multiple authentication methods. Each authentication method is called a provider. For example, email and password authentication is a provider, Google authentication is a provider, etc.

When a user signs in using a provider, an account is created for the user. The account stores the authentication data returned by the provider. This data includes the access token, refresh token, and other information returned by the provider.

The account table stores the authentication data of the user [Click here to view the schema] (/docs/concepts/database#account)

### ### List User Accounts

To list user accounts you can use `client.user.listAccounts` method. Which will return all accounts associated with a user.

```
```ts
const accounts = await authClient.listAccounts();
```
```

### ### Token Encryption

Better Auth doesn't encrypt tokens by default and that's intentional. We want you to have full control over how encryption and decryption are handled, rather than baking in behavior that could be confusing or limiting. If you need to store encrypted tokens (like accessToken or refreshToken), you can use databaseHooks to encrypt them before they're saved to your database.

```

```ts
export const auth = betterAuth({
  databaseHooks: {
    account: {
      create: {
        before(account, context) {
          const withEncryptedTokens = { ...account };
          if (account.accessToken) {
            const encryptedAccessToken = encrypt(account.accessToken) // [!code highlight]
            withEncryptedTokens.accessToken = encryptedAccessToken;
          }
          if (account.refreshToken) {
            const encryptedRefreshToken = encrypt(account.refreshToken); // [!code highlight]
            withEncryptedTokens.refreshToken = encryptedRefreshToken;
          }
          return {
            data: resultAccount
          }
        },
      },
    },
  },
});
```

```

Then whenever you retrieve back the account make sure to decrypt the tokens before using them.

### ### Account Linking

Account linking enables users to associate multiple authentication methods with a single account. With Better Auth, users can connect additional social sign-ons or OAuth providers to their existing accounts if the provider confirms the user's email as verified.

If account linking is disabled, no accounts can be linked, regardless of the provider or email verification status.

```

```ts title="auth.ts"
export const auth = betterAuth({
  account: {
    accountLinking: {
      enabled: true,
    },
  },
});
```

```

### #### Forced Linking

You can specify a list of "trusted providers." When a user logs in using a trusted provider, their account will be automatically linked even if the provider doesn't confirm the email verification status. Use this with caution as it may increase the risk of account takeover.

```

```ts title="auth.ts"
export const auth = betterAuth({
  account: {
    accountLinking: {
      enabled: true,
      trustedProviders: ["google", "github"]
    },
  },
});
```

```

### #### Manually Linking Accounts

Users already signed in can manually link their account to additional social providers or credential-based accounts.



**\*\*Linking Social Accounts:\*\*** Use the `linkSocial` method on the client to link a social provider to the user's account.

```
```ts
await authClient.linkSocial({
  provider: "google", // Provider to link
  callbackURL: "/callback" // Callback URL after linking completes
});
```
```

You can also request specific scopes when linking a social account, which can be different from the scopes used during the initial authentication:

```
```ts
await authClient.linkSocial({
  provider: "google",
  callbackURL: "/callback",
  scopes: ["https://www.googleapis.com/auth/drive.readonly"] // Request additional scopes
});
```
```

If you want your users to be able to link a social account with a different email address than the user, or if you want to use a provider that does not return email addresses, you will need to enable this in the account linking settings.

```
```ts title="auth.ts"
export const auth = betterAuth({
  account: {
    accountLinking: {
      allowDifferentEmails: true
    }
  },
});
```
```

**\*\*Linking Credential-Based Accounts:\*\*** To link a credential-based account (e.g., email and password), users can initiate a "forgot password" flow, or you can call the `setPassword` method on the server.

```
```ts
await auth.api.setPassword({
  headers: /* headers containing the user's session token */,
  password: /* new password */
});
```
```

<Callout>  
`setPassword` can't be called from the client for security reasons.  
 </Callout>

### ### Account Unlinking

You can unlink a user account by providing a `providerId`.

```
```ts
await authClient.unlinkAccount({
  providerId: "google"
});

// Unlink a specific account
await authClient.unlinkAccount({
  providerId: "google",
  accountId: "123"
});
```
```

If the account doesn't exist, it will throw an error. Additionally, if the user only has one account, the unlinking process will

fail to prevent account lockout unless `allowUnlinkingAll` is set to `true`.

```
```ts title="auth.ts"
export const auth = betterAuth({
  account: {
    accountLinking: {
      allowUnlinkingAll: true
    }
  },
});
```
```

```
file: ./content/docs/examples/astro.mdx
meta: {
  "title": "Astro Example",
  "description": "Better Auth Astro example."
}
```

This is an example of how to use Better Auth with Astro. It uses Solid for building the components.

**\*\*Implements the following features:\*\***

Email & Password . Social Sign-in with Google . Passkeys . Email Verification . Password Reset . Two Factor Authentication . Profile Update . Session Management

<ForkButton url="better-auth/better-auth/tree/main/examples/astro-example" />

```
<iframe
  src="https://stackblitz.com/github/better-auth/better-auth/tree/main/examples/astro-example?
codemirror=1&fontsize=14&hidenavigation=1&runonclick=1&hidedevtools=1"
  style={{
    width: "100%",
    height: "500px",
    border: 0,
    borderRadius: "4px",
    overflow: "hidden"
  }}
  title="Better Auth Astro+Solid Example"
  allow="accelerometer; ambient-light-sensor; camera; encrypted-media; geolocation; gyroscope; hid; microphone; midi;
payment; usb; vr; xr-spatial-tracking"
  sandbox="allow-forms allow-modals allow-popups allow-presentation allow-same-origin allow-scripts"
/>
```

### ## How to run

1. Clone the code sandbox (or the repo) and open it in your code editor

2. Provide .env file with the following variables

```
```txt
GOOGLE_CLIENT_ID=
GOOGLE_CLIENT_SECRET=
BETTER_AUTH_SECRET=
```
```

//if you don't have these, you can get them from the google developer console. If you don't want to use google sign-in, you can remove the google config from the `auth.ts` file.

3. Run the following commands

```
```bash
pnpm install
pnpm run dev
```
```

4. Open the browser and navigate to `http://localhost:3000`

```
file: ./content/docs/examples/next-js.mdx
meta: {
  "title": "Next.js Example",
  "description": "Better Auth Next.js example."
}
```

This is an example of how to use Better Auth with Next.

**\*\*Implements the following features:\*\***

Email & Password . Social Sign-in . Passkeys . Email Verification . Password Reset . Two Factor Authentication . Profile Update . Session Management . Organization, Members and Roles

See [Demo](https://demo.better-auth.com)

```
<ForkButton url="better-auth/better-auth/tree/main/demo/nextjs" />
```

```
<iframe
  src="https://stackblitz.com/github/better-auth/better-auth/tree/main/demo/nextjs?
codemirror=1&fontsize=14&hidenavigation=1&runonclick=1&hidedevtools=1"
  style={{
    width: "100%",
    height: "500px",
    border: 0,
    borderRadius: "4px",
    overflow: "hidden"
  }}
  title="Better Auth Next.js Example"
  allow="accelerometer; ambient-light-sensor; camera; encrypted-media; geolocation; gyroscope; hid; microphone; midi;
payment; usb; vr; xr-spatial-tracking"
  sandbox="allow-forms allow-modals allow-popups allow-presentation allow-same-origin allow-scripts"
/>
```

### ## How to run

1. Clone the code sandbox (or the repo) and open it in your code editor
2. Move .env.example to .env and provide necessary variables
3. Run the following commands

```
```bash
pnpm install
pnpm dev
```
```

4. Open the browser and navigate to `http://localhost:3000`

```
file: ./content/docs/examples/nuxt.mdx
meta: {
  "title": "Nuxt Example",
  "description": "Better Auth Nuxt example."
}
```

This is an example of how to use Better Auth with Nuxt.

**\*\*Implements the following features:\*\***

Email & Password . Social Sign-in with Google

```
<ForkButton url="better-auth/better-auth/tree/main/examples/nuxt-example" />
```

```
<iframe
  src="https://stackblitz.com/github/better-auth/better-auth/tree/main/examples/nuxt-example?
codemirror=1&fontsize=14&hidenavigation=1&runonclick=1&hidedevtools=1"
  style={{
    width: "100%",
    height: "500px",
    border: 0,
    borderRadius: "4px",
```

```

      overflow: "hidden"
    }}
    title="Better Auth Nuxt Example"
    allow="accelerometer; ambient-light-sensor; camera; encrypted-media; geolocation; gyroscope; hid; microphone; midi;
payment; usb; vr; xr-spatial-tracking"
    sandbox="allow-forms allow-modals allow-popups allow-presentation allow-same-origin allow-scripts"
  />

```

### ## How to run

1. Clone the code sandbox (or the repo) and open it in your code editor
2. Move .env.example to .env and provide necessary variables
3. Run the following commands

```

```bash
pnpm install
pnpm dev
```

```

4. Open the browser and navigate to `http://localhost:3000`

```

file: ./content/docs/examples/remix.mdx
meta: {
  "title": "Remix Example",
  "description": "Better Auth Remix example."
}

```

This is an example of how to use Better Auth with Remix.

**\*\*Implements the following features:\*\***

Email & Password . Social Sign-in with Google . Passkeys . Email Verification . Password Reset . Two Factor Authentication . Profile Update . Session Management

```
<ForkButton url="better-auth/better-auth/tree/main/examples/remix-example" />
```

```

<iframe
  src="https://stackblitz.com/github/better-auth/better-auth/tree/main/examples/remix-example?
codemirror=1&fontsize=14&hidenavigation=1&runonclick=1&hidedevtools=1"
  style={{
    width: "100%",
    height: "500px",
    border: 0,
    borderRadius: "4px",
    overflow: "hidden"
  }}
  title="Better Auth Remix Example"
  allow="accelerometer; ambient-light-sensor; camera; encrypted-media; geolocation; gyroscope; hid; microphone; midi;
payment; usb; vr; xr-spatial-tracking"
  sandbox="allow-forms allow-modals allow-popups allow-presentation allow-same-origin allow-scripts"
/>

```

### ## How to run

1. Clone the code sandbox (or the repo) and open it in your code editor
2. Provide .env file with by copying the `.env.example` file and adding the variables
3. Run the following commands

```

```bash
pnpm install
pnpm run dev
```

```

4. Open the browser and navigate to `http://localhost:3000`

```

file: ./content/docs/examples/svelte-kit.mdx
meta: {
  "title": "SvelteKit Example",

```

```
"description": "Better Auth SvelteKit example."
}
```

This is an example of how to use Better Auth with SvelteKit.

**\*\*Implements the following features:\*\***

Email & Password . [Social Sign-in with Google](#) . Passkeys . Email Verification . Password Reset . Two Factor Authentication . Profile Update . Session Management

```
<ForkButton url="better-auth/better-auth/tree/main/examples/svelte-kit-example" />
```

```
<iframe
  src="https://stackblitz.com/github/better-auth/better-auth/tree/main/examples/svelte-kit-example?
codemirror=1&fontsize=14&hidenavigation=1&runonclick=1&hidedevtools=1"
  style={{
    width: "100%",
    height: "500px",
    border: 0,
    borderRadius: "4px",
    overflow: "hidden"
  }}
  title="Better Auth SvelteKit Example"
  allow="accelerometer; ambient-light-sensor; camera; encrypted-media; geolocation; gyroscope; hid; microphone; midi;
payment; usb; vr; xr-spatial-tracking"
  sandbox="allow-forms allow-modals allow-popups allow-presentation allow-same-origin allow-scripts"
/>
```

### ## How to run

1. Clone the code sandbox (or the repo) and open it in your code editor
2. Move .env.example to .env and provide necessary variables
3. Run the following commands

```
```bash
pnpm install
pnpm dev
```
```

4. Open the browser and navigate to `http://localhost:3000`

```
file: ./content/docs/guides/browser-extension-guide.mdx
meta: {
  "title": "Browser Extension Guide",
  "description": "A step-by-step guide to creating a browser extension with Better Auth."
}
```

In this guide, we'll walk you through the steps of creating a browser extension using [Plasmo](https://docs.plasmo.com/) with Better Auth for authentication.

If you would like to view a completed example, you can check out the [browser extension example](https://github.com/better-auth/better-auth/tree/main/examples/browser-extension-example).

```
<Callout type="warn">
```

The Plasmo framework does not provide a backend for the browser extension.

This guide assumes you have{" "}

[a backend setup](/docs/integrations/hono) of Better Auth and are ready to create a browser extension to connect to it.

```
</Callout>
```

```
<Steps>
```

```
<Step>
```

### ## Setup & Installations

Initialize a new Plasmo project with TailwindCSS and a src directory.

```
```bash
```

```
pnpm create plasmo --with-tailwindcss --with-src
```

```

Then, install the Better Auth package.

```
```bash
pnpm add better-auth
```
```

To start the Plasmo development server, run the following command.

```
```bash
pnpm dev
```
```

</Step>

<Step>

## Configure tsconfig

Configure the `tsconfig.json` file to include `strict` mode.

For this demo, we have also changed the import alias from `~` to `@` and set it to the `src` directory.

```
```json title="tsconfig.json"
{
  "compilerOptions": {
    "paths": {
      "@/_": [
        "./src/_/"
      ]
    },
    "strict": true,
    "baseUrl": "."
  }
}
```
```

</Step>

<Step>

## Create the client auth instance

Create a new file at `src/auth/auth-client.ts` and add the following code.

```
<Files>
  <Folder name="src" defaultOpen>
    <Folder name="auth" defaultOpen>
      <File name="auth-client.ts" />
    </Folder>
  </Folder>
</Files>
```

```
```ts title="auth-client.ts"
import { createAuthClient } from "better-auth/react"

export const authClient = createAuthClient({
  baseUrl: "http://localhost:3000" /* Base URL of your Better Auth backend. */,
  plugins: [],
});
```
```

</Step>

<Step>

## Configure the manifest

We must ensure the extension knows the URL to the Better Auth backend.

Head to your package.json file, and add the following code.

```
```\njson title="package.json"\n{\n  //...\n  "manifest": {\n    "host_permissions": [\n      "https://URL_TO_YOUR_BACKEND" // localhost works too (e.g. http://localhost:3000)\n    ]\n  }\n}\n```\n</Step>
```

<Step>  
## You're now ready!

You have now set up Better Auth for your browser extension.

Add your desired UI and create your dream extension!

To learn more about the client Better Auth API, check out the <Link href="/docs/concepts/client">client documentation</Link>.

Here's a quick example 🕶️

```
```\ntsx title="src/popup.tsx"\nimport { authClient } from "../auth/auth-client"\n\nfunction IndexPopup() {\n  const {data, isPending, error} = authClient.useSession();\n  if(isPending){\n    return <>Loading...</>\n  }\n  if(error){\n    return <>Error: {error.message}</>\n  }\n  if(data){\n    return <>Signed in as {data.user.name}</>\n  }\n}\n\nexport default IndexPopup;\n```\n</Step>
```

<Step>  
## Bundle your extension

To get a production build, run the following command.

```
```\nbash\npnpm build\n```\n
```

Head over to <Link href="chrome://extensions" target="\_blank">chrome://extensions</Link> and enable developer mode.



Click on "Load Unpacked" and navigate to your extension's `build/chrome-mv3-dev` (or `build/chrome-mv3-prod`) directory.

To see your popup, click on the puzzle piece icon on the Chrome toolbar, and click on your extension.

Learn more about [bundling your extension here](https://docs.plasmo.com/framework#loading-the-extension-in-chrome).

</Step>


<Step>

## Configure the server auth instance

First, we will need your extension URL.

An extension URL formed like this: `chrome-extension://YOUR\_EXTENSION\_ID`.

You can find your extension ID at <chrome://extensions>.

 />

Head to your server's auth file, and make sure that your extension's URL is added to the `trustedOrigins` list.

```
```ts title="server.ts"
import { betterAuth } from "better-auth"
import { auth } from "@auth/auth"

export const auth = betterAuth({
  trustedOrigins: ["chrome-extension://YOUR_EXTENSION_ID"],
})
```
```

If you're developing multiple extensions or need to support different browser extensions with different IDs, you can use wildcard patterns:

```
```ts title="server.ts"
export const auth = betterAuth({
  trustedOrigins: [
    // Support a specific extension ID
    "chrome-extension://YOUR_EXTENSION_ID",

    // Or support multiple extensions with wildcard (less secure)
    "chrome-extension://*",
  ],
})
```
```

<Callout type="warn">

Using wildcards for extension origins (`chrome-extension://\*`) reduces security by trusting all extensions.

It's safer to explicitly list each extension ID you trust. Only use wildcards for development and testing.

</Callout>

</Step>

<Step>

## That's it!

Everything is set up! You can now start developing your extension. 🎉

</Step>

</Steps>

## ## Wrapping Up

Congratulations! You've successfully created a browser extension using Better Auth and Plasmo.

We highly recommend you visit the [Plasmo documentation](https://docs.plasmo.com/) to learn more about the framework.

If you would like to view a completed example, you can check out the [browser extension example](https://github.com/better-auth/better-auth/tree/main/examples/browser-extension-example).



If you have any questions, feel free to open an issue on our [GitHub repo](https://github.com/better-auth/better-auth/issues), or join our [Discord server](https://discord.gg/better-auth) for support.

file: ./content/docs/guides/clerk-migration-guide.mdx

```
meta: {
  "title": "Migrating from Clerk to Better Auth",
  "description": "A step-by-step guide to transitioning from Clerk to Better Auth."
}
```

In this guide, we'll walk through the steps to migrate a project from Clerk to Better Auth — including email/password with proper hashing, social/external accounts, phone number, two-factor data, and more.

<Callout type="warn">

This migration will invalidate all active sessions. This guide doesn't currently show you how to migrate Organization but it should be possible with additional steps and the [Organization](/docs/plugins/organization) Plugin.

</Callout>

## ## Before You Begin

Before starting the migration process, set up Better Auth in your project. Follow the [installation guide](/docs/installation) to get started. And go to

<Steps>

<Step>

#### Connect to your database

You'll need to connect to your database to migrate the users and accounts. You can use any database you want, but for this example, we'll use PostgreSQL.

<Tabs groupId="package-manager" persist items={}>

<Tab value="npm">

```bash

npm install pg

```

</Tab>

<Tab value="pnpm">

```bash

pnpm add pg

```

</Tab>

<Tab value="yarn">

```bash

yarn add pg

```

</Tab>

<Tab value="bun">

```bash

bun add pg

```

</Tab>

</Tabs>

And then you can use the following code to connect to your database.

```
```ts title="auth.ts"
```

```
import { Pool } from "pg";
```

```
export const auth = betterAuth({
```

```
  database: new Pool({
```

```
    connectionString: process.env.DATABASE_URL
```

```
  }},
```

```
  })
  ...
</Step>
```

```
<Step>
#### Enable Email and Password (Optional)
```

Enable the email and password in your auth config and implement your own logic for sending verification emails, reset password emails, etc.

```
```ts title="auth.ts"
import { betterAuth } from "better-auth";

export const auth = betterAuth({
  database: new Pool({
    connectionString: process.env.DATABASE_URL
  }),
  emailAndPassword: { // [!code highlight]
    enabled: true, // [!code highlight]
  }, // [!code highlight]
  emailVerification: {
    sendVerificationEmail: async({ user, url })=>{
      // implement your logic here to send email verification
    }
  },
});
```
```

See [Email and Password](/docs/authentication/email-password) for more configuration options.

```
</Step>
```

```
<Step>
#### Setup Social Providers (Optional)
```

Add social providers you have enabled in your Clerk project in your auth config.

```
```ts title="auth.ts"
import { betterAuth } from "better-auth";

export const auth = betterAuth({
  database: new Pool({
    connectionString: process.env.DATABASE_URL
  }),
  emailAndPassword: {
    enabled: true,
  },
  socialProviders: { // [!code highlight]
    github: { // [!code highlight]
      clientId: process.env.GITHUB_CLIENT_ID, // [!code highlight]
      clientSecret: process.env.GITHUB_CLIENT_SECRET, // [!code highlight]
    } // [!code highlight]
  } // [!code highlight]
});
```
```

```
</Step>
```

```
<Step>
#### Add Plugins (Optional)
```

You can add the following plugins to your auth config based on your needs.

[Admin](/docs/plugins/admin) Plugin will allow you to manage users, user impersonations and app level roles and permissions.

[Two Factor](/docs/plugins/2fa) Plugin will allow you to add two-factor authentication to your application.

[Phone Number](/docs/plugins/phone-number) Plugin will allow you to add phone number authentication to your application.

[Username](/docs/plugins/username) Plugin will allow you to add username authentication to your application.

```
```ts title="auth.ts"
import { Pool } from "pg";
import { betterAuth } from "better-auth";
import { admin, twoFactor, phoneNumber, username } from "better-auth/plugins";

export const auth = betterAuth({
  database: new Pool({
    connectionString: process.env.DATABASE_URL
  }),
  emailAndPassword: {
    enabled: true,
  },
  socialProviders: {
    github: {
      clientId: process.env.GITHUB_CLIENT_ID!,
      clientSecret: process.env.GITHUB_CLIENT_SECRET!,
    }
  },
  plugins: [admin(), twoFactor(), phoneNumber(), username()], // [!code highlight]
});
```
```

</Step>

<Step>

#### Generate Schema

If you're using a custom database adapter, generate the schema:

```
```sh
npx @better-auth/cli generate
```
```

or if you're using the default adapter, you can use the following command:

```
```sh
npx @better-auth/cli migrate
```
```

</Step>

<Step>

#### Export Clerk Users

Go to the Clerk dashboard and export the users. Check how to do it [here]

(<https://clerk.com/docs/deployments/exporting-users#export-your-users-data-from-the-clerk-dashboard>). It will download a CSV file with the users data. You need to save it as `exported\_users.csv` and put it in the root of your project.

</Step>

<Step>

#### Create the migration script

Create a new file called `migrate-clerk.ts` in the `scripts` folder and add the following code:

```
```ts title="scripts/migrate-clerk.ts"
import { generateRandomString, symmetricEncrypt } from "better-auth/crypto";

import { auth } from "@lib/auth"; // import your auth instance

function getCSVData(csv: string) {
  const lines = csv.split('\n').filter(line => line.trim());
  const headers = lines[0]?.split(',').map(header => header.trim()) || [];
}
```

```
const jsonData = lines.slice(1).map(line => {
  const values = line.split(',').map(value => value.trim());
  return headers.reduce((obj, header, index) => {
    obj[header] = values[index] || '';
    return obj;
  }, {} as Record<string, string>);
});
```

```
return jsonData as Array<{
  id: string;
  first_name: string;
  last_name: string;
  username: string;
  primary_email_address: string;
  primary_phone_number: string;
  verified_email_addresses: string;
  unverified_email_addresses: string;
  verified_phone_numbers: string;
  unverified_phone_numbers: string;
  totp_secret: string;
  password_digest: string;
  password_hasher: string;
}>;
}
```

```
const exportedUserCSV = await Bun.file("exported_users.csv").text(); // this is the file you downloaded from Clerk
```

```
async function getClerkUsers(totalUsers: number) {
  const clerkUsers: {
    id: string;
    first_name: string;
    last_name: string;
    username: string;
    image_url: string;
    password_enabled: boolean;
    two_factor_enabled: boolean;
    totp_enabled: boolean;
    backup_code_enabled: boolean;
    banned: boolean;
    locked: boolean;
    lockout_expires_in_seconds: number;
    created_at: number;
    updated_at: number;
    external_accounts: {
      id: string;
      provider: string;
      identification_id: string;
      provider_user_id: string;
      approved_scopes: string;
      email_address: string;
      first_name: string;
      last_name: string;
      image_url: string;
      created_at: number;
      updated_at: number;
    }[]
  }[] = [];
  for (let i = 0; i < totalUsers; i += 500) {
    const response = await fetch(`https://api.clerk.com/v1/users?offset=${i}&limit=${500}`, {
      headers: {
        'Authorization': `Bearer ${process.env.CLERK_SECRET_KEY}`
      }
    });
    if (!response.ok) {
      throw new Error(`Failed to fetch users: ${response.statusText}`);
    }
  }
}
```

```

    }
    const clerkUsersData = await response.json();
    // biome-ignore lint/suspicious/noExplicitAny: <explanation>
    clerkUsers.push(...clerkUsersData as any);
  }
  return clerkUsers;
}

export async function generateBackupCodes(
  secret: string,
){
  const key = secret;
  const backupCodes = Array.from({ length: 10 })
    .fill(null)
    .map(() => generateRandomString(10, "a-z", "0-9", "A-Z"))
    .map((code) => `${code.slice(0, 5)}-${code.slice(5)}`);
  const encCodes = await symmetricEncrypt({
    data: JSON.stringify(backupCodes),
    key: key,
  });
  return encCodes
}

// Helper function to safely convert timestamp to Date
function safeDateConversion(timestamp?: number): Date {
  if (!timestamp) return new Date();

  // Convert seconds to milliseconds
  const date = new Date(timestamp * 1000);

  // Check if the date is valid
  if (isNaN(date.getTime())) {
    console.warn(` Invalid timestamp: ${timestamp}, falling back to current date` );
    return new Date();
  }

  // Check for unreasonable dates (before 2000 or after 2100)
  const year = date.getFullYear();
  if (year < 2000 || year > 2100) {
    console.warn(` Suspicious date year: ${year}, falling back to current date` );
    return new Date();
  }

  return date;
}

async function migrateFromClerk() {
  const jsonData = getCSVData(exportedUserCSV);
  const clerkUsers = await getClerkUsers(jsonData.length);
  const ctx = await auth.$context
  const isAdminEnabled = ctx.options?.plugins?.find(plugin => plugin.id === "admin");
  const isTwoFactorEnabled = ctx.options?.plugins?.find(plugin => plugin.id === "two-factor");
  const isUsernameEnabled = ctx.options?.plugins?.find(plugin => plugin.id === "username");
  const isPhoneNumberEnabled = ctx.options?.plugins?.find(plugin => plugin.id === "phone-number");
  for (const user of jsonData) {
    const { id, first_name, last_name, username, primary_email_address, primary_phone_number,
    verified_email_addresses, unverified_email_addresses, verified_phone_numbers, unverified_phone_numbers, totp_secret,
    password_digest, password_hasher } = user;
    const clerkUser = clerkUsers.find(clerkUser => clerkUser?.id === id);

    // create user
    const createdUser = await ctx.adapter.create<{
      id: string;
    }>({

```

```

model: "user",
data: {
  id,
  email: primary_email_address,
  emailVerified: verified_email_addresses.length > 0,
  name: `${first_name} ${last_name}`,
  image: clerkUser?.image_url,
  createdAt: safeDateConversion(clerkUser?.created_at),
  updatedAt: safeDateConversion(clerkUser?.updated_at),
  // # Two Factor (if you enabled two factor plugin)
  ...(isTwoFactorEnabled ? {
    twoFactorEnabled: clerkUser?.two_factor_enabled
  } : {}),
  // # Admin (if you enabled admin plugin)
  ...(isAdminEnabled ? {
    banned: clerkUser?.banned,
    banExpiresAt: clerkUser?.lockout_expires_in_seconds,
    role: "user"
  } : {}),
  // # Username (if you enabled username plugin)
  ...(isUsernameEnabled ? {
    username: username,
  } : {}),
  // # Phone Number (if you enabled phone number plugin)
  ...(isPhoneNumberEnabled ? {
    phoneNumber: primary_phone_number,
    phoneNumberVerified: verified_phone_numbers.length > 0,
  } : {}),
},
forceAllowId: true
}).catch(async e => {
  return await ctx.adapter.findOne<{
    id: string;
  }>({
    model: "user",
    where: [{
      field: "id",
      value: id
    }]
  })
})
// create external account
const externalAccounts = clerkUser?.external_accounts;
if (externalAccounts) {
  for (const externalAccount of externalAccounts) {
    const { id, provider, identification_id, provider_user_id, approved_scopes, email_address, first_name, last_name,
image_url, created_at, updated_at } = externalAccount;
    if (externalAccount.provider === "credential") {
      await ctx.adapter.create({
        model: "account",
        data: {
          id,
          providerId: provider,
          accountId: externalAccount.provider_user_id,
          scope: approved_scopes,
          userId: createdUser?.id,
          createdAt: safeDateConversion(created_at),
          updatedAt: safeDateConversion(updated_at),
          password: password_digest,
        }
      })
    }
  }
} else {
  await ctx.adapter.create({
    model: "account",
    data: {

```

```

      id,
      providerId: provider.replace("oauth_", ""),
      accountId: externalAccount.provider_user_id,
      scope: approved_scopes,
      userId: createdUser?.id,
      createdAt: safeDateConversion(created_at),
      updatedAt: safeDateConversion(updated_at),
    },
    forceAllowId: true
  })
}
}
}

//two factor
if (isTwoFactorEnabled) {
  await ctx.adapter.create({
    model: "twoFactor",
    data: {
      userId: createdUser?.id,
      secret: totp_secret,
      backupCodes: await generateBackupCodes(totp_secret)
    }
  })
}
}
}
}

migrateFromClerk()
  .then(() => {
    console.log(' Migration completed');
    process.exit(0);
  })
  .catch((error) => {
    console.error(' Migration failed:', error);
    process.exit(1);
  });
````

```

Make sure to replace the `process.env.CLERK\_SECRET\_KEY` with your own Clerk secret key. Feel free to customize the script to your needs.

</Step>

<Step>

#### Run the migration

Run the migration:

```

```sh
bun run script/migrate-clerk.ts # you can use any thing you like to run the script
```

```

<Callout type="warning">

Make sure to:

1. Test the migration in a development environment first
2. Monitor the migration process for any errors
3. Verify the migrated data in Better Auth before proceeding
4. Keep Clerk installed and configured until the migration is complete

</Callout>

</Step>

<Step>

#### Verify the migration

After running the migration, verify that all users have been properly migrated by checking the database.

<Step>

#### Update your components

Now that the data is migrated, you can start updating your components to use Better Auth. Here's an example for the sign-in component:

```
```tsx title="components/auth/sign-in.tsx"
import { authClient } from "better-auth/client";

export const SignIn = () => {
  const handleSignIn = async () => {
    const { data, error } = await authClient.signIn.email({
      email: "user@example.com",
      password: "password",
    });

    if (error) {
      console.error(error);
      return;
    }
    // Handle successful sign in
  };

  return (
    <form onSubmit={handleSignIn}>
      <button type="submit">Sign in</button>
    </form>
  );
};
```
```

</Step>

<Step>

#### Update the middleware

Replace your Clerk middleware with Better Auth's middleware:

```
```ts title="middleware.ts"

import { NextRequest, NextResponse } from "next/server";
import { getSessionCookie } from "better-auth/cookies";
export async function middleware(request: NextRequest) {
  const sessionCookie = getSessionCookie(request);
  const { pathname } = request.nextUrl;
  if (sessionCookie && ["/login", "/signup"].includes(pathname)) {
    return NextResponse.redirect(new URL("/dashboard", request.url));
  }
  if (!sessionCookie && pathname.startsWith("/dashboard")) {
    return NextResponse.redirect(new URL("/login", request.url));
  }
  return NextResponse.next();
}
```

```
export const config = {
  matcher: ["/dashboard", "/login", "/signup"],
};
```
```

</Step>

<Step>

#### Remove Clerk Dependencies



Once you've verified that everything is working correctly with Better Auth, you can remove Clerk:

```
```bash title="Remove Clerk"
npm remove @clerk/nextjs @clerk/themes @clerk/types
```
```

</Step>

</Steps>

## ## Additional Resources

[Goodbye Clerk, Hello Better Auth - Full Migration Guide!](https://www.youtube.com/watch?v=Za\_QihbDSuk)

## ## Wrapping Up

Congratulations! You've successfully migrated from Clerk to Better Auth.

Better Auth offers greater flexibility and more features—be sure to explore the [documentation](/docs) to unlock its full potential.

```
file: ./content/docs/guides/create-a-db-adapter.mdx
meta: {
  "title": "Create a Database Adapter",
  "description": "Learn how to create a custom database adapter for Better-Auth"
}
```

Learn how to create a custom database adapter for Better-Auth using `createAdapter`.

Our `createAdapter` function is designed to be very flexible, and we've done our best to make it easy to understand and use.

Our hope is to allow you to focus on writing database logic, and not have to worry about how the adapter is working with Better-Auth.

Anything from custom schema configurations, custom ID generation, safe JSON parsing, and more is handled by the `createAdapter` function.

All you need to do is provide the database logic, and the `createAdapter` function will handle the rest.

## ## Quick Start

<Steps>

<Step>

#### Get things ready

1. Import `createAdapter`.
2. Create `CustomAdapterConfig` interface that represents your adapter config options.
3. Create the adapter!

```
```ts
import { createAdapter, type AdapterDebugLogs } from "better-auth/adapters";
```

```
// Your custom adapter config options
```

```
interface CustomAdapterConfig {
```

```
  /**
```

```
   * Helps you debug issues with the adapter.
```

```
  */
```

```
  debugLogs?: AdapterDebugLogs;
```

```
  /**
```

```
   * If the table names in the schema are plural.
```

```
  */
```

```
  usePlural?: boolean;
```

```
}
```

```
export const myAdapter = (config: CustomAdapterConfig = {}) =>
  createAdapter({
    // ...
```

```
});
...
</Step>
```

```
<Step>
#### Configure the adapter
```

The `config` object is mostly used to provide information about the adapter to Better-Auth. We try to minimize the amount of code you need to write in your adapter functions, and these `config` options are used to help us do that.

```
```ts
// ...
export const myAdapter = (config: CustomAdapterConfig = {}) =>
  createAdapter({
    config: {
      adapterId: "custom-adapter", // A unique identifier for the adapter.
      adapterName: "Custom Adapter", // The name of the adapter.
      usePlural: config.usePlural ?? false, // Whether the table names in the schema are plural.
      debugLogs: config.debugLogs ?? false, // Whether to enable debug logs.
      supportsJSON: false, // Whether the database supports JSON. (Default: false)
      supportsDates: true, // Whether the database supports dates. (Default: true)
      supportsBooleans: true, // Whether the database supports booleans. (Default: true)
      supportsNumericIds: true, // Whether the database supports auto-incrementing numeric IDs. (Default: true)
    },
    // ...
  });
...
</Step>
```

```
<Step>
#### Create the adapter
```

The `adapter` function is where you write the code that interacts with your database.

```
```ts
// ...
export const myAdapter = (config: CustomAdapterConfig = {}) =>
  createAdapter({
    config: {
      // ...
    },
    adapter: ({}) => {
      return {
        create: async ({ data, model, select }) => {
          // ...
        },
        update: async ({ data, model, select }) => {
          // ...
        },
        updateMany: async ({ data, model, select }) => {
          // ...
        },
        delete: async ({ data, model, select }) => {
          // ...
        },
        // ...
      };
    },
  });
...

```

```
<Callout>
  Learn more about the `adapter` here [here](/docs/concepts/database#adapters).
</Callout>
```

```
</Step>
</Steps>
```

## ## Adapter

The ``adapter`` function is where you write the code that interacts with your database.

If you haven't already, check out the ``options`` object in the [config section](#config), as it can be useful for your adapter.

Before we get into the adapter function, let's go over the parameters that are available to you.

- \* ``options``: The Better Auth options.
- \* ``schema``: The schema from the user's Better Auth instance.
- \* ``debugLog``: The debug log function.
- \* ``getField``: The get field function.
- \* ``getDefaultModelName``: The get default model name function.
- \* ``getDefaultFieldName``: The get default field name function.
- \* ``getFieldAttributes``: The get field attributes function.

```
```ts title="Example"
```

```
adapter: ({
  options,
  schema,
  debugLog,
  getField,
  getDefaultModelName,
  getDefaultFieldName,
}) => {
  return {
    // ...
  };
};
```
```

## ### Adapter Methods

\* All ``model`` values are already transformed into the correct model name for the database based on the end-user's schema configuration.

\* This also means that if you need access to the ``schema`` version of a given model, you can't use this exact ``model`` value, you'll need to use the ``getDefaultModelName`` function provided in the options to convert the ``model`` to the ``schema`` version.

\* We will automatically fill in any missing fields you return based on the user's ``schema`` configuration.

\* Any method that includes a ``select`` parameter, is only for the purpose of getting data from your database more efficiently. You do not need to worry about only returning what the ``select`` parameter states, as we will handle that for you.

## ### ``create`` method

The ``create`` method is used to create a new record in the database.

### <Callout>

Note:

If the user has enabled the ``useNumberId`` option, or if ``generateId`` is ``false`` in the user's Better Auth config, then it's expected that the ``id`` is provided in the ``data`` object. Otherwise, the ``id`` will be automatically generated.

Additionally, it's possible to pass ``forceAllowId`` as a parameter to the ``create`` method, which allows ``id`` to be provided in the ``data`` object.

We handle ``forceAllowId`` internally, so you don't need to worry about it.

</Callout>

parameters:

- \* ``model``: The model/table name that new data will be inserted into.
- \* ``data``: The data to insert into the database.
- \* ``select``: An array of fields to return from the database.

<Callout>

Make sure to return the data that is inserted into the database.

</Callout>

```
```ts title="Example"
```

```
create: async ({ model, data, select }) => {
  // Example of inserting data into the database.
  return await db.insert(model).values(data);
};
```
```

### ### `update` method

The `update` method is used to update a record in the database.

parameters:

- \* `model`: The model/table name that the record will be updated in.
- \* `where`: The `where` clause to update the record by.
- \* `update`: The data to update the record with.

<Callout>

Make sure to return the data in the row which is updated. This includes any fields that were not updated.

</Callout>

```
```ts title="Example"
```

```
update: async ({ model, where, update }) => {
  // Example of updating data in the database.
  return await db.update(model).set(update).where(where);
};
```
```

### ### `updateMany` method

The `updateMany` method is used to update multiple records in the database.

parameters:

- \* `model`: The model/table name that the records will be updated in.
- \* `where`: The `where` clause to update the records by.
- \* `update`: The data to update the records with.

<Callout>Make sure to return the number of records that were updated.</Callout>

```
```ts title="Example"
```

```
updateMany: async ({ model, where, update }) => {
  // Example of updating multiple records in the database.
  return await db.update(model).set(update).where(where);
};
```
```

### ### `delete` method

The `delete` method is used to delete a record from the database.

parameters:

- \* `model`: The model/table name that the record will be deleted from.
- \* `where`: The `where` clause to delete the record by.

```
```ts title="Example"
```

```
delete: async ({ model, where }) => {
  // Example of deleting a record from the database.
  await db.delete(model).where(where);
};
```

```
}
```,
```

### #### `deleteMany` method

The `deleteMany` method is used to delete multiple records from the database.

parameters:

\* `model`: The model/table name that the records will be deleted from.

\* `where`: The `where` clause to delete the records by.

<Callout>Make sure to return the number of records that were deleted.</Callout>

```
```ts title="Example"
deleteMany: async ({ model, where }) => {
  // Example of deleting multiple records from the database.
  return await db.delete(model).where(where);
};
```,
```

### #### `findOne` method

The `findOne` method is used to find a single record in the database.

parameters:

\* `model`: The model/table name that the record will be found in.

\* `where`: The `where` clause to find the record by.

\* `select`: The `select` clause to return.

<Callout>Make sure to return the data that is found in the database.</Callout>

```
```ts title="Example"
findOne: async ({ model, where, select }) => {
  // Example of finding a single record in the database.
  return await db.select().from(model).where(where).limit(1);
};
```,
```

### #### `findMany` method

The `findMany` method is used to find multiple records in the database.

parameters:

\* `model`: The model/table name that the records will be found in.

\* `where`: The `where` clause to find the records by.

\* `limit`: The limit of records to return.

\* `sortBy`: The `sortBy` clause to sort the records by.

\* `offset`: The offset of records to return.

<Callout>

Make sure to return the array of data that is found in the database.

</Callout>

```
```ts title="Example"
findMany: async ({ model, where, limit, sortBy, offset }) => {
  // Example of finding multiple records in the database.
  return await db
    .select()
    .from(model)
    .where(where)
    .limit(limit)
    .offset(offset)
```

```
.orderBy(sortBy);
};
```

```

### #### `count` method

The `count` method is used to count the number of records in the database.

parameters:

- \* `model`: The model/table name that the records will be counted in.
- \* `where`: The `where` clause to count the records by.

<Callout>Make sure to return the number of records that were counted.</Callout>

```
```ts title="Example"
count: async ({ model, where }) => {
  // Example of counting the number of records in the database.
  return await db.select().from(model).where(where).count();
};
```

```

### #### `options` (optional)

The `options` object is for any potential config that you got from your custom adapter options.

```
```ts title="Example"
const myAdapter = (config: CustomAdapterConfig) => {
  createAdapter({
    config: {
      // ...
    },
    adapter: ({ options }) => {
      return {
        options: config,
      };
    },
  });
};
```

```

### #### `createSchema` (optional)

The `createSchema` method allows the [Better Auth CLI](/docs/concepts/cli) to [generate](/docs/concepts/cli/#generate) a schema for the database.

parameters:

- \* `tables`: The tables from the user's Better-Auth instance schema; which is expected to be generated into the schema file.
- \* `file`: The file the user may have passed in to the `generate` command as the expected schema file output path.

```
```ts title="Example"
createSchema: async ({ file, tables }) => {
  // ... Custom logic to create a schema for the database.
};
```

```

## ## Test your adapter

We've provided a test suite that you can use to test your adapter. It requires you to use `vitest`.

```
```ts title="my-adapter.test.ts"
import { expect, test, describe } from "vitest";
import { runAdapterTest } from "better-auth/adapters/test";
import { myAdapter } from "../my-adapter";

```

```
describe("My Adapter Tests", async () => {
  afterAll(async () => {
    // Run DB cleanup here...
  });
  const adapter = myAdapter({
    debugLogs: {
      // If your adapter config allows passing in debug logs, then pass this here.
      isRunningAdapterTests: true, // This is our super secret flag to let us know to only log debug logs if a test fails.
    },
  });

  await runAdapterTest({
    getAdapter: async (betterAuthOptions = {}) => {
      return adapter(betterAuthOptions);
    },
  });
});
````
```

### ### Numeric ID tests

If your database supports numeric IDs, then you should run this test as well:

```
````ts title="my-adapter.number-id.test.ts"
import { expect, test, describe } from "vitest";
import { runNumberIdAdapterTest } from "better-auth/adapters/test";
import { myAdapter } from "../my-adapter";

describe("My Adapter Numeric ID Tests", async () => {
  afterAll(async () => {
    // Run DB cleanup here...
  });
  const adapter = myAdapter({
    debugLogs: {
      // If your adapter config allows passing in debug logs, then pass this here.
      isRunningAdapterTests: true, // This is our super secret flag to let us know to only log debug logs if a test fails.
    },
  });

  await runNumberIdAdapterTest({
    getAdapter: async (betterAuthOptions = {}) => {
      return adapter(betterAuthOptions);
    },
  });
});
````
```

### ## Config

The `config` object is used to provide information about the adapter to Better-Auth.

We **highly recommend** going through and reading each provided option below, as it will help you understand how to properly configure your adapter.

#### ### Required Config

##### ### `adapterId`

A unique identifier for the adapter.

##### ### `adapterName`

The name of the adapter.

### #### Optional Config

#### #### `supportsNumericIds`

Whether the database supports numeric IDs. If this is set to `false` and the user's config has enabled `useNumberId`, then we will throw an error.

#### #### `supportsJSON`

Whether the database supports JSON. If the database doesn't support JSON, we will use a `string` to save the JSON data. And when we retrieve the data, we will safely parse the `string` back into a JSON object.

#### #### `supportsDates`

Whether the database supports dates. If the database doesn't support dates, we will use a `string` to save the date. (ISO string) When we retrieve the data, we will safely parse the `string` back into a `Date` object.

#### #### `supportsBooleans`

Whether the database supports booleans. If the database doesn't support booleans, we will use a `0` or `1` to save the boolean value. When we retrieve the data, we will safely parse the `0` or `1` back into a boolean value.

#### #### `usePlural`

Whether the table names in the schema are plural. This is often defined by the user, and passed down through your custom adapter options. If you do not intend to allow the user to customize the table names, you can ignore this option, or set this to `false`.

```
```ts title="Example"
const adapter = myAdapter({
  // This value then gets passed into the `usePlural`
  // option in the createAdapter `config` object.
  usePlural: true,
});
```
```

#### #### `debugLogs`

Used to enable debug logs for the adapter. You can pass in a boolean, or an object with the following keys: `create`, `update`, `updateMany`, `findOne`, `findMany`, `delete`, `deleteMany`, `count`. If any of the keys are `true`, the debug logs will be enabled for that method.

```
```ts title="Example"
// Will log debug logs for all methods.
const adapter = myAdapter({
  debugLogs: true,
});
```
```

```
```ts title="Example"
// Will only log debug logs for the `create` and `update` methods.
const adapter = myAdapter({
  debugLogs: {
    create: true,
    update: true,
  },
});
```
```

#### #### `disableIdGeneration`

Whether to disable ID generation. If this is set to `true`, then the user's `generateId` option will be ignored.

#### #### `customIdGenerator`



If your database only supports a specific custom ID generation, then you can use this option to generate your own IDs.

#### #### `mapKeysTransformInput`

If your database uses a different key name for a given situation, you can use this option to map the keys. This is useful for databases that expect a different key name for a given situation. For example, MongoDB uses `\_id` while in Better-Auth we use `id`.

Each key in the returned object represents the old key to replace. The value represents the new key.

This can be a partial object that only transforms some keys.

```
```ts title="Example"
mapKeysTransformInput: () => {
  return {
    id: "_id", // We want to replace `id` to `_id` to save into MongoDB
  };
},
```
```

#### #### `mapKeysTransformOutput`

If your database uses a different key name for a given situation, you can use this option to map the keys. This is useful for databases that use a different key name for a given situation. For example, MongoDB uses `\_id` while in Better-Auth we use `id`.

Each key in the returned object represents the old key to replace. The value represents the new key.

This can be a partial object that only transforms some keys.

```
```ts title="Example"
mapKeysTransformOutput: () => {
  return {
    _id: "id", // We want to replace `_id` (from MongoDB) to `id` (for Better-Auth)
  };
},
```
```

#### #### `customTransformInput`

If you need to transform the input data before it is saved to the database, you can use this option to transform the data.

<Callout type="warn">

If you're using `supportsJSON`, `supportsDates`, or `supportsBooleans`, then the transformations will be applied before your `customTransformInput` function is called.

</Callout>

The `customTransformInput` function receives the following arguments:

- \* `data`: The data to transform.
- \* `field`: The field that is being transformed.
- \* `fieldAttributes`: The field attributes of the field that is being transformed.
- \* `select`: The `select` values which the query expects to return.
- \* `model`: The model that is being transformed.
- \* `schema`: The schema that is being transformed.
- \* `options`: Better Auth options.

The `customTransformInput` function runs at every key in the data object of a given action.

```
```ts title="Example"
customTransformInput: ({ field, data }) => {
  if (field === "id") {
```

```

    return "123"; // Force the ID to be "123"
  }

  return data;
};
```

```

### #### `customTransformOutput`

If you need to transform the output data before it is returned to the user, you can use this option to transform the data. The `customTransformOutput` function is used to transform the output data. Similar to the `customTransformInput` function, it runs at every key in the data object of a given action, but it runs after the data is retrieved from the database.

```

```ts title="Example"
customTransformOutput: ({ field, data }) => {
  if (field === "name") {
    return "Bob"; // Force the name to be "Bob"
  }

  return data;
};
```

```

```

```ts
const some_data = await adapter.create({
  model: "user",
  data: {
    name: "John",
  },
});
```

```

```

// The name will be "Bob"
console.log(some_data.name);
```

```

```

file: ./content/docs/guides/next-auth-migration-guide.mdx
meta: {
  "title": "Migrating from NextAuth.js to Better Auth",
  "description": "A step-by-step guide to transitioning from NextAuth.js to Better Auth."
}

```

In this guide, we'll walk through the steps to migrate a project from [NextAuth.js](https://authjs.dev/) to Better Auth, ensuring no loss of data or functionality. While this guide focuses on Next.js, it can be adapted for other frameworks as well.

\*\*\*

## ## Before You Begin

Before starting the migration process, set up Better Auth in your project. Follow the [installation guide](/docs/installation) to get started.

\*\*\*

### <Steps>

#### <Step>

### #### Mapping Existing Columns

Instead of altering your existing database column names, you can map them to match Better Auth's expected structure. This allows you to retain your current database schema.

### #### User Schema

Map the following fields in the user schema:

\* (next-auth v4) `emailVerified` : datetime → boolean

#### #### Session Schema

Map the following fields in the session schema:

\* `expires` → `expiresAt`

\* `sessionToken` → `token`

\* (next-auth v4) add `createdAt` with datetime type

\* (next-auth v4) add `updatedAt` with datetime type

```
```typescript title="auth.ts"
export const auth = betterAuth({
  // Other configs
  session: {
    fields: {
      expiresAt: "expires", // Map your existing `expires` field to Better Auth's `expiresAt`
      token: "sessionToken" // Map your existing `sessionToken` field to Better Auth's `token`
    }
  },
});
```
```

Make sure to have `createdAt` and `updatedAt` fields on your session schema.

#### #### Account Schema

Map these fields in the account schema:

\* (next-auth v4) `provider` → `providerId`

\* `providerAccountId` → `accountId`

\* `refresh\_token` → `refreshToken`

\* `access\_token` → `accessToken`

\* (next-auth v3) `access\_token\_expires` → `accessTokenExpiresAt` and int → datetime

\* (next-auth v4) `expires\_at` → `accessTokenExpiresAt` and int → datetime

\* `id\_token` → `idToken`

\* (next-auth v4) add `createdAt` with datetime type

\* (next-auth v4) add `updatedAt` with datetime type

Remove the `session\_state`, `type`, and `token\_type` fields, as they are not required by Better Auth.

```
```typescript title="auth.ts"
export const auth = betterAuth({
  // Other configs
  account: {
    fields: {
      accountId: "providerAccountId",
      refreshToken: "refresh_token",
      accessToken: "access_token",
      accessTokenExpiresAt: "access_token_expires",
      idToken: "id_token",
    }
  },
});
```
```

**Note:** If you use ORM adapters, you can map these fields in your schema file.

**Example with Prisma:**

```
```prisma title="schema.prisma"
model Session {
```

```

    id      String  @id @default(cuid())
    expiresAt DateTime @map("expires") // Map your existing `expires` field to Better Auth's `expiresAt`
    token    String  @map("sessionToken") // Map your existing `sessionToken` field to Better Auth's `token`
    userId   String
    user     User    @relation(fields: [userId], references: [id])
  }
  ...

```

Make sure to have `createdAt` and `updatedAt` fields on your account schema.

</Step>

<Step>

#### Update the Route Handler

In the `app/api/auth` folder, rename the `[...nextauth]` file to `[...all]` to avoid confusion. Then, update the `route.ts` file as follows:

```

```typescript title="app/api/auth/[...all]/route.ts"
import { toNextJsHandler } from "better-auth/next-js";
import { auth } from "~/server/auth";

export const { POST, GET } = toNextJsHandler(auth);
```

```

</Step>

<Step>

#### Update the Client

Create a file named `auth-client.ts` in the `lib` folder. Add the following code:

```

```typescript title="auth-client.ts"
import { createAuthClient } from "better-auth/react";

export const authClient = createAuthClient({
  baseUrl: process.env.BASE_URL! // Optional if the API base URL matches the frontend
});

export const { signIn, signOut, useSession } = authClient;
```

```

##### Social Login Functions

Update your social login functions to use Better Auth. For example, for Discord:

```

```typescript
import { signIn } from "~/lib/auth-client";

export const signInDiscord = async () => {
  const data = await signIn.social({
    provider: "discord"
  });
  return data;
};
```

```

##### Update `useSession` Calls

Replace `useSession` calls with Better Auth's version. Example:

```

```typescript title="Profile.tsx"
import { useSession } from "~/lib/auth-client";

export const Profile = () => {
  const { data } = useSession();
  return (

```

```

    <div>
      <pre>
        {JSON.stringify(data, null, 2)}
      </pre>
    </div>
  );
};
...
</Step>

```

<Step>  
 #### Server-Side Session Handling

Use the `auth` instance to get session data on the server:

```

```typescript title="actions.ts"
"use server";

import { auth } from "~/server/auth";
import { headers } from "next/headers";

export const protectedAction = async () => {
  const session = await auth.api.getSession({
    headers: await headers(),
  });
};
...
</Step>

```

<Step>  
 #### Middleware

To protect routes with middleware, refer to the [Next.js middleware guide](/docs/integrations/next#middleware).

</Step>  
 </Steps>

## ## Wrapping Up

Congratulations! You've successfully migrated from NextAuth.js to Better Auth. For a complete implementation with multiple authentication methods, check out the [demo repository](https://github.com/Bekacru/t3-app-better-auth).

Better Auth offers greater flexibility and more features—be sure to explore the [documentation](/docs) to unlock its full potential.

```

file: ./content/docs/guides/optimizing-for-performance.mdx
meta: {
  "title": "Optimizing for Performance",
  "description": "A guide to optimizing your Better Auth application for performance."
}

```

In this guide, we'll go over some of the ways you can optimize your application for a more performant Better Auth app.

## ### Caching

Caching is a powerful technique that can significantly improve the performance of your Better Auth application by reducing the number of database queries and speeding up response times.

### #### Cookie Cache

Calling your database every time `useSession` or `getSession` invoked isn't ideal, especially if sessions don't change frequently. Cookie caching handles this by storing session data in a short-lived, signed cookie similar to how JWT access tokens are used with refresh tokens.

To turn on cookie caching, just set `session.cookieCache` in your auth config:

```

```ts title="auth.ts"
import { betterAuth } from "better-auth";

export const auth = betterAuth({
  session: {
    cookieCache: {
      enabled: true,
      maxAge: 5 * 60, // Cache duration in seconds
    },
  },
});
```

```

Read more about [cookie caching](/docs/concepts/session-management#cookie-cache).

### ### Framework Caching

Here are examples of how you can do caching in different frameworks and environments:

<Tabs items=["Next", "Remix", "SolidStart", "React Query"]>  
 <Tab value="Next">

Since Next v15, we can use the `use cache` directive to cache the response of a server function.

```

```ts
export async function getUsers() {
  'use cache' // [!code highlight]
  const { users } = await auth.api.listUsers();
  return users
}
```

```

Learn more about NextJS use cache directive <Link href="https://nextjs.org/docs/app/api-reference/directives/use-cache">here</Link>.

</Tab>

<Tab value="Remix">

In Remix, you can use the `cache` option in the `loader` function to cache responses on the server. Here's an example:

```

```ts
import { json } from '@remix-run/node';

export const loader = async () => {
  const { users } = await auth.api.listUsers();
  return json(users, {
    headers: {
      'Cache-Control': 'max-age=3600', // Cache for 1 hour
    },
  });
};
```

```

You can read a nice guide on Loader vs Route Cache Headers in Remix <Link href="https://sergiodxa.com/articles/loader-vs-route-cache-headers-in-remix">here</Link>.

</Tab>

<Tab value="SolidStart">

In SolidStart, you can use the `query` function to cache data. Here's an example:

```

```tsx
const getUsers = query(
  async () => (await auth.api.listUsers()).users,
  "getUsers"
);
```

```

Learn more about SolidStart `query` function <Link href="https://docs.solidjs.com/solid-router/reference/data-apis/query">here</Link>.

&lt;/Tab&gt;

&lt;Tab value="React Query"&gt;

With React Query you can use the `useQuery` hook to cache data. Here's an example:

```
```ts
import { useQuery } from '@tanstack/react-query';

const fetchUsers = async () => {
  const { users } = await auth.api.listUsers();
  return users;
};

export default function Users() {
  const { data: users, isLoading } = useQuery('users', fetchUsers, {
    staleTime: 1000 * 60 * 15, // Cache for 15 minutes
  });

  if (isLoading) return <div>Loading...</div>;

  return (
    <ul>
      {users.map(user => (
        <li key={user.id}>{user.name}</li>
      ))}
    </ul>
  );
}
```
```

Learn more about React Query use cache directive <Link href="https://react-query.tanstack.com/reference/useQuery#usecache">here</Link>.

&lt;/Tab&gt;

&lt;/Tabs&gt;

### ## SSR Optimizations

If you're using a framework that supports server-side rendering, it's usually best to pre-fetch user session on the server and use it as a fallback on the client.

```
```ts
const session = await auth.api.getSession({
  headers: await headers(),
});
//then pass the session to the client
```
```

### ## Database optimizations

Optimizing database performance is essential to get the best out of Better Auth.

#### #### Recommended fields to index

| Table         | Fields                     | Plugin       |
|---------------|----------------------------|--------------|
| users         | `email`                    |              |
| accounts      | `userId`                   |              |
| sessions      | `userId`, `token`          |              |
| verifications | `identifier`               |              |
| invitations   | `email`, `organizationId`  | organization |
| members       | `userId`, `organizationId` | organization |
| organizations | `slug`                     | organization |

```
| passkey | `userId` | passkey |
| twoFactor | `secret` | twoFactor |
```

<Callout>

We intend to add indexing support in our schema generation tool in the future.

</Callout>

file: ./content/docs/guides/supabase-migration-guide.mdx

meta: {

"title": "Migrating from Supabase Auth to Better Auth",

"description": "A step-by-step guide to transitioning from Supabase Auth to Better Auth."

}

In this guide, we'll walk through the steps to migrate a project from Supabase Auth to Better Auth.

<Callout type="warn">

This migration will invalidate all active sessions. While this guide doesn't currently cover migrating two-factor (2FA) or Row Level Security (RLS) configurations, both should be possible with additional steps.

</Callout>

## ## Before You Begin

Before starting the migration process, set up Better Auth in your project. Follow the [installation guide](/docs/installation) to get started.

<Steps>

<Step>

#### Connect to your database

You'll need to connect to your database to migrate the users and accounts. Copy your `DATABASE\_URL` from your Supabase project and use it to connect to your database. And for this example, we'll need to install `pg` to connect to the database.

<Tabs groupId="package-manager" persist items={}>

<Tab value="npm">

```bash

npm install pg

```

</Tab>

<Tab value="pnpm">

```bash

pnpm add pg

```

</Tab>

<Tab value="yarn">

```bash

yarn add pg

```

</Tab>

<Tab value="bun">

```bash

bun add pg

```

</Tab>

</Tabs>

And then you can use the following code to connect to your database.

```ts title="auth.ts"

import { Pool } from "pg";



```
export const auth = betterAuth({
  database: new Pool({
    connectionString: process.env.DATABASE_URL
  }),
})
```,
</Step>
```

<Step>  
**### Enable Email and Password (Optional)**

Enable the email and password in your auth config.

```
```ts title="auth.ts"
import { admin, anonymous } from "better-auth/plugins";

export const auth = betterAuth({
  database: new Pool({
    connectionString: process.env.DATABASE_URL
  }),
  emailVerification: {
    sendEmailVerification: async(user)=>{
      // send email verification email
      // implement your own logic here
    }
  },
  emailAndPassword: { // [!code highlight]
    enabled: true, // [!code highlight]
  } // [!code highlight]
})
```,
</Step>
```

<Step>  
**### Setup Social Providers (Optional)**

Add social providers you have enabled in your Supabase project in your auth config.

```
```ts title="auth.ts"
import { admin, anonymous } from "better-auth/plugins";

export const auth = betterAuth({
  database: new Pool({
    connectionString: process.env.DATABASE_URL
  }),
  emailAndPassword: {
    enabled: true,
  },
  socialProviders: { // [!code highlight]
    github: { // [!code highlight]
      clientId: process.env.GITHUB_CLIENT_ID, // [!code highlight]
      clientSecret: process.env.GITHUB_CLIENT_SECRET, // [!code highlight]
    } // [!code highlight]
  } // [!code highlight]
})
```,
</Step>
```

<Step>  
**### Add admin and anonymous plugins (Optional)**

Add the [admin](/docs/plugins/admin) and [anonymous](/docs/plugins/anonymous) plugins to your auth config.

```
```ts title="auth.ts"
import { admin, anonymous } from "better-auth/plugins";
```

```
export const auth = betterAuth({
  database: new Pool({
    connectionString: process.env.DATABASE_URL
  }),
  emailAndPassword: {
    enabled: true,
  },
  socialProviders: {
    github: {
      clientId: process.env.GITHUB_CLIENT_ID!,
      clientSecret: process.env.GITHUB_CLIENT_SECRET!,
    }
  },
  plugins: [admin(), anonymous()], // [!code highlight]
})
````
```

</Step>

<Step>

#### Run the migration

Run the migration to create the necessary tables in your database.

```
```bash title="Terminal"
npx @better-auth/cli migrate
```
```

This will create the following tables in your database:

```
* [ `user` ](/docs/concepts/database#user)
* [ `account` ](/docs/concepts/database#account)
* [ `session` ](/docs/concepts/database#session)
* [ `verification` ](/docs/concepts/database#verification)
```

These tables will be created on the `public` schema.

</Step>

<Step>

#### Copy the migration script

Now that we have the necessary tables in our database, we can run the migration script to migrate the users and accounts from Supabase to Better Auth.

Start by creating a `.ts` file in your project.

```
```bash title="Terminal"
touch migration.ts
```
```

And then copy and paste the following code into the file.

```
```ts title="migration.ts"
import { Pool } from "pg";
import { auth } from "../auth";
import { User as SupabaseUser } from "@supabase/supabase-js";

type User = SupabaseUser & {
  is_super_admin: boolean;
  raw_user_meta_data: {
    avatar_url: string;
  };
  encrypted_password: string;
  email_confirmed_at: string;
  created_at: string;
}
```

```

    updated_at: string;
    is_anonymous: boolean;
    identities: {
      provider: string;
      identity_data: {
        sub: string;
        email: string;
      };
      created_at: string;
      updated_at: string;
    };
  };
};

const migrateFromSupabase = async () => {
  const ctx = await auth.$context;
  const db = ctx.options.database as Pool;
  const users = await db
    .query(`
      SELECT
        u.*,
        COALESCE(
          json_agg(
            i.* ORDER BY i.id
          ) FILTER (WHERE i.id IS NOT NULL),
          '[]'::json
        ) as identities
      FROM auth.users u
      LEFT JOIN auth.identities i ON u.id = i.user_id
      GROUP BY u.id
    `)
    .then((res) => res.rows as User[]);
  for (const user of users) {
    if (!user.email) {
      continue;
    }
    await ctx.adapter
      .create({
        model: "user",
        data: {
          id: user.id,
          email: user.email,
          name: user.email,
          role: user.is_super_admin ? "admin" : user.role,
          emailVerified: !!user.email_confirmed_at,
          image: user.raw_user_meta_data.avatar_url,
          createdAt: new Date(user.created_at),
          updatedAt: new Date(user.updated_at),
          isAnonymous: user.is_anonymous,
        },
      })
      .catch(() => {});
    for (const identity of user.identities) {
      const existingAccounts = await ctx.internalAdapter.findAccounts(user.id);

      if (identity.provider === "email") {
        const hasCredential = existingAccounts.find(
          (account) => account.providerId === "credential",
        );
        if (!hasCredential) {
          await ctx.adapter
            .create({
              model: "account",
              data: {
                userId: user.id,
                providerId: "credential",

```

```

        accountId: user.id,
        password: user.encrypted_password,
        createdAt: new Date(user.created_at),
        updatedAt: new Date(user.updated_at),
      },
    ))
    .catch(() => {});
  }
}
const supportedProviders = Object.keys(ctx.options.socialProviders || {})
if (supportedProviders.includes(identity.provider)) {
  const hasAccount = existingAccounts.find(
    (account) => account.providerId === identity.provider,
  );
  if (!hasAccount) {
    await ctx.adapter.create({
      model: "account",
      data: {
        userId: user.id,
        providerId: identity.provider,
        accountId: identity.identity_data?.sub,
        createdAt: new Date(identity.created_at ?? user.created_at),
        updatedAt: new Date(identity.updated_at ?? user.updated_at),
      },
    });
  }
}
}
};
migrateFromSupabase();
``,`

```

</Step>

<Step>

#### Customize the migration script (Optional)

\* `name`: the migration script will use the user's email as the name. You might want to customize it if you have the user display name in your database.

\* `socialProviderList`: the migration script will use the social providers you have enabled in your auth config. You might want to customize it if you have additional social providers that you haven't enabled in your auth config.

\* `role`: remove `role` if you're not using the `admin` plugin

\* `isAnonymous`: remove `isAnonymous` if you're not using the `anonymous` plugin.

\* update other tables that reference the `users` table to use the `id` field.

</Step>

<Step>

#### Run the migration script

Run the migration script to migrate the users and accounts from Supabase to Better Auth.

```
`` `bash title="Terminal"
```

```
bun migration.ts # or use node, ts-node, etc.
```

```
`` `
```

</Step>

<Step>

#### Update your code

Update your codebase from Supabase auth calls to Better Auth API.

Here's a list of the Supabase auth API calls and their Better Auth counterparts.

\* `supabase.auth.signUp` -> `authClient.signUp.email`

\* `supabase.auth.signInWithPassword` -> `authClient.signIn.email`

```
* `supabase.auth.signInWithOAuth` -> `authClient.signIn.social`
* `supabase.auth.signInAnonymously` -> `authClient.signIn.anonymous`
* `supabase.auth.signOut` -> `authClient.signOut`
* `supabase.auth.getSession` -> `authClient.getSession` - you can also use `authClient.useSession` for reactive state
```

Learn more:

```
* [Basic Usage](/docs/basic-usage): Learn how to use the auth client to sign up, sign in, and sign out.
* [Email and Password](/docs/authentication/email-and-password): Learn how to add email and password authentication to your project.
* [Anonymous](/docs/plugins/anonymous): Learn how to add anonymous authentication to your project.
* [Admin](/docs/plugins/admin): Learn how to add admin authentication to your project.
* [Email OTP](/docs/authentication/email-otp): Learn how to add email OTP authentication to your project.
* [Hooks](/docs/concepts/hooks): Learn how to use the hooks to listen for events.
* [Next.js](/docs/integrations/next): Learn how to use the auth client in a Next.js project.
```

</Step>

</Steps>

### ### Middleware

To protect routes with middleware, refer to the [Next.js middleware guide](/docs/integrations/next#middleware) or your framework's documentation.

## ## Wrapping Up

Congratulations! You've successfully migrated from Supabase Auth to Better Auth.

Better Auth offers greater flexibility and more features—be sure to explore the [documentation](/docs) to unlock its full potential.

```
file: ./content/docs/guides/your-first-plugin.mdx
meta: {
  "title": "Create your first plugin",
  "description": "A step-by-step guide to creating your first Better Auth plugin."
}
```

In this guide, we'll walk you through the steps of creating your first Better Auth plugin.

```
<Callout type="warn">
  This guide assumes you have <Link href="/docs/installation">setup the basics</Link> of Better Auth and are ready to create your first plugin.
</Callout>
```

```
<Steps>
<Step>
  ## Plan your idea
```

Before beginning, you must know what plugin you intend to create.

```
In this guide, we'll create a **birthday plugin** to keep track of user birth dates.
</Step>
```

```
<Step>
  ## Server plugin first
```

Better Auth plugins operate as a pair: a <Link href="/docs/concepts/plugins#create-a-server-plugin">server plugin</Link> and a <Link href="/docs/concepts/plugins#create-a-client-plugin">client plugin</Link>.

The server plugin forms the foundation of your authentication system, while the client plugin provides convenient frontend APIs to interact with your server implementation.

```
<Callout>
  You can read more about server/client plugins in our <Link href="/docs/concepts/plugins#creating-a-plugin">documentation</Link>.
</Callout>
```

### #### Creating the server plugin

Go ahead and find a suitable location create an your birthday plugin folder, with an `index.ts` file within.

```
<Files>
  <Folder name="birthday-plugin" defaultOpen>
    <File name="index.ts" />
  </Folder>
</Files>
```

In the `index.ts` file, we'll export a function that represents our server plugin. This will be what we will later add to our plugin list in the `auth.ts` file.

```
` `` ts title="index.ts"
import { createAuthClient } from "better-auth/client";
import type { BetterAuthPlugin } from "better-auth";

export const birthdayPlugin = () =>
({
  id: "birthdayPlugin",
} satisfies BetterAuthPlugin);

` ``
```

Although this does nothing, you have technically just made yourself your first plugin, congratulations! 🎉

</Step>

<Step>

### #### Defining a schema

In order to save each user's birthday data, we must create a schema on top of the `user` model.

By creating a schema here, this also allows [Better Auth's CLI](/docs/concepts/cli) to generate the schemas required to update your database.

```
<Callout type="info">
  You can learn more about plugin schemas here.
</Callout>
```

```
` `` ts title="index.ts"
//...
export const birthdayPlugin = () =>
({
  id: "birthdayPlugin",
  schema: { // [!code highlight]
    user: { // [!code highlight]
      fields: { // [!code highlight]
        birthday: { // [!code highlight]
          type: "date", // string, number, boolean, date // [!code highlight]
          required: true, // if the field should be required on a new record. (default: false) // [!code highlight]
          unique: false, // if the field should be unique. (default: false) // [!code highlight]
          references: null // if the field is a reference to another table. (default: null) // [!code highlight]
        }, // [!code highlight]
      }, // [!code highlight]
    }, // [!code highlight]
  },
} satisfies BetterAuthPlugin);

` ``
```

</Step>

<Step>

### #### Authorization logic

For this example guide, we'll set up authentication logic to check and ensure that the user who signs-up is older than 5.

But the same concept could be applied for something like verifying users agreeing to the TOS or anything alike.

To do this, we'll utilize `<Link href="/docs/concepts/plugins#hooks">Hooks</Link>`, which allows us to run code ``before`` or ``after`` an action is performed.

```
`` ts title="index.ts"
export const birthdayPlugin = () => ({
  //...
  // In our case, we want to write authorization logic,
  // meaning we want to intercept it `before` hand.
  hooks: {
    before: [
      {
        matcher: (context) => /* ... */,
        handler: createAuthMiddleware(async (ctx) => {
          //...
        }),
      },
    ],
  },
}) satisfies BetterAuthPlugin)
````
```

In our case we want to match any requests going to the signup path:

```
`` ts title="Before hook"
{
  matcher: (context) => context.path.startsWith("/sign-up/email"),
  //...
}
````
```

And for our logic, we'll write the following code to check the if user's birthday makes them above 5 years old.

```
`` ts title="Imports"
import { APIError } from "better-auth/api";
import { createAuthMiddleware } from "better-auth/plugins";
````

`` ts title="Before hook"
{
  //...
  handler: createAuthMiddleware(async (ctx) => {
    const { birthday } = ctx.body;
    if(!birthday instanceof Date) {
      throw new APIError("BAD_REQUEST", { message: "Birthday must be of type Date." });
    }

    const today = new Date();
    const fiveYearsAgo = new Date(today.setFullYear(today.getFullYear() - 5));

    if(birthday >= fiveYearsAgo) {
      throw new APIError("BAD_REQUEST", { message: "User must be above 5 years old." });
    }

    return { context: ctx };
  }),
}
````
```

**\*\*Authorized!\*\*** 

We've now successfully written code to ensure authorization for users above 5!  
`</Step>`

<Step>  
 ### Client Plugin

We're close to the finish line! 🏁

Now that we have created our server plugin, the next step is to develop our client plugin. Since there isn't much frontend APIs going on for this plugin, there isn't much to do!

First, let's create our `client.ts` file first:

```
<Files>
  <Folder name="birthday-plugin" defaultOpen>
    <File name="index.ts" />

    <File name="client.ts" />
  </Folder>
</Files>
```

Then, add the following code:

```
```ts title="client.ts"
import { BetterAuthClientPlugin } from "better-auth";
import type { birthdayPlugin } from "../index"; // make sure to import the server plugin as a type // [!code highlight]

type BirthdayPlugin = typeof birthdayPlugin;

export const birthdayClientPlugin = () => {
  return {
    id: "birthdayPlugin",
    $InferServerPlugin: {} as ReturnType<BirthdayPlugin>,
  } satisfies BetterAuthClientPlugin;
};
```
```

What we've done is allow the client plugin to infer the types defined by our schema from the server plugin.

And that's it! This is all it takes for the birthday client plugin. 🍰

</Step>

<Step>  
 ### Initiate your plugin!

Both the `client` and `server` plugins are now ready, the last step is to import them to both your `auth-client.ts` and your `server.ts` files respectively to initiate the plugin.

### Server initiation

```
```ts title="server.ts"
import { betterAuth } from "better-auth";
import { birthdayPlugin } from "../birthday-plugin"; // [!code highlight]

export const auth = betterAuth({
  plugins: [
    birthdayPlugin(), // [!code highlight]
  ]
});
```
```

### Client initiation

```
```ts title="auth-client.ts"
import { createAuthClient } from "better-auth/client";
import { birthdayClientPlugin } from "../birthday-plugin/client"; // [!code highlight]

const authClient = createAuthClient({
```



```

    plugins: [
      birthdayClientPlugin()// [!code highlight]
    ]
  });
  ...

```

#### Oh yeah, the schemas!

Don't forget to add your `birthday` field to your `user` table model!

Or, use the `generate` [CLI command](/docs/concepts/cli#generate):

```

```bash
npx @better-auth/cli@latest generate
```

```

</Step>  
</Steps>

## ## Wrapping Up

Congratulations! You've successfully created your first ever Better Auth plugin. We highly recommend you visit our [plugins documentation](/docs/concepts/plugins) to learn more information.

If you have a plugin you'd like to share with the community, feel free to let us know through our [Discord server](https://discord.gg/better-auth), or through a [pull-request](https://github.com/better-auth/better-auth/pulls) and we may add it to the [community-plugins](/docs/plugins/community-plugins) list!

```

file: ./content/docs/integrations/astro.mdx
meta: {
  "title": "Astro Integration",
  "description": "Integrate Better Auth with Astro."
}

```

Better Auth comes with first class support for Astro. This guide will show you how to integrate Better Auth with Astro.

Before you start, make sure you have a Better Auth instance configured. If you haven't done that yet, check out the [\[installation\]](/docs/installation).

## #### Mount the handler

To enable Better Auth to handle requests, we need to mount the handler to a catch all API route. Create a file inside `/pages/api/auth` called `[...all].ts` and add the following code:

```

```ts title="pages/api/auth/[...all].ts"
import { auth } from "~/auth";
import type { APIRoute } from "astro";

export const ALL: APIRoute = async (ctx) => {
  // If you want to use rate limiting, make sure to set the 'x-forwarded-for' header to the request headers from the context
  // ctx.request.headers.set("x-forwarded-for", ctx.clientAddress);
  return auth.handler(ctx.request);
};
```

```

### <Callout>

You can change the path on your better-auth configuration but it's recommended to keep it as `/api/auth/[...all]`

### </Callout>

## ## Create a client

Astro supports multiple frontend frameworks, so you can easily import your client based on the framework you're using.

If you're not using a frontend framework, you can still import the vanilla client.

```
<Tabs
  items={[ "vanilla", "react", "vue", "svelte", "solid",
  ]}
  defaultValue="react"
>
  <Tab value="vanilla">
    `` ts title="lib/auth-client.ts"
    import { createAuthClient } from "better-auth/client"
    export const authClient = createAuthClient()
    ``
  </Tab>

  <Tab value="react" title="lib/auth-client.ts">
    `` ts title="lib/auth-client.ts"
    import { createAuthClient } from "better-auth/react"
    export const authClient = createAuthClient()
    ``
  </Tab>

  <Tab value="vue" title="lib/auth-client.ts">
    `` ts title="lib/auth-client.ts"
    import { createAuthClient } from "better-auth/vue"
    export const authClient = createAuthClient()
    ``
  </Tab>

  <Tab value="svelte" title="lib/auth-client.ts">
    `` ts title="lib/auth-client.ts"
    import { createAuthClient } from "better-auth/svelte"
    export const authClient = createAuthClient()
    ``
  </Tab>

  <Tab value="solid" title="lib/auth-client.ts">
    `` ts title="lib/auth-client.ts"
    import { createAuthClient } from "better-auth/solid"
    export const authClient = createAuthClient()
    ``
  </Tab>
</Tabs>
```

### ## Auth Middleware

#### ### Astro Locals types

To have types for your Astro locals, you need to set it inside the `env.d.ts` file.

```
`` ts title="env.d.ts"

/// <reference path="../../astro/types.d.ts" />

declare namespace App {
  // Note: 'import {} from ""' syntax does not work in .d.ts files.
  interface Locals {
    user: import("better-auth").User | null;
    session: import("better-auth").Session | null;
  }
}
```

#### ### Middleware

To protect your routes, you can check if the user is authenticated using the `getSession` method in middleware and set the user and session data using the Astro locals with the types we set before. Start by creating a `middleware.ts` file in the root of your project and follow the example below:

```
```ts title="middleware.ts"
import { auth } from "@auth";
import { defineMiddleware } from "astro:middleware";

export const onRequest = defineMiddleware(async (context, next) => {
  const isAuthenticated = await auth.api
    .getSession({
      headers: context.request.headers,
    })

  if (isAuthenticated) {
    context.locals.user = isAuthenticated.user;
    context.locals.session = isAuthenticated.session;
  } else {
    context.locals.user = null;
    context.locals.session = null;
  }

  return next();
});
```
```

### ### Getting session on the server inside `.astro` file

You can use `Astro.locals` to check if the user has session and get the user data from the server side. Here is an example of how you can get the session inside an `.astro` file:

```
```astro
---
import { UserCard } from "@components/user-card";

const session = () => {
  if (Astro.locals.session) {
    return Astro.locals.session;
  } else {
    // Redirect to login page if the user is not authenticated
    return Astro.redirect("/login");
  }
}

---

<UserCard initialSession={session} />
```
```

```
file: ./content/docs/integrations/elysia.mdx
meta: {
  "title": "Elysia Integration",
  "description": "Integrate Better Auth with Elysia."
}
```

This integration guide is assuming you are using Elysia with bun server.

Before you start, make sure you have a Better Auth instance configured. If you haven't done that yet, check out the [installation](/docs/installation).

### ### Mount the handler

We need to mount the handler to Elysia endpoint.

```

` `` ts
import { Elysia } from "elysia";
import { auth } from "../auth";

const app = new Elysia().mount(auth.handler).listen(3000);

console.log(
  ` 🍪 Elysia is running at ${app.server?.hostname}:${app.server?.port}`,
);
` ``

```

### ### CORS

To configure cors, you can use the `cors` plugin from `@elysiajs/cors`.

```

` `` ts
import { Elysia } from "elysia";
import { cors } from "@elysiajs/cors";

import { auth } from "../auth";

const app = new Elysia()
  .use(
    cors({
      origin: "http://localhost:3001",
      methods: ["GET", "POST", "PUT", "DELETE", "OPTIONS"],
      credentials: true,
      allowedHeaders: ["Content-Type", "Authorization"],
    }),
  )
  .mount(auth.handler)
  .listen(3000);

console.log(
  ` 🍪 Elysia is running at ${app.server?.hostname}:${app.server?.port}`,
);
` ``

```

### ### Macro

You can use [macro](https://elysiajs.com/patterns/macro.html#macro) with [resolve](https://elysiajs.com/essential/handler.html#resolve) to provide session and user information before pass to view.

```

` `` ts
import { Elysia } from "elysia";
import { auth } from "../auth";

// user middleware (compute user and session and pass to routes)
const betterAuth = new Elysia({ name: "better-auth" })
  .mount(auth.handler)
  .macro({
    auth: {
      async resolve({ status, request: { headers } }) {
        const session = await auth.api.getSession({
          headers,
        });

        if (!session) return status(401);

        return {
          user: session.user,
          session: session.session,
        };
      },
    },
  },
);

```

```
});

const app = new Elysia()
  .use(betterAuth)
  .get("/user", ({ user }) => user, {
    auth: true,
  })
  .listen(3000);

console.log(
  `🐼 Elysia is running at ${app.server?.hostname}:${app.server?.port}`,
);
``,`
```

This will allow you to access the `user` and `session` object in all of your routes.

```
file: ./content/docs/integrations/expo.mdx
meta: {
  "title": "Expo Integration",
  "description": "Integrate Better Auth with Expo."
}
```

Expo is a popular framework for building cross-platform apps with React Native. Better Auth supports both Expo native and web apps.

## ## Installation

### <Steps>

#### <Step>

##### ## Configure A Better Auth Backend

Before using Better Auth with Expo, make sure you have a Better Auth backend set up. You can either use a separate server or leverage Expo's new [API Routes](https://docs.expo.dev/router/reference/api-routes) feature to host your Better Auth instance.

To get started, check out our [installation](/docs/installation) guide for setting up Better Auth on your server. If you prefer to check out the full example, you can find it [here](https://github.com/better-auth/better-auth/tree/main/examples/expo-example).

To use the new API routes feature in Expo to host your Better Auth instance you can create a new API route in your Expo app and mount the Better Auth handler.

```
````ts title="app/api/auth/[...auth]+api.ts"
import { auth } from "@lib/auth"; // import Better Auth handler

const handler = auth.handler;
export { handler as GET, handler as POST }; // export handler for both GET and POST requests
````
```

#### </Step>

#### <Step>

##### ## Install Server Dependencies

Install both the Better Auth package and Expo plugin into your server application.

```
<Tabs groupId="package-manager" persist items={}>
  <Tab value="npm">
    ````bash
    npm install better-auth @better-auth/expo
    ````
  </Tab>

  <Tab value="pnpm">
    ````bash
```

```

    pnpm add better-auth @better-auth/expo
    ```
  </Tab>

  <Tab value="yarn">
    ``` bash
    yarn add better-auth @better-auth/expo
    ```
  </Tab>

  <Tab value="bun">
    ``` bash
    bun add better-auth @better-auth/expo
    ```
  </Tab>
</Tabs>
</Step>

```

```

<Step>
## Install Client Dependencies

```

You also need to install both the Better Auth package and Expo plugin into your Expo application.

```

<Tabs groupId="package-manager" persist items={}>
  <Tab value="npm">
    ``` bash
    npm install better-auth @better-auth/expo
    ```
  </Tab>

  <Tab value="pnpm">
    ``` bash
    pnpm add better-auth @better-auth/expo
    ```
  </Tab>

  <Tab value="yarn">
    ``` bash
    yarn add better-auth @better-auth/expo
    ```
  </Tab>

  <Tab value="bun">
    ``` bash
    bun add better-auth @better-auth/expo
    ```
  </Tab>
</Tabs>

```

If you plan on using our social integrations (Google, Apple etc.) then there are a few more dependencies that are required in your Expo app. In the default Expo template these are already installed so you may be able to skip this step if you have these dependencies already.

```

<Tabs groupId="package-manager" persist items={}>
  <Tab value="npm">
    ``` bash
    npm install expo-linking expo-web-browser expo-constants
    ```
  </Tab>

  <Tab value="pnpm">
    ``` bash
    pnpm add expo-linking expo-web-browser expo-constants

```

```

    ` ` `
  </Tab>

  <Tab value="yarn">
    ` ` ` bash
    yarn add expo-linking expo-web-browser expo-constants

    ` ` `
  </Tab>

  <Tab value="bun">
    ` ` ` bash
    bun add expo-linking expo-web-browser expo-constants

    ` ` `
  </Tab>
</Tabs>
</Step>

```

<Step>  
## Add the Expo Plugin on Your Server

Add the Expo plugin to your Better Auth server.

```

` ` ` ts title="lib/auth.ts"
import { betterAuth } from "better-auth";
import { expo } from "@better-auth/expo";

export const auth = betterAuth({
  plugins: [expo()],
  emailAndPassword: {
    enabled: true, // Enable authentication using email and password.
  },
});
` ` `
</Step>

```

<Step>  
## Initialize Better Auth Client

To initialize Better Auth in your Expo app, you need to call `createAuthClient` with the base URL of your Better Auth backend. Make sure to import the client from `./react`.

Make sure you install the `expo-secure-store` package into your Expo app. This is used to store the session data and cookies securely.

```

<Tabs groupId="package-manager" persist items={}>
  <Tab value="npm">
    ` ` ` bash
    npm install expo-secure-store
    ` ` `
  </Tab>

  <Tab value="pnpm">
    ` ` ` bash
    pnpm add expo-secure-store
    ` ` `
  </Tab>

  <Tab value="yarn">
    ` ` ` bash
    yarn add expo-secure-store
    ` ` `
  </Tab>

```

```

<Tab value="bun">
  ```bash
  bun add expo-secure-store
  ```
</Tab>
</Tabs>

```

You need to also import client plugin from `@better-auth/expo/client` and pass it to the `plugins` array when initializing the auth client.

This is important because:

**Social Authentication Support:** enables social auth flows by handling authorization URLs and callbacks within the Expo web browser.

**Secure Cookie Management:** stores cookies securely and automatically adds them to the headers of your auth requests.

```

```ts title="lib/auth-client.ts"
import { createAuthClient } from "better-auth/react";
import { expoClient } from "@better-auth/expo/client";
import * as SecureStore from "expo-secure-store";

export const authClient = createAuthClient({
  baseURL: "http://localhost:8081", // Base URL of your Better Auth backend.
  plugins: [
    expoClient({
      scheme: "myapp",
      storagePrefix: "myapp",
      storage: SecureStore,
    })
  ]
});
```

```

<Callout>

Be sure to include the full URL, including the path, if you've changed the default path from `/api/auth`.

</Callout>

</Step>

<Step>

## Scheme and Trusted Origins

Better Auth uses deep links to redirect users back to your app after authentication. To enable this, you need to add your app's scheme to the `trustedOrigins` list in your Better Auth config.

First, make sure you have a scheme defined in your `app.json` file.

```

```json title="app.json"
{
  "expo": {
    "scheme": "myapp"
  }
}
```

```

Then, update your Better Auth config to include the scheme in the `trustedOrigins` list.

```

```ts title="auth.ts"
export const auth = betterAuth({
  trustedOrigins: ["myapp://"]
});
```

```

If you have multiple schemes or need to support deep linking with various paths, you can use specific patterns or wildcards:



```

` `` ts title="auth.ts"
export const auth = betterAuth({
  trustedOrigins: [
    // Basic scheme
    "myapp://",

    // Production & staging schemes
    "myapp-prod://",
    "myapp-staging://",

    // Wildcard support for all paths following the scheme
    "myapp:/*"
  ]
})
` ``

```

<Callout>

The wildcard pattern can be particularly useful if your app uses different URL formats for deep linking based on features or screens.

</Callout>

</Step>

<Step>

## Configure Metro Bundler

To resolve Better Auth exports you'll need to enable `unstable\_enablePackageExports` in your metro config.

```

` `` js title="metro.config.js"
const { getDefaultConfig } = require("expo/metro-config");

const config = getDefaultConfig(__dirname)

config.resolver.unstable_enablePackageExports = true; // [!code highlight]

module.exports = config;
` ``

```

<Callout>In case you don't have a `metro.config.js` file in your project run `npx expo customize metro.config.js`.</Callout>

If you can't enable `unstable\_enablePackageExports` option, you can use [babel-plugin-module-resolver] (<https://github.com/tleunen/babel-plugin-module-resolver>) to manually resolve the paths.

```

` `` ts title="babel.config.js"
module.exports = function (api) {
  api.cache(true);
  return {
    presets: ["babel-preset-expo"],
    plugins: [
      [
        "module-resolver",
        {
          alias: {
            "better-auth/react": "../node_modules/better-auth/dist/client/react/index.cjs",
            "better-auth/client/plugins": "../node_modules/better-auth/dist/client/plugins/index.cjs",
            "@better-auth/expo/client": "../node_modules/@better-auth/expo/dist/client.cjs",
          },
        },
      ],
    ],
  },
];
}
` ``

```

<Callout>In case you don't have a `babel.config.js` file in your project run `npx expo customize babel.config.js`.</Callout>

Don't forget to clear the cache after making changes.

```
```bash
npx expo start --clear
```

</Step>
</Steps>
```

## ## Usage

### ### Authenticating Users

With Better Auth initialized, you can now use the `authClient` to authenticate users in your Expo app.

```
<Tabs items={["sign-in", "sign-up"]}>
  <Tab value="sign-in">
    ```tsx title="app/sign-in.tsx"
    import { useState } from "react";
    import { View, TextInput, Button } from "react-native";
    import { authClient } from "@lib/auth-client";

    export default function SignIn() {
      const [email, setEmail] = useState("");
      const [password, setPassword] = useState("");

      const handleLogin = async () => {
        await authClient.signIn.email({
          email,
          password,
        })
      };

      return (
        <View>
          <TextInput
            placeholder="Email"
            value={email}
            onChangeText={setEmail}
          />
          <TextInput
            placeholder="Password"
            value={password}
            onChangeText={setPassword}
          />
          <Button title="Login" onPress={handleLogin} />
        </View>
      );
    }
  </Tab>

  <Tab value="sign-up">
    ```tsx title="app/sign-up.tsx"
    import { useState } from "react";
    import { View, TextInput, Button } from "react-native";
    import { authClient } from "@lib/auth-client";

    export default function SignUp() {
      const [email, setEmail] = useState("");
      const [name, setName] = useState("");
      const [password, setPassword] = useState("");

      const handleLogin = async () => {
        await authClient.signUp.email({
```

```

        email,
        password,
        name
    ))
};

return (
  <View>
    <TextInput
      placeholder="Name"
      value={name}
      onChangeText={setName}
    />
    <TextInput
      placeholder="Email"
      value={email}
      onChangeText={setEmail}
    />
    <TextInput
      placeholder="Password"
      value={password}
      onChangeText={setPassword}
    />
    <Button title="Login" onPress={handleLogin} />
  </View>
);
}
...
</Tab>
</Tabs>

```

#### #### Social Sign-In

For social sign-in, you can use the `authClient.signIn.social` method with the provider name and a callback URL.

```

```tsx title="app/social-sign-in.tsx"
import { Button } from "react-native";

export default function SocialSignIn() {
  const handleLogin = async () => {
    await authClient.signIn.social({
      provider: "google",
      callbackURL: "/dashboard" // this will be converted to a deep link (eg. `myapp://dashboard`) on native
    })
  };
  return <Button title="Login with Google" onPress={handleLogin} />;
}
```

```

#### #### IdToken Sign-In

If you want to make provider request on the mobile device and then verify the ID token on the server, you can use the `authClient.signIn.social` method with the `idToken` option.

```

```tsx title="app/social-sign-in.tsx"
import { Button } from "react-native";

export default function SocialSignIn() {
  const handleLogin = async () => {
    await authClient.signIn.social({
      provider: "google", // only google, apple and facebook are supported for idToken signIn
      idToken: {
        token: "...", // ID token from provider
        nonce: "...", // nonce from provider (optional)
      }
    })
  };
}

```

```

    callbackURL: "/dashboard" // this will be converted to a deep link (eg. `myapp://dashboard`) on native
  })
};
return <Button title="Login with Google" onPress={handleLogin} />;
}
` ``

```

### ### Session

Better Auth provides a `useSession` hook to access the current user's session in your app.

```

` `` tsx title="app/index.tsx"
import { Text } from "react-native";
import { authClient } from "@lib/auth-client";

export default function Index() {
  const { data: session } = authClient.useSession();

  return <Text>Welcome, {session?.user.name}</Text>;
}
` ``

```

On native, the session data will be cached in SecureStore. This will allow you to remove the need for a loading spinner when the app is reloaded. You can disable this behavior by passing the `disableCache` option to the client.

### ### Making Authenticated Requests to Your Server

To make authenticated requests to your server that require the user's session, you have to retrieve the session cookie from `SecureStore` and manually add it to your request headers.

```

` `` tsx
import { authClient } from "@lib/auth-client";

const makeAuthenticatedRequest = async () => {
  const cookies = authClient.getCookie(); // [!code highlight]
  const headers = {
    "Cookie": cookies, // [!code highlight]
  };
  const response = await fetch("http://localhost:8081/api/secure-endpoint", { headers });
  const data = await response.json();
  return data;
};
` ``

```

### \*\*Example: Usage With TRPC\*\*

```

` `` tsx title="lib/trpc-provider.tsx"
//...other imports
import { authClient } from "@lib/auth-client"; // [!code highlight]

export const api = createTRPCReact<AppRouter>();

export function TRPCProvider(props: { children: React.ReactNode }) {
  const [queryClient] = useState(() => new QueryClient());
  const [trpcClient] = useState(() =>
    api.createClient({
      links: [
        httpBatchLink({
          //...your other options
          headers() {
            const headers = new Map<string, string>(); // [!code highlight]
            const cookies = authClient.getCookie(); // [!code highlight]
            if (cookies) { // [!code highlight]
              headers.set("Cookie", cookies); // [!code highlight]
            } // [!code highlight]
          }
        })
      ]
    })
  );
}

```

```

    return Object.fromEntries(headers); // [!code highlight]
  },
 )),
],
)),
);

return (
  <api.Provider client={trpcClient} queryClient={queryClient}>
    <QueryClientProvider client={queryClient}>
      {props.children}
    </QueryClientProvider>
  </api.Provider>
);
}
``,`

```

### ### Options

#### #### Expo Client

**\*\*storage\*\***: the storage mechanism used to cache the session data and cookies.

```

``,` ts title="lib/auth-client.ts"
import { createAuthClient } from "better-auth/react";
import SecureStorage from "expo-secure-store";

const authClient = createAuthClient({
  baseURL: "http://localhost:8081",
  storage: SecureStorage
});
``,`

```

**\*\*scheme\*\***: scheme is used to deep link back to your app after a user has authenticated using oAuth providers. By default, Better Auth tries to read the scheme from the `app.json` file. If you need to override this, you can pass the scheme option to the client.

```

``,` ts title="lib/auth-client.ts"
import { createAuthClient } from "better-auth/react";

const authClient = createAuthClient({
  baseURL: "http://localhost:8081",
  scheme: "myapp"
});
``,`

```

**\*\*disableCache\*\***: By default, the client will cache the session data in SecureStore. You can disable this behavior by passing the `disableCache` option to the client.

```

``,` ts title="lib/auth-client.ts"
import { createAuthClient } from "better-auth/react";

const authClient = createAuthClient({
  baseURL: "http://localhost:8081",
  disableCache: true
});
``,`

```

#### #### Expo Servers

Server plugin options:

**\*\*overrideOrigin\*\***: Override the origin for Expo API routes (default: false). Enable this if you're facing cors origin issues with Expo API routes.

```
file: ./content/docs/integrations/express.mdx
meta: {
  "title": "Express Integration",
  "description": "Integrate Better Auth with Express."
}
```

This guide will show you how to integrate Better Auth with `[express.js](https://expressjs.com/)`.

Before you start, make sure you have a Better Auth instance configured. If you haven't done that yet, check out the [\[installation\]\(/docs/installation\)](#).

<Callout>

Note that CommonJS (cjs) isn't supported. Use ECMAScript Modules (ESM) by setting `"type": "module"` in your `package.json` or configuring your `tsconfig.json` to use ES modules.

</Callout>

### ### Mount the handler

To enable Better Auth to handle requests, we need to mount the handler to an API route. Create a catch-all route to manage all requests to `/api/auth/*` in case of ExpressJS v4 or `/api/auth/*splat` in case of ExpressJS v5 (or any other path specified in your Better Auth options).

<Callout type="warn">

Don't use `express.json()` before the Better Auth handler. Use it only for other routes, or the client API will get stuck on "pending".

</Callout>

```
```ts title="server.ts"
import express from "express";
import { toNodeHandler } from "better-auth/node";
import { auth } from "./auth";

const app = express();
const port = 3005;

app.all("/api/auth/*", toNodeHandler(auth)); // For ExpressJS v4
// app.all("/api/auth/*splat", toNodeHandler(auth)); For ExpressJS v5

// Mount express json middleware after Better Auth handler
// or only apply it to routes that don't interact with Better Auth
app.use(express.json());

app.listen(port, () => {
  console.log(`Example app listening on port ${port}`);
});
```
```

After completing the setup, start your server. Better Auth will be ready to use. You can send a `GET` request to the `/ok` endpoint (`/api/auth/ok`) to verify that the server is running.

### ### Cors Configuration

To add CORS (Cross-Origin Resource Sharing) support to your Express server when integrating Better Auth, you can use the `cors` middleware. Below is an updated example showing how to configure CORS for your server:

```
```ts
import express from "express";
import cors from "cors"; // Import the CORS middleware
import { toNodeHandler, fromNodeHeaders } from "better-auth/node";
import { auth } from "./auth";

const app = express();
const port = 3005;
```

```
// Configure CORS middleware
app.use(
  cors({
    origin: "http://your-frontend-domain.com", // Replace with your frontend's origin
    methods: ["GET", "POST", "PUT", "DELETE"], // Specify allowed HTTP methods
    credentials: true, // Allow credentials (cookies, authorization headers, etc.)
  })
);
````
```

### ### Getting the User Session

To retrieve the user's session, you can use the `getSession`` method provided by the `auth`` object. This method requires the request headers to be passed in a specific format. To simplify this process, Better Auth provides a `fromNodeHeaders`` helper function that converts Node.js request headers to the format expected by Better Auth (a `Headers`` object).

Here's an example of how to use `getSession`` in an Express route:

```
```ts title="server.ts"
import { fromNodeHeaders } from "better-auth/node";
import { auth } from "../auth"; // Your Better Auth instance

app.get("/api/me", async (req, res) => {
  const session = await auth.api.getSession({
    headers: fromNodeHeaders(req.headers),
  });
  return res.json(session);
});
```
```

```
file: ./content/docs/integrations/fastify.mdx
meta: {
  "title": "Better Auth Fastify Integration Guide",
  "description": "Learn how to seamlessly integrate Better Auth with your Fastify application."
}
```

This guide provides step-by-step instructions for configuring both essential handlers and CORS settings.

<Callout type="important">

A configured Better Auth instance is required before proceeding. If you haven't set this up yet, please consult our [Installation Guide](/docs/installation).

</Callout>

### ### Prerequisites

Verify the following requirements before integration:

```
***Node.js Environment***: v16 or later installed
***ES Module Support***: Enable ES modules in either:
* `package.json`: `{ "type": "module" }`
* TypeScript `tsconfig.json`: `{ "module": "ESNext" }`
***Fastify Dependencies***:
<Tabs groupId="package-manager" persist items={}
  <Tab value="npm">
    ```bash
    npm install fastify @fastify/cors
    ```
  </Tab>

  <Tab value="pnpm">
    ```bash
    pnpm add fastify @fastify/cors
    ```
  </Tab>
```

```

<Tab value="yarn">
  ```bash
  yarn add fastify @fastify/cors
  ```
</Tab>

<Tab value="bun">
  ```bash
  bun add fastify @fastify/cors
  ```
</Tab>
</Tabs>

```

<Callout type="tip"> For TypeScript: Ensure your `tsconfig.json` includes `"esModuleInterop": true` for optimal compatibility. </Callout>

### Authentication Handler Setup

Configure Better Auth to process authentication requests by creating a catch-all route:

```

```ts title="server.ts"
import Fastify from "fastify";
import { auth } from "../auth"; // Your configured Better Auth instance

const fastify = Fastify({ logger: true });

// Register authentication endpoint
fastify.route({
  method: ["GET", "POST"],
  url: "/api/auth/*",
  async handler(request, reply) {
    try {
      // Construct request URL
      const url = new URL(request.url, `http://${request.headers.host}`);

      // Convert Fastify headers to standard Headers object
      const headers = new Headers();
      Object.entries(request.headers).forEach(([key, value]) => {
        if (value) headers.append(key, value.toString());
      });

      // Create Fetch API-compatible request
      const req = new Request(url.toString(), {
        method: request.method,
        headers,
        body: request.body ? JSON.stringify(request.body) : undefined,
      });

      // Process authentication request
      const response = await auth.handler(req);

      // Forward response to client
      reply.status(response.status);
      response.headers.forEach((value, key) => reply.header(key, value));
      reply.send(response.body ? await response.text() : null);
    } catch (error) {
      fastify.log.error("Authentication Error:", error);
      reply.status(500).send({
        error: "Internal authentication error",
        code: "AUTH_FAILURE"
      });
    }
  }
});

```



```
});

// Initialize server
fastify.listen({ port: 4000 }, (err) => {
  if (err) {
    fastify.log.error(err);
    process.exit(1);
  }
  console.log("Server running on port 4000");
});
```,
```

### ### Trusted origins

When a request is made from a different origin, the request will be blocked by default. You can add trusted origins to the `auth` instance.

```
```ts
export const auth = betterAuth({
  trustedOrigins: ["http://localhost:3000", "https://example.com"],
});
```,
```

### ### Configuring CORS

Secure your API endpoints with proper CORS configuration:

```
```ts
import fastifyCors from "@fastify/cors";

// Configure CORS policies
fastify.register(fastifyCors, {
  origin: process.env.CLIENT_ORIGIN || "http://localhost:3000",
  methods: ["GET", "POST", "PUT", "DELETE", "OPTIONS"],
  allowedHeaders: [
    "Content-Type",
    "Authorization",
    "X-Requested-With"
  ],
  credentials: true,
  maxAge: 86400
});

// Mount authentication handler after CORS registration
// (Use previous handler configuration here)
```,
```

<Callout type="warning"> Always restrict CORS origins in production environments. Use environment variables for dynamic configuration. </Callout>

```
file: ./content/docs/integrations/hono.mdx
meta: {
  "title": "Hono Integration",
  "description": "Integrate Better Auth with Hono."
}
```

Before you start, make sure you have a Better Auth instance configured. If you haven't done that yet, check out the [installation](/docs/installation).

### ### Mount the handler

We need to mount the handler to Hono endpoint.

```
```ts
```

```
import { Hono } from "hono";
import { auth } from "../auth";
import { serve } from "@hono/node-server";
import { cors } from "hono/cors";

const app = new Hono();

app.on(["POST", "GET"], "/api/auth/*", (c) => {
  return auth.handler(c.req.raw);
});

serve(app);
````
```

### ### Cors

To configure cors, you need to use the `cors` plugin from `hono/cors`.

```
````ts
import { Hono } from "hono";
import { auth } from "../auth";
import { serve } from "@hono/node-server";
import { cors } from "hono/cors";

const app = new Hono();

app.use(
  "/api/auth/*", // or replace with "*" to enable cors for all routes
  cors({
    origin: "http://localhost:3001", // replace with your origin
    allowHeaders: ["Content-Type", "Authorization"],
    allowMethods: ["POST", "GET", "OPTIONS"],
    exposeHeaders: ["Content-Length"],
    maxAge: 600,
    credentials: true,
  }),
);
````
```

### ### Middleware

You can add a middleware to save the `session` and `user` in a `context` and also add validations for every route.

```
````ts
import { Hono } from "hono";
import { auth } from "../auth";
import { serve } from "@hono/node-server";
import { cors } from "hono/cors";

const app = new Hono<{
  Variables: {
    user: typeof auth.$Infer.Session.user | null;
    session: typeof auth.$Infer.Session.session | null
  }
}>();

app.use("*", async (c, next) => {
  const session = await auth.api.getSession({ headers: c.req.raw.headers });

  if (!session) {
    c.set("user", null);
    c.set("session", null);
    return next();
  }
});
```

```

    c.set("user", session.user);
    c.set("session", session.session);
    return next();
  });

app.on(["POST", "GET"], "/api/auth/*", (c) => {
  return auth.handler(c.req.raw);
});

serve(app);
```

```

This will allow you to access the `user` and `session` object in all of your routes.

```

```ts
app.get("/session", (c) => {
  const session = c.get("session")
  const user = c.get("user")

  if(!user) return c.body(null, 401);

  return c.json({
    session,
    user
  });
});
```

```

### ### Cross-Domain Cookies

By default, all Better Auth cookies are set with `SameSite=Lax`. If you need to use cookies across different domains, you'll need to set `SameSite=None` and `Secure=true`. However, we recommend using subdomains whenever possible, as this allows you to keep `SameSite=Lax`. To enable cross-subdomain cookies, simply turn on `crossSubDomainCookies` in your auth config.

```

```ts title="auth.ts"
export const auth = createAuth({
  advanced: {
    crossSubDomainCookies: {
      enabled: true
    }
  }
});
```

```

If you still need to set `SameSite=None` and `Secure=true`, you can adjust these attributes globally through `cookieOptions` in the `createAuth` configuration.

```

```ts title="auth.ts"
export const auth = createAuth({
  advanced: {
    defaultCookieAttributes: {
      sameSite: "none",
      secure: true,
      partitioned: true // New browser standards will mandate this for foreign cookies
    }
  }
});
```

```

You can also customize cookie attributes individually by setting them within `cookies` in your auth config.

```

```ts title="auth.ts"

```

```
export const auth = createAuth({
  advanced: {
    cookies: {
      sessionToken: {
        sameSite: "none",
        secure: true,
        partitioned: true // New browser standards will mandate this for foreign cookies
      }
    }
  }
})
````
```

### ### Client-Side Configuration

When using the Hono client (`@hono/client`) to make requests to your Better Auth-protected endpoints, you need to configure it to send credentials (cookies) with cross-origin requests.

```
```` ts title="api.ts"
import { hc } from "hono/client";
import type { AppType } from "../server"; // Your Hono app type

const client = hc<AppType>("http://localhost:8787", {
  fetch: ((input, init) => {
    return fetch(input, {
      ...init,
      credentials: "include" // Required for sending cookies cross-origin
    });
  }) satisfies typeof fetch,
});

// Now your client requests will include credentials
const response = await client.someProtectedEndpoint.$get();
````
```

This configuration is necessary when:

- \* Your client and server are on different domains/ports during development
- \* You're making cross-origin requests in production
- \* You need to send authentication cookies with your requests

The `credentials: "include"` option tells the fetch client to send cookies even for cross-origin requests. This works in conjunction with the CORS configuration on your server that has `credentials: true`.

> **Note:** Make sure your CORS configuration on the server matches your client's domain, and that `credentials: true` is set in both the server's CORS config and the client's fetch config.

```
file: ./content/docs/integrations/next.mdx
meta: {
  "title": "Next.js integration",
  "description": "Integrate Better Auth with Next.js."
}
```

Better Auth can be easily integrated with Next.js. Before you start, make sure you have a Better Auth instance configured. If you haven't done that yet, check out the [installation](/docs/installation).

### ### Create API Route

We need to mount the handler to an API route. Create a route file inside `/api/auth/[...all]` directory. And add the following code:

```
```` ts title="api/auth/[...all]/route.ts"
import { auth } from "@lib/auth";
import { toNextJsHandler } from "better-auth/next-js";
```

```
export const { GET, POST } = toNextJsHandler(auth.handler);
````
```

<Callout type="info">

You can change the path on your better-auth configuration but it's recommended to keep it as `/api/auth/[...all]`

</Callout>

For `pages` route, you need to use `toNodeHandler` instead of `toNextJsHandler` and set `bodyParser` to `false` in the `config` object. Here is an example:

```
````ts title="pages/api/auth/[...all].ts"
import { toNodeHandler } from "better-auth/node"
import { auth } from "@lib/auth"
```

```
// Disallow body parsing, we will parse it manually
export const config = { api: { bodyParser: false } }
```

```
export default toNodeHandler(auth.handler)
````
```

### ## Create a client

Create a client instance. You can name the file anything you want. Here we are creating `client.ts` file inside the `lib/` directory.

```
````ts title="auth-client.ts"
import { createAuthClient } from "better-auth/react" // make sure to import from better-auth/react

export const authClient = createAuthClient({
  //you can pass client configuration here
})
````
```

Once you have created the client, you can use it to sign up, sign in, and perform other actions.

Some of the actions are reactive. The client uses [nano-store](https://github.com/nanostores/nanostores) to store the state and re-render the components when the state changes.

The client also uses [better-fetch](https://github.com/bekacru/better-fetch) to make the requests. You can pass the fetch configuration to the client.

### ## RSC and Server actions

The `api` object exported from the auth instance contains all the actions that you can perform on the server. Every endpoint made inside Better Auth is invocable as a function. Including plugins endpoints.

#### **\*\*Example: Getting Session on a server action\*\***

```
````tsx title="server.ts"
import { auth } from "@lib/auth"
import { headers } from "next/headers"

const someAuthenticatedAction = async () => {
  "use server";
  const session = await auth.api.getSession({
    headers: await headers()
  })
};
````
```

#### **\*\*Example: Getting Session on a RSC\*\***

```
````tsx
import { auth } from "@lib/auth"
import { headers } from "next/headers"
```

```
export async function ServerComponent() {
  const session = await auth.api.getSession({
    headers: await headers()
  })
  if(!session) {
    return <div>Not authenticated</div>
  }
  return (
    <div>
      <h1>Welcome {session.user.name}</h1>
    </div>
  )
}
``,`
```

**<Callout type="warn">**As RSCs cannot set cookies, the [cookie cache](/docs/concepts/session-management#cookie-cache) will not be refreshed until the server is interacted with from the client via Server Actions or Route Handlers.**</Callout>**

### ### Server Action Cookies

When you call a function that needs to set cookies, like `signInEmail` or `signUpEmail` in a server action, cookies won't be set. This is because server actions need to use the `cookies` helper from Next.js to set cookies.

To simplify this, you can use the `nextCookies` plugin, which will automatically set cookies for you whenever a `Set-Cookie` header is present in the response.

```
` `` ts title="auth.ts"
import { betterAuth } from "better-auth";
import { nextCookies } from "better-auth/next-js";

export const auth = betterAuth({
  //...your config
  plugins: [nextCookies()] // make sure this is the last plugin in the array // [!code highlight]
})
``,`
```

Now, when you call functions that set cookies, they will be automatically set.

```
` `` ts
"use server";
import { auth } from "@/lib/auth"

const signIn = async () => {
  await auth.api.signInEmail({
    body: {
      email: "user@email.com",
      password: "password",
    }
  })
}
}
``,`
```

### ### Middleware

In Next.js middleware, it's recommended to only check for the existence of a session cookie to handle redirection. To avoid blocking requests by making API or database calls.

You can use the `getSessionCookie` helper from Better Auth for this purpose:

**<Callout type="warn">**The `getSessionCookie()` function does not automatically reference the auth config specified in `auth.ts`. Therefore, if you customized the cookie name or prefix, you need to ensure that the configuration in `getSessionCookie()` matches the config defined in your `auth.ts`.**</Callout>**

```

```ts
import { NextRequest, NextResponse } from "next/server";
import { getSessionCookie } from "better-auth/cookies";

export async function middleware(request: NextRequest) {
  const sessionCookie = getSessionCookie(request);

  if (!sessionCookie) {
    return NextResponse.redirect(new URL("/", request.url));
  }

  return NextResponse.next();
}

export const config = {
  matcher: ["/dashboard"], // Specify the routes the middleware applies to
};
```

```

<Callout type="info">

If you have a custom cookie name or prefix, you can pass it to the `getSessionCookie` function.

```

```ts
const sessionCookie = getSessionCookie(request, {
  cookieName: "my_session_cookie",
  cookiePrefix: "my_prefix"
});
```

```

</Callout>

Alternatively, you can use the `getCookieCache` helper to get the session object from the cookie cache.

```

```ts
import { getCookieCache } from "better-auth/cookies";

export async function middleware(request: NextRequest) {
  const session = await getCookieCache(request);
  if (!session) {
    return NextResponse.redirect(new URL("/sign-in", request.url));
  }
  return NextResponse.next();
}
```

```

### For Next.js release `15.1.7` and below

If you need the full session object, you'll have to fetch it from the `/get-session` API route. Since Next.js middleware doesn't support running Node.js APIs directly, you must make an HTTP request.

<Callout>

The example uses [better-fetch](https://better-fetch.vercel.app), but you can use any fetch library.

</Callout>

```

```ts
import { betterFetch } from "@better-fetch/fetch";
import type { auth } from "@lib/auth";
import { NextRequest, NextResponse } from "next/server";

type Session = typeof auth.$Infer.Session;

export async function middleware(request: NextRequest) {
  const { data: session } = await betterFetch<Session>("/api/auth/get-session", {
    baseURL: request.nextUrl.origin,
    headers: {

```

```

    cookie: request.headers.get("cookie") || "", // Forward the cookies from the request
  },
});

if (!session) {
  return NextResponse.redirect(new URL("/sign-in", request.url));
}

return NextResponse.next();
}

export const config = {
  matcher: ["/dashboard"], // Apply middleware to specific routes
};

```

### For Next.js release `15.2.0` and above

From the version 15.2.0, Next.js allows you to use the `Node.js` runtime in middleware. This means you can use the `auth.api` object directly in middleware.

<Callout type="warn">

You may refer to the [Next.js documentation](https://nextjs.org/docs/app/building-your-application/routing/middleware#runtime) for more information about runtime configuration, and how to enable it. Be careful when using the new runtime. It's an experimental feature and it may be subject to breaking changes.

</Callout>

```

` ` ` ts
import { NextRequest, NextResponse } from "next/server";
import { headers } from "next/headers";
import { auth } from "@lib/auth";

export async function middleware(request: NextRequest) {
  const session = await auth.api.getSession({
    headers: await headers()
  });

  if (!session) {
    return NextResponse.redirect(new URL("/sign-in", request.url));
  }

  return NextResponse.next();
}

export const config = {
  runtime: "nodejs",
  matcher: ["/dashboard"], // Apply middleware to specific routes
};

```

```

file: ./content/docs/integrations/nitro.mdx
meta: {
  "title": "Nitro Integration",
  "description": "Integrate Better Auth with Nitro."
}

```

Better Auth can be integrated with your [Nitro Application](https://nitro.build/) (an open source framework to build web servers).

This guide aims to help you integrate Better Auth with your Nitro application in a few simple steps.

### ## Create a new Nitro Application

Start by scaffolding a new Nitro application using the following command:



```
```bash title="Terminal"
npx giget@latest nitro nitro-app --install
```
```

This will create the `nitro-app` directory and install all the dependencies. You can now open the `nitro-app` directory in your code editor.

### ### Prisma Adapter Setup

<Callout>

This guide assumes that you have a basic understanding of Prisma. If you are new to Prisma, you can check out the [Prisma documentation](https://www.prisma.io/docs/getting-started).

The `sqlite` database used in this guide will not work in a production environment. You should replace it with a production-ready database like `PostgreSQL`.

</Callout>

For this guide, we will be using the Prisma adapter. You can install prisma client by running the following command:

<Tabs groupId="package-manager" persist items={}>

<Tab value="npm">

```
```bash
npm install @prisma/client
```
```

</Tab>

<Tab value="pnpm">

```
```bash
pnpm add @prisma/client
```
```

</Tab>

<Tab value="yarn">

```
```bash
yarn add @prisma/client
```
```

</Tab>

<Tab value="bun">

```
```bash
bun add @prisma/client
```
```

</Tab>

</Tabs>

`prisma` can be installed as a dev dependency using the following command:

<Tabs groupId="package-manager" persist items={}>

<Tab value="npm">

```
```bash
npm install -D prisma
```
```

</Tab>

<Tab value="pnpm">

```
```bash
pnpm add -D prisma
```
```

</Tab>

<Tab value="yarn">

```
```bash
yarn add --dev prisma
```
```

```

</Tab>

<Tab value="bun">
  ```bash
  bun add --dev prisma
  ```
</Tab>
</Tabs>

```

Generate a `schema.prisma` file in the `prisma` directory by running the following command:

```

```bash title="Terminal"
npx prisma init
```

```

You can now replace the contents of the `schema.prisma` file with the following:

```

```prisma title="prisma/schema.prisma"
generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "sqlite"
  url      = env("DATABASE_URL")
}

// Will be deleted. Just need it to generate the prisma client
model Test {
  id Int @id @default(autoincrement())
  name String
}
```

```

Ensure that you update the `DATABASE\_URL` in your `.env` file to point to the location of your database.

```

```env title=".env"
DATABASE_URL="file:./dev.db"
```

```

Run the following command to generate the Prisma client & sync the database:

```

```bash title="Terminal"
npx prisma db push
```

```

### ### Install & Configure Better Auth

Follow steps 1 & 2 from the [installation guide](/docs/installation) to install Better Auth in your Nitro application & set up the environment variables.

Once that is done, create your Better Auth instance within the `server/utils/auth.ts` file.

```

```ts title="server/utils/auth.ts"
import { betterAuth } from "better-auth";
import { prismaAdapter } from "better-auth/adapters/prisma";
import { PrismaClient } from "@prisma/client";

const prisma = new PrismaClient();
export const auth = betterAuth({
  database: prismaAdapter(prisma, { provider: "sqlite" }),
  emailAndPassword: { enabled: true },
});
```

```

### #### Update Prisma Schema

Use the Better Auth CLI to update your Prisma schema with the required models by running the following command:

```
```bash title="Terminal"
npx @better-auth/cli generate --config server/utils/auth.ts
```
```

<Callout>

The `--config`` flag is used to specify the path to the file where you have created your Better Auth instance.

</Callout>

Head over to the ``prisma/schema.prisma`` file & save the file to trigger the format on save.

After saving the file, you can run the ``npx prisma db push`` command to update the database schema.

### ## Mount The Handler

You can now mount the Better Auth handler in your Nitro application. You can do this by adding the following code to your ``server/routes/api/auth/[...all].ts`` file:

```
```ts title="server/routes/api/auth/[...all].ts"
export default defineEventHandler((event) => {
  return auth.handler(toWebRequest(event));
});
```
```

<Callout>

This is a [catch-all](https://nitro.build/guide/routing#catch-all-route) route that will handle all requests to ``/api/auth/*``.

</Callout>

### #### CORS

You can configure CORS for your Nitro app by creating a plugin.

Start by installing the cors package:

```
<Tabs groupId="package-manager" persist items={}>
  <Tab value="npm">
    ```bash
    npm install cors
    ```
  </Tab>

  <Tab value="pnpm">
    ```bash
    pnpm add cors
    ```
  </Tab>

  <Tab value="yarn">
    ```bash
    yarn add cors
    ```
  </Tab>

  <Tab value="bun">
    ```bash
    bun add cors
    ```
  </Tab>
</Tabs>
```

You can now create a new file ``server/plugins/cors.ts`` and add the following code:

```

` `` ts title="server/plugins/cors.ts"
import cors from "cors";
export default defineNitroPlugin((plugin) => {
  plugin.h3App.use(
    fromNodeMiddleware(
      cors({
        origin: "*",
      }),
    ),
  );
});
` ``

```

<Callout>

This will enable CORS for all routes. You can customize the `origin` property to allow requests from specific domains. Ensure that the config is in sync with your frontend application.

</Callout>

### ### Auth Guard/Middleware

You can add an auth guard to your Nitro application to protect routes that require authentication. You can do this by creating a new file `server/utils/require-auth.ts` and adding the following code:

```

` `` ts title="server/utils/require-auth.ts"
import { EventHandler, H3Event } from "h3";
import { fromNodeHeaders } from "better-auth/node";

/**
 * Middleware used to require authentication for a route.
 *
 * Can be extended to check for specific roles or permissions.
 */
export const requireAuth: EventHandler = async (event: H3Event) => {
  const headers = event.headers;

  const session = await auth.api.getSession({
    headers: headers,
  });
  if (!session)
    throw createError({
      statusCode: 401,
      statusMessage: "Unauthorized",
    });
  // You can save the session to the event context for later use
  event.context.auth = session;
};
` ``

```

You can now use this event handler/middleware in your routes to protect them:

```

` `` ts title="server/routes/api/secret.get.ts"
// Object syntax of the route handler
export default defineEventHandler({
  // The user has to be logged in to access this route
  onRequest: [requireAuth],
  handler: async (event) => {
    setResponseStatus(event, 201, "Secret data");
    return { message: "Secret data" };
  },
});
` ``

```

### ### Example

You can find an example of a Nitro application integrated with Better Auth & Prisma [here]

(<https://github.com/BayBreezy/nitrojs-better-auth-prisma>).

```
file: ./content/docs/integrations/nuxt.mdx
meta: {
  "title": "Nuxt Integration",
  "description": "Integrate Better Auth with Nuxt."
}
```

Before you start, make sure you have a Better Auth instance configured. If you haven't done that yet, check out the [installation](/docs/installation).

### ### Create API Route

We need to mount the handler to an API route. Create a file inside `/server/api/auth` called `[...all].ts` and add the following code:

```
```ts title="server/api/auth/[...all].ts"
import { auth } from "~/lib/auth"; // import your auth config

export default defineEventHandler((event) => {
  return auth.handler(toWebRequest(event));
});
```
```

<Callout type="info">

You can change the path on your better-auth configuration but it's recommended to keep it as `/api/auth/[...all]`

</Callout>

### ### Migrate the database

Run the following command to create the necessary tables in your database:

```
```bash
npx @better-auth/cli migrate
```
```

### ## Create a client

Create a client instance. You can name the file anything you want. Here we are creating `client.ts` file inside the `lib/` directory.

```
```ts title="auth-client.ts"
import { createAuthClient } from "better-auth/vue" // make sure to import from better-auth/vue

export const authClient = createAuthClient({
  //you can pass client configuration here
})
```
```

Once you have created the client, you can use it to sign up, sign in, and perform other actions. Some of the actions are reactive.

### ### Example usage

```
```vue title="index.vue"
<script setup lang="ts">
import { authClient } from "~/lib/client"
const session = authClient.useSession()
</script>

<template>
  <div>
    <button v-if="!session?.data" @click="() => authClient.signIn.social({
      provider: 'github'
    })">
      Sign in with GitHub
    </button>
  </div>
</template>
```
```

```

    })">
      Continue with GitHub
    </button>
  </div>
  <pre>{{ session.data }}</pre>
  <button v-if="session.data" @click="authClient.signOut()">
    Sign out
  </button>
</div>
</div>
</template>
` ``

```

### ### Server Usage

The `api` object exported from the auth instance contains all the actions that you can perform on the server. Every endpoint made inside Better Auth is a invocable as a function. Including plugins endpoints.

**\*\*Example: Getting Session on a server API route\*\***

```

` `` tsx title="server/api/example.ts"
import { auth } from "~/lib/auth";

export default defineEventHandler((event) => {
  const session = await auth.api.getSession({
    headers: event.headers
  });

  if(session) {
    // access the session.session && session.user
  }
});
` ``

```

### ### SSR Usage

If you are using Nuxt with SSR, you can use the `useSession` function in the `setup` function of your page component and pass `useFetch` to make it work with SSR.

```

` `` vue title="index.vue"
<script setup lang="ts">
import { authClient } from "~/lib/auth-client";

const { data: session } = await authClient.useSession(useFetch);
</script>

<template>
  <p>
    {{ session }}
  </p>
</template>
` ``

```

### ### Middleware

To add middleware to your Nuxt project, you can use the `useSession` method from the client.

```

` `` ts title="middleware/auth.global.ts"
import { authClient } from "~/lib/auth-client";
export default defineNuxtRouteMiddleware(async (to, from) => {
  const { data: session } = await authClient.useSession(useFetch);
  if (!session.value) {
    if (to.path === "/dashboard") {
      return navigateTo("/");
    }
  }
});

```

```
    }
  });
  ...

```

### ### Resources & Examples

- \* [Nuxt and Nuxt Hub example](https://github.com/atinux/nuxt-hub-better-auth) on GitHub.
- \* [NuxtZzzle is Nuxt, Drizzle ORM example](https://github.com/leamsigc/nuxt-better-auth-drizzle) on GitHub [preview] (https://nuxt-better-auth.giessen.dev/)
- \* [Nuxt example](https://stackblitz.com/github/better-auth/better-auth/tree/main/examples/nuxt-example) on StackBlitz.
- \* [NuxSaaS (Github)](https://github.com/NuxSaaS/NuxSaaS) is a full-stack SaaS Starter Kit that leverages Better Auth for secure and efficient user authentication. [Demo](https://nuxsaas.com/)

```
file: ./content/docs/integrations/remix.mdx
meta: {
  "title": "Remix Integration",
  "description": "Integrate Better Auth with Remix."
}

```

Better Auth can be easily integrated with Remix. This guide will show you how to integrate Better Auth with Remix.

You can follow the steps from [installation](/docs/installation) to get started or you can follow this guide to make it the Remix-way.

If you have followed the installation steps, you can skip the first step.

### ## Create auth instance

Create a file named `auth.server.ts` in one of these locations:

- \* Project root
- \* `lib/` folder
- \* `utils/` folder

You can also nest any of these folders under `app/` folder. (e.g. `app/lib/auth.server.ts`)

And in this file, import Better Auth and create your instance.

<Callout type="warn">  
Make sure to export the auth instance with the variable name `auth` or as a `default` export.  
</Callout>

```
```ts title="app/lib/auth.server.ts"
import { betterAuth } from "better-auth"

export const auth = betterAuth({
  database: {
    provider: "postgres", //change this to your database provider
    url: process.env.DATABASE_URL, // path to your database or connection string
  }
})
```

```

### ## Create API Route

We need to mount the handler to a API route. Create a resource route file `api.auth.\$ts` inside `app/routes/` directory. And add the following code:

```
```ts title="app/routes/api.auth.$ts"
import { auth } from '~/lib/auth.server' // Adjust the path as necessary
import type { LoaderFunctionArgs, ActionFunctionArgs } from "@remix-run/node"

export async function loader({ request }: LoaderFunctionArgs) {

```

```

    return auth.handler(request)
  }

export async function action({ request }: ActionFunctionArgs) {
  return auth.handler(request)
}
```

```

<Callout type="info">

You can change the path on your better-auth configuration but it's recommended to keep it as `routes/api.auth.\$ts`

</Callout>

## ## Create a client

Create a client instance. Here we are creating `auth-client.ts` file inside the `lib/` directory.

```

```ts title="app/lib/auth-client.ts"
import { createAuthClient } from "better-auth/react" // make sure to import from better-auth/react

export const authClient = createAuthClient({
  //you can pass client configuration here
})
```

```

Once you have created the client, you can use it to sign up, sign in, and perform other actions.

## ### Example usage

### #### Sign Up

```

```ts title="app/routes/signup.tsx"
import { Form } from "@remix-run/react"
import { useState } from "react"
import { authClient } from "~/lib/auth.client"

export default function SignUp() {
  const [email, setEmail] = useState("")
  const [name, setName] = useState("")
  const [password, setPassword] = useState("")

  const signUp = async () => {
    await authClient.signUp.email(
      {
        email,
        password,
        name,
      },
      {
        onRequest: (ctx) => {
          // show loading state
        },
        onSuccess: (ctx) => {
          // redirect to home
        },
        onError: (ctx) => {
          alert(ctx.error)
        },
      },
    )
  }

  return (
    <div>
      <h2>
        Sign Up

```



```

</h2>
<Form
  onSubmit={signUp}
>
  <input
    type="text"
    value={name}
    onChange={(e) => setName(e.target.value)}
    placeholder="Name"
  />
  <input
    type="email"
    value={email}
    onChange={(e) => setEmail(e.target.value)}
    placeholder="Email"
  />
  <input
    type="password"
    value={password}
    onChange={(e) => setPassword(e.target.value)}
    placeholder="Password"
  />
  <button
    type="submit"
  >
    Sign Up
  </button>
</Form>
</div>
)
}
...

```

#### #### Sign In

```

```ts title="app/routes/signin.tsx"
import { Form } from "@remix-run/react"
import { useState } from "react"
import { authClient } from "~/services/auth.client"

export default function SignIn() {
  const [email, setEmail] = useState("")
  const [password, setPassword] = useState("")

  const signIn = async () => {
    await authClient.signIn.email(
      {
        email,
        password,
      },
      {
        onRequest: (ctx) => {
          // show loading state
        },
        onSuccess: (ctx) => {
          // redirect to home
        },
        onError: (ctx) => {
          alert(ctx.error)
        },
      },
    )
  }
}

```

```

return (
  <div>
    <h2>
      Sign In
    </h2>
    <Form onSubmit={signIn}>
      <input
        type="email"
        value={email}
        onChange={(e) => setEmail(e.target.value)}
      />
      <input
        type="password"
        value={password}
        onChange={(e) => setPassword(e.target.value)}
      />
      <button
        type="submit"
      >
        Sign In
      </button>
    </Form>
  </div>
)
}
`

```

file: ./content/docs/integrations/solid-start.mdx

```

meta: {
  "title": "SolidStart Integration",
  "description": "Integrate Better Auth with SolidStart."
}

```

Before you start, make sure you have a Better Auth instance configured. If you haven't done that yet, check out the [\[installation\]\(/docs/installation\)](#).

### ### Mount the handler

We need to mount the handler to SolidStart server. Put the following code in your `\*auth.ts` file inside `/routes/api/auth` folder.

```

` ` ` ts title="*auth.ts"
import { auth } from "~/lib/auth";
import { toSolidStartHandler } from "better-auth/solid-start";

export const { GET, POST } = toSolidStartHandler(auth);
` ` `

```

file: ./content/docs/integrations/svelte-kit.mdx

```

meta: {
  "title": "SvelteKit Integration",
  "description": "Integrate Better Auth with SvelteKit."
}

```

Before you start, make sure you have a Better Auth instance configured. If you haven't done that yet, check out the [\[installation\]\(/docs/installation\)](#).

### ### Mount the handler

We need to mount the handler to SvelteKit server hook.

```

` ` ` ts title="hooks.server.ts"
import { auth } from "$lib/auth";

```

```
import { svelteKitHandler } from "better-auth/svelte-kit";

export async function handle({ event, resolve }) {
  return svelteKitHandler({ event, resolve, auth });
}
...
```

### ## Create a client

Create a client instance. You can name the file anything you want. Here we are creating `client.ts` file inside the `lib/` directory.

```
```ts title="auth-client.ts"
import { createAuthClient } from "better-auth/svelte" // make sure to import from better-auth/svelte

export const authClient = createAuthClient({
  // you can pass client configuration here
})
...```
```

Once you have created the client, you can use it to sign up, sign in, and perform other actions.

Some of the actions are reactive. The client use [nano-store](https://github.com/nanostores/nanostores) to store the state and reflect changes when there is a change like a user signing in or out affecting the session state.

### ### Example usage

```
```svelte
<script lang="ts">
  import { authClient } from "$lib/client";
  const session = authClient.useSession();
</script>
<div>
  {#if $session.data}
    <div>
      <p>
        {$session?.data?.user.name}
      </p>
      <button
        on:click={async () => {
          await authClient.signOut();
        }}
      >
        Sign Out
      </button>
    </div>
  {:else}
    <button
      on:click={async () => {
        await authClient.signIn.social({
          provider: "github",
        });
      }}
    >
      Continue with GitHub
    </button>
  {/if}
</div>
...```
```

file: ./content/docs/integrations/tanstack.mdx

```
meta: {
  "title": "TanStack Start Integration",
  "description": "Integrate Better Auth with TanStack Start."
}
```

This integration guide is assuming you are using TanStack Start.

Before you start, make sure you have a Better Auth instance configured. If you haven't done that yet, check out the [\[installation\]\(/docs/installation\)](#).

### ### Mount the handler

We need to mount the handler to a TanStack API endpoint.

Create a new file: `/app/routes/api/auth/\$.ts`

```
```ts title="routes/api/auth/$.ts"
import { auth } from '@lib/auth' // import your auth instance
import { createAPIFileRoute } from '@tanstack/react-start/api'

export const APIRoute = createAPIFileRoute('/api/auth/$')({
  GET: ({ request }) => {
    return auth.handler(request)
  },
  POST: ({ request }) => {
    return auth.handler(request)
  },
})
```
```

If you haven't defined an API Route yet, you can do so by creating a file: `/app/api.ts`

```
```ts title="app/api.ts"
import {
  createStartAPIHandler,
  defaultAPIFileRouteHandler,
} from '@tanstack/react-start/api'

export default createStartAPIHandler(defaultAPIFileRouteHandler)
```
```

### ### Usage tips

- \* We recommend using the client SDK or `authClient` to handle authentication, rather than server actions with `auth.api`.
- \* When you call functions that need to set cookies (like `signInEmail` or `signUpEmail`), you'll need to handle cookie setting for TanStack Start. Better Auth provides a `reactStartCookies` plugin to automatically handle this for you.

```
```ts title="auth.ts"
import { betterAuth } from "better-auth";
import { reactStartCookies } from "better-auth/react-start";

export const auth = betterAuth({
  //...your config
  plugins: [reactStartCookies()] // make sure this is the last plugin in the array
})
```
```

Now, when you call functions that set cookies, they will be automatically set using TanStack Start's cookie handling system.

```
```ts
import { auth } from "@lib/auth"

const signIn = async () => {
  await auth.api.signInEmail({
    body: {
      email: "user@email.com",
      password: "password",
    }
  })
}
```
```

```
file: ./content/docs/plugins/2fa.mdx
meta: {
  "title": "Two-Factor Authentication (2FA)",
  "description": "Enhance your app's security with two-factor authentication."
}
```

`OTP` `TOTP` `Backup Codes` `Trusted Devices`

Two-Factor Authentication (2FA) adds an extra security step when users log in. Instead of just using a password, they'll need to provide a second form of verification. This makes it much harder for unauthorized people to access accounts, even if they've somehow gotten the password.

This plugin offers two main methods to do a second factor verification:

1. **OTP (One-Time Password)**: A temporary code sent to the user's email or phone.
2. **TOTP (Time-based One-Time Password)**: A code generated by an app on the user's device.

**Additional features include:**

- \* Generating backup codes for account recovery
- \* Enabling/disabling 2FA
- \* Managing trusted devices

## ## Installation

<Steps>

<Step>

#### Add the plugin to your auth config

Add the two-factor plugin to your auth configuration and specify your app name as the issuer.

```
```ts title="auth.ts"
import { betterAuth } from "better-auth"
import { twoFactor } from "better-auth/plugins" // [!code highlight]

export const auth = betterAuth({
  // ... other config options
  appName: "My App", // provide your app name. It'll be used as an issuer. // [!code highlight]
  plugins: [
    twoFactor() // [!code highlight]
  ]
})
```
```

</Step>

<Step>

#### Migrate the database

Run the migration or generate the schema to add the necessary fields and tables to the database.

```
<Tabs items=["migrate", "generate"]>
<Tab value="migrate">
  ```bash
  npx @better-auth/cli migrate
  ```
</Tab>

<Tab value="generate">
  ```bash
  npx @better-auth/cli generate
  ```
</Tab>
```

</Tabs>

See the [Schema](#schema) section to add the fields manually.

</Step>

<Step>

#### Add the client plugin

Add the client plugin and Specify where the user should be redirected if they need to verify 2nd factor

```
```ts title="auth-client.ts"
import { createAuthClient } from "better-auth/client"
import { twoFactorClient } from "better-auth/client/plugins"

export const authClient = createAuthClient({
  plugins: [
    twoFactorClient()
  ]
})
```
```

</Step>

</Steps>

## Usage

#### Enabling 2FA

To enable two-factor authentication, call `twoFactor.enable` with the user's password and issuer (optional):

```
```ts title="two-factor.ts"
const { data } = await authClient.twoFactor.enable({
  password: "password", // user password required
  issuer: "my-app-name", // Optional, defaults to the app name
});
```
```

When 2FA is enabled:

- \* An encrypted `secret` and `backupCodes` are generated.
- \* `enable` returns `totpURI` and `backupCodes`.

Note: `twoFactorEnabled` won't be set to `true` until the user verifies their TOTP code.

To verify, display the QR code for the user to scan with their authenticator app. After they enter the code, call `verifyTotp`:

```
```ts
await authClient.twoFactor.verifyTotp({
  code: "" // user input
})
```
```

<Callout>

You can skip verification by setting `skipVerificationOnEnable` to true in your plugin config.

</Callout>

#### Sign In with 2FA

When a user with 2FA enabled tries to sign in via email, the response will contain `twoFactorRedirect` set to `true`. This indicates that the user needs to verify their 2FA code.

```
```ts title="sign-in.ts"
await authClient.signIn.email({
  email: "user@example.com",
  password: "password123",
})
```
```

```
})
```,
```

You can handle this in the `onSuccess` callback or by providing a `onTwoFactorRedirect` callback in the plugin config.

```
```ts title="sign-in.ts"
import { createAuthClient } from "better-auth/client";
import { twoFactorClient } from "better-auth/client/plugins";

const authClient = createAuthClient({
  plugins: [twoFactorClient({
    onTwoFactorRedirect(){
      // Handle the 2FA verification globally
    }
  })]
})
```,
```

Or you can handle it in place:

```
```ts
await authClient.signIn.email({
  email: "user@example.com",
  password: "password123",
}, {
  async onSuccess(context) {
    if (context.data.twoFactorRedirect) {
      // Handle the 2FA verification in place
    }
  }
})
```,
```

#### #### Using `auth.api`

When you call `auth.api.signInEmail` on the server, and the user has 2FA enabled, it will, by default, respond with an object where `twoFactorRedirect` is set to `true`. This behavior isn't inferred in TypeScript, which can be misleading. We recommend passing `asResponse: true` to receive the Response object instead.

```
```ts
const response = await auth.api.signInEmail({
  email: "my-email@email.com",
  password: "secure-password",
  asResponse: true
})
```,
```

#### ### Disabling 2FA

To disable two-factor authentication, call `twoFactor.disable` with the user's password:

```
```ts title="two-factor.ts"
const { data } = await authClient.twoFactor.disable({
  password: "password" // user password required
})
```,
```

#### ### TOTP

TOTP (Time-Based One-Time Password) is an algorithm that generates a unique password for each login attempt using time as a counter. Every fixed interval (Better Auth defaults to 30 seconds), a new password is generated. This addresses several issues with traditional passwords: they can be forgotten, stolen, or guessed. OTPs solve some of these problems, but their delivery via SMS or email can be unreliable (or even risky, considering it opens new attack vectors).

TOTP, however, generates codes offline, making it both secure and convenient. You just need an authenticator app on your phone, and you're set—no internet required.

#### #### Getting TOTP URI

After enabling 2FA, you can get the TOTP URI to display to the user. This URI is generated by the server using the `secret` and `issuer` and can be used to generate a QR code for the user to scan with their authenticator app.

```
```ts
const { data, error } = await authClient.twoFactor.getTotpUri({
  password: "password" // user password required
})
```
```

#### \*\*Example: Using React\*\*

```
```tsx title="user-card.tsx"
import QRCode from "react-qr-code";

export default function UserCard(){
  const { data: session } = client.useSession();
  const { data: qr } = useQuery({
    queryKey: ["two-factor-qr"],
    queryFn: async () => {
      const res = await authClient.twoFactor.getTotpUri();
      return res.data;
    },
    enabled: !!session?.user.twoFactorEnabled,
  });
  return (
    <QRCode value={qr?.totpURI || ""} />
  )
}
```
```

#### <Callout>

By default the issuer for TOTP is set to the app name provided in the auth config or if not provided it will be set to `Better Auth`. You can override this by passing `issuer` to the plugin config.

#### </Callout>

#### #### Verifying TOTP

After the user has entered their 2FA code, you can verify it using `twoFactor.verifyTotp` method.

```
```ts
const verifyTotp = async (code: string) => {
  const { data, error } = await authClient.twoFactor.verifyTotp({ code })
}
```
```

#### ### OTP

OTP (One-Time Password) is similar to TOTP but a random code is generated and sent to the user's email or phone.

Before using OTP to verify the second factor, you need to configure `sendOTP` in your Better Auth instance. This function is responsible for sending the OTP to the user's email, phone, or any other method supported by your application.

```
```ts title="auth.ts"
import { betterAuth } from "better-auth"
import { twoFactor } from "better-auth/plugins"

export const auth = betterAuth({
  plugins: [
    twoFactor({
      otpOptions: {

```



```

        async sendOTP({ user, otp }, request) {
          // send otp to user
        },
      },
    ))
  ]
})
```

```

#### #### Sending OTP

Sending an OTP is done by calling the `twoFactor.sendOtp` function. This function will trigger your sendOTP implementation that you provided in the Better Auth configuration.

```

```ts
const { data, error } = await authClient.twoFactor.sendOtp()
if (data) {
  // redirect or show the user to enter the code
}
```

```

#### #### Verifying OTP

After the user has entered their OTP code, you can verify it

```

```ts
const verifyOtp = async (code: string) => {
  await authClient.twoFactor.verifyOtp({ code }, {
    onSuccess(){
      //redirect the user on success
    },
    onError(ctx){
      alert(ctx.error.message)
    }
  })
}
```

```

#### #### Backup Codes

Backup codes are generated and stored in the database. This can be used to recover access to the account if the user loses access to their phone or email.

#### #### Generating Backup Codes

Generate backup codes for account recovery:

```

```ts
const { data, error } = await authClient.twoFactor.generateBackupCodes({
  password: "password" // user password required
})
if (data) {
  // Show the backup codes to the user
}
```

```

#### #### Using Backup Codes

You can now allow users to provide backup code as account recover method.

```

```ts
await authClient.twoFactor.verifyBackupCode({code: ""}, {
  onSuccess(){
    //redirect the user on success
  },
}
```

```

```

    onError(ctx){
      alert(ctx.error.message)
    }
  })
  ``

```

once a backup code is used, it will be removed from the database and can't be used again.

#### #### Viewing Backup Codes

You can view the backup codes at any time by calling `viewBackupCodes`. This action can only be performed on the server using `auth.api`.

```

````ts
await auth.api.viewBackupCodes({
  body: {
    userId: "user-id"
  }
})
````

```

#### ### Trusted Devices

You can mark a device as trusted by passing `trustDevice` to `verifyTotp` or `verifyOtp`.

```

````ts
const verify2FA = async (code: string) => {
  const { data, error } = await authClient.twoFactor.verifyTotp({
    code,
    callbackURL: "/dashboard",
    trustDevice: true // Mark this device as trusted
  })
  if (data) {
    // 2FA verified and device trusted
  }
}
````

```

When `trustDevice` is set to `true`, the current device will be remembered for 60 days. During this period, the user won't be prompted for 2FA on subsequent sign-ins from this device. The trust period is refreshed each time the user signs in successfully.

#### ### Issuer

By adding an `issuer` you can set your application name for the 2fa application.

For example, if your user uses Google Auth, the default appName will show up as `Better Auth`. However, by using the following code, it will show up as `my-app-name`.

```

````ts
twoFactor({
  issuer: "my-app-name" // [!code highlight]
})
````

```

\*\*\*

#### ## Schema

The plugin requires 1 additional fields in the `user` table and 1 additional table to store the two factor authentication data.

```

<DatabaseTable
  fields={
    { name: "twoFactorEnabled", type: "boolean", description: "Whether two factor authentication is enabled for the user.",
      isOptional: true },

```

```
  }
/>
```

Table: `twoFactor`

```
<DatabaseTable
  fields=[
    { name: "id", type: "string", description: "The ID of the two factor authentication.", isPrimaryKey: true },
    { name: "userId", type: "string", description: "The ID of the user", isForeignKey: true },
    { name: "secret", type: "string", description: "The secret used to generate the TOTP code.", isOptional: true },
    { name: "backupCodes", type: "string", description: "The backup codes used to recover access to the account if the user loses access to their phone or email.", isOptional: true },
  ]
/>
```

## Options

### Server

**\*\*twoFactorTable\*\***: The name of the table that stores the two factor authentication data. Default: `twoFactor`.

**\*\*skipVerificationOnEnable\*\***: Skip the verification process before enabling two factor for a user.

**\*\*Issuer\*\***: The issuer is the name of your application. It's used to generate TOTP codes. It'll be displayed in the authenticator apps.

**\*\*TOTP options\*\***

these are options for TOTP.

```
<TypeTable
  type={{
    digits:{
      description: "The number of digits the otp to be",
      type: "number",
      default: 6,
    },
    period: {
      description: "The period for otp in seconds.",
      type: "number",
      default: 30,
    },
  }}
/>
```

**\*\*OTP options\*\***

these are options for OTP.

```
<TypeTable
  type={{
    sendOTP: {
      description: "a function that sends the otp to the user's email or phone number. It takes two parameters: user and otp",
      type: "function",
    },
    period: {
      description: "The period for otp in seconds.",
      type: "number",
      default: 30,
    },
  }}
/>
```

**\*\*Backup Code Options\*\***

backup codes are generated and stored in the database when the user enabled two factor authentication. This can be used to recover access to the account if the user loses access to their phone or email.

```
<TypeTable
  type={{
    amount: {
      description: "The amount of backup codes to generate",
      type: "number",
      default: 10,
    },
    length: {
      description: "The length of the backup codes",
      type: "number",
      default: 10,
    },
    customBackupCodesGenerate: {
      description: "A function that generates custom backup codes. It takes no parameters and returns an array of
strings.",
      type: "function",
    },
  }}
/>
```

### ### Client

To use the two factor plugin in the client, you need to add it on your plugins list.

```
```ts title="auth-client.ts"
import { createAuthClient } from "better-auth/client"
import { twoFactorClient } from "better-auth/client/plugins"

const authClient = createAuthClient({
  plugins: [
    twoFactorClient({ // [!code highlight]
      onTwoFactorRedirect(){
        window.location.href = "/2fa" // Handle the 2FA verification redirect
      }
    }) // [!code highlight]
  ]
})
```
```

### \*\*Options\*\*

`onTwoFactorRedirect`: A callback that will be called when the user needs to verify their 2FA code. This can be used to redirect the user to the 2FA page.

```
file: ./content/docs/plugins/admin.mdx
meta: {
  "title": "Admin",
  "description": "Admin plugin for Better Auth"
}
```

The Admin plugin provides a set of administrative functions for user management in your application. It allows administrators to perform various operations such as creating users, managing user roles, banning/unbanning users, impersonating users, and more.

### ## Installation

```
<Steps>
<Step>
  ### Add the plugin to your auth config
```

To use the Admin plugin, add it to your auth config.

```

```ts title="auth.ts"
import { betterAuth } from "better-auth"
import { admin } from "better-auth/plugins" // [!code highlight]

export const auth = betterAuth({
  // ... other config options
  plugins: [
    admin() // [!code highlight]
  ]
})
```

```

&lt;/Step&gt;

&lt;Step&gt;

### ### Migrate the database

Run the migration or generate the schema to add the necessary fields and tables to the database.

```

<Tabs items={["migrate", "generate"]}>
  <Tab value="migrate">
    ```bash
    npx @better-auth/cli migrate
    ```
  </Tab>

```

```

  <Tab value="generate">
    ```bash
    npx @better-auth/cli generate
    ```
  </Tab>
</Tabs>

```

See the [Schema](#schema) section to add the fields manually.

&lt;/Step&gt;

&lt;Step&gt;

### ### Add the client plugin

Next, include the admin client plugin in your authentication client instance.

```

```ts title="auth-client.ts"
import { createAuthClient } from "better-auth/client"
import { adminClient } from "better-auth/client/plugins"

export const authClient = createAuthClient({
  plugins: [
    adminClient()
  ]
})
```

```

&lt;/Step&gt;

&lt;/Steps&gt;

## ## Usage

Before performing any admin operations, the user must be authenticated with an admin account. An admin is any user assigned the `admin` role or any user whose ID is included in the `adminUserIds` option.

### ### Create User

Allows an admin to create a new user.

```

```ts title="admin.ts"
const newUser = await authClient.admin.createUser({

```

```

    name: "Test User",
    email: "test@example.com",
    password: "password123",
    role: "user", // this can also be an array for multiple roles (e.g. ["user", "sale"])
    data: {
      // any additional on the user table including plugin fields and custom fields
      customField: "customValue",
    },
  });
```,

```

### #### List Users

Allows an admin to list all users in the database.

```

```ts title="admin.ts"
const users = await authClient.admin.listUsers({
  query: {
    limit: 10,
  },
});
```,

```

By default, 100 users are returned. You can adjust the limit and offset using the following query parameters:

- \* `search`: The search query to apply to the users. It can be an object with the following properties:
- \* `field`: The field to search on, which can be `email` or `name`.
- \* `operator`: The operator to use for the search. It can be `contains`, `starts\_with`, or `ends\_with`.
- \* `value`: The value to search for.
- \* `limit`: The number of users to return.
- \* `offset`: The number of users to skip.
- \* `sortBy`: The field to sort the users by.
- \* `sortDirection`: The direction to sort the users by. Defaults to `asc`.
- \* `filter`: The filter to apply to the users. It can be an array of objects.

```

```ts title="admin.ts"
const users = await authClient.admin.listUsers({
  query: {
    searchField: "email",
    searchOperator: "contains",
    searchValue: "@example.com",
    limit: 10,
    offset: 0,
    sortBy: "createdAt",
    sortDirection: "desc",
    filterField: "role",
    filterOperator: "eq",
    filterValue: "admin"
  }
});
```,

```

### #### Query Filtering

The `listUsers` function supports various filter operators including `eq`, `contains`, `starts\_with`, and `ends\_with`.

### #### Pagination

The `listUsers` function supports pagination by returning metadata alongside the user list. The response includes the following fields:

```

```ts
{
  users: User[], // Array of returned users
  total: number, // Total number of users after filters and search queries
}

```

```

    limit: number | undefined, // The limit provided in the query
    offset: number | undefined // The offset provided in the query
  }
  ```

```

## ##### How to Implement Pagination

To paginate results, use the `total`, `limit`, and `offset` values to calculate:

```

* **Total pages:** `Math.ceil(total / limit)`
* **Current page:** `(offset / limit) + 1`
* **Next page offset:** `Math.min(offset + limit, (total - 1))` - The value to use as `offset` for the next page, ensuring it
does not exceed the total number of pages.
* **Previous page offset:** `Math.max(0, offset - limit)` - The value to use as `offset` for the previous page (ensuring it
doesn't go below zero).

```

## ##### Example Usage

Fetching the second page with 10 users per page:

```

```ts title="admin.ts"
const pageSize = 10;
const currentPage = 2;

const users = await authClient.admin.listUsers({
  query: {
    limit: pageSize,
    offset: (currentPage - 1) * pageSize
  }
});

const totalUsers = users.total;
const totalPages = Math.ceil(totalUsers / limit)
```

```

## ### Set User Role

Changes the role of a user.

```

```ts title="admin.ts"
const updatedUser = await authClient.admin.setRole({
  userId: "user_id_here",
  role: "admin", // this can also be an array for multiple roles (e.g. ["admin", "sale"])
});
```

```

## ### Ban User

Bans a user, preventing them from signing in and revokes all of their existing sessions.

```

```ts title="admin.ts"
const bannedUser = await authClient.admin.banUser({
  userId: "user_id_here",
  banReason: "Spamming", // Optional (if not provided, the default ban reason will be used - No reason)
  banExpiresIn: 60 * 60 * 24 * 7, // Optional (if not provided, the ban will never expire)
});
```

```

## ### Unban User

Removes the ban from a user, allowing them to sign in again.

```

```ts title="admin.ts"
const unbannedUser = await authClient.admin.unbanUser({
  userId: "user_id_here",

```

```
});
```

```

### #### List User Sessions

Lists all sessions for a user.

```
```ts title="admin.ts"
const sessions = await authClient.admin.listUserSessions({
  userId: "user_id_here",
});
```

```

### #### Revoke User Session

Revokes a specific session for a user.

```
```ts title="admin.ts"
const revokedSession = await authClient.admin.revokeUserSession({
  sessionToken: "session_token_here",
});
```

```

### #### Revoke All Sessions for a User

Revokes all sessions for a user.

```
```ts title="admin.ts"
const revokedSessions = await authClient.admin.revokeUserSessions({
  userId: "user_id_here",
});
```

```

### #### Impersonate User

This feature allows an admin to create a session that mimics the specified user. The session will remain active until either the browser session ends or it reaches 1 hour. You can change this duration by setting the `impersonationSessionDuration` option.

```
```ts title="admin.ts"
const impersonatedSession = await authClient.admin.impersonateUser({
  userId: "user_id_here",
});
```

```

### #### Stop Impersonating User

To stop impersonating a user and continue with the admin account, you can use `stopImpersonating`

```
```ts title="admin.ts"
await authClient.admin.stopImpersonating();
```

```

### #### Remove User

Hard deletes a user from the database.

```
```ts title="admin.ts"
const deletedUser = await authClient.admin.removeUser({
  userId: "user_id_here",
});
```

```

## ## Access Control



The admin plugin offers a highly flexible access control system, allowing you to manage user permissions based on their role. You can define custom permission sets to fit your needs.

### Roles

By default, there are two roles:

``admin``: Users with the admin role have full control over other users.

``user``: Users with the user role have no control over other users.

<Callout>

A user can have multiple roles. Multiple roles are stored as string separated by comma (",").

</Callout>

### Permissions

By default, there are two resources with up to six permissions.

**user:**

``create` `list` `set-role` `ban` `impersonate` `delete` `set-password``

**session:**

``list` `revoke` `delete``

Users with the admin role have full control over all the resources and actions. Users with the user role have no control over any of those actions.

### Custom Permissions

The plugin provides an easy way to define your own set of permissions for each role.

<Steps>

<Step>

#### Create Access Control

You first need to create an access controller by calling the ``createAccessControl`` function and passing the statement object. The statement object should have the resource name as the key and the array of actions as the value.

```
```ts title="permissions.ts"
import { createAccessControl } from "better-auth/plugins/access";

/**
 * make sure to use `as const` so typescript can infer the type correctly
 */
const statement = { // [!code highlight]
  project: ["create", "share", "update", "delete"], // [!code highlight]
} as const; // [!code highlight]

const ac = createAccessControl(statement); // [!code highlight]
```
```

</Step>

<Step>

#### Create Roles

Once you have created the access controller you can create roles with the permissions you have defined.

```
```ts title="permissions.ts"
import { createAccessControl } from "better-auth/plugins/access";

export const statement = {
  project: ["create", "share", "update", "delete"], // <-- Permissions available for created roles
} as const;
```

```
const ac = createAccessControl(statement);

export const user = ac.newRole({ // [!code highlight]
  project: ["create"], // [!code highlight]
}); // [!code highlight]

export const admin = ac.newRole({ // [!code highlight]
  project: ["create", "update"], // [!code highlight]
}); // [!code highlight]

export const myCustomRole = ac.newRole({ // [!code highlight]
  project: ["create", "update", "delete"], // [!code highlight]
  user: ["ban"], // [!code highlight]
}); // [!code highlight]
````
```

When you create custom roles for existing roles, the predefined permissions for those roles will be overridden. To add the existing permissions to the custom role, you need to import `defaultStatements` and merge it with your new statement, plus merge the roles' permissions set with the default roles.

```
````ts title="permissions.ts"
import { createAccessControl } from "better-auth/plugins/access";
import { defaultStatements, adminAc } from "better-auth/plugins/admin/access";

const statement = {
  ...defaultStatements, // [!code highlight]
  project: ["create", "share", "update", "delete"],
} as const;

const ac = createAccessControl(statement);

const admin = ac.newRole({
  project: ["create", "update"],
  ...adminAc.statements, // [!code highlight]
});
````
```

</Step>

<Step>

#### Pass Roles to the Plugin

Once you have created the roles you can pass them to the admin plugin both on the client and the server.

```
````ts title="auth.ts"
import { betterAuth } from "better-auth"
import { admin as adminPlugin } from "better-auth/plugins"
import { ac, admin, user } from "@auth/permissions"

export const auth = betterAuth({
  plugins: [
    adminPlugin({
      ac,
      roles: {
        admin,
        user,
        myCustomRole
      }
    })
  ],
});
````
```

You also need to pass the access controller and the roles to the client plugin.

```
````ts title="auth-client.ts"
```

```
import { createAuthClient } from "better-auth/client"
import { adminClient } from "better-auth/client/plugins"
import { ac, admin, user, myCustomRole } from "@auth/permissions"

export const client = createAuthClient({
  plugins: [
    adminClient({
      ac,
      roles: {
        admin,
        user,
        myCustomRole
      }
    })
  ]
})
```,
</Step>
</Steps>
```

### ### Access Control Usage

#### **\*\*Has Permission\*\*:**

To check a user's permissions, you can use the `hasPermission` function provided by the client.

```
```ts title="auth-client.ts"
const canCreateProject = await authClient.admin.hasPermission({
  permissions: {
    project: ["create"],
  },
});

// You can also check multiple resource permissions at the same time
const canCreateProjectAndCreateSale = await authClient.admin.hasPermission({
  permissions: {
    project: ["create"],
    sale: ["create"]
  },
});
```
```

If you want to check a user's permissions server-side, you can use the `userHasPermission` action provided by the `api` to check the user's permissions.

```
```ts title="api.ts"
import { auth } from "@auth";

await auth.api.userHasPermission({
  body: {
    userId: 'id', //the user id
    permissions: {
      project: ["create"], // This must match the structure in your access control
    },
  },
});

// You can also just pass the role directly
await auth.api.userHasPermission({
  body: {
    role: "admin",
    permissions: {
      project: ["create"], // This must match the structure in your access control
    },
  },
});
```

```
});

// You can also check multiple resource permissions at the same time
await auth.api.userHasPermission({
  body: {
    role: "admin",
    permissions: {
      project: ["create"], // This must match the structure in your access control
      sale: ["create"]
    },
  },
});
```,
```

### **\*\*Check Role Permission\*\*:**

Use the `checkRolePermission` function on the client side to verify whether a given **\*\*role\*\*** has a specific **\*\*permission\*\***. This is helpful after defining roles and their permissions, as it allows you to perform permission checks without needing to contact the server.

Note that this function does **\*\*not\*\*** check the permissions of the currently logged-in user directly. Instead, it checks what permissions are assigned to a specified role. The function is synchronous, so you don't need to use `await` when calling it.

```
```ts title="auth-client.ts"
const canCreateProject = authClient.admin.checkRolePermission({
  permissions: {
    user: ["delete"],
  },
  role: "admin",
});

// You can also check multiple resource permissions at the same time
const canDeleteUserAndRevokeSession = authClient.admin.checkRolePermission({
  permissions: {
    user: ["delete"],
    session: ["revoke"]
  },
  role: "admin",
});
```,
```

### **## Schema**

This plugin adds the following fields to the `user` table:

```
<DatabaseTable
  fields=[
    {
      name: "role",
      type: "string",
      description:
        "The user's role. Defaults to `user`. Admins will have the `admin` role.",
      isOptional: true,
    },
    {
      name: "banned",
      type: "boolean",
      description: "Indicates whether the user is banned.",
      isOptional: true,
    },
    {
      name: "banReason",
      type: "string",
      description: "The reason for the user's ban.",
      isOptional: true,
```

```

    },
    {
      name: "banExpires",
      type: "date",
      description: "The date when the user's ban will expire.",
      isOptional: true,
    },
  ]
}

```

And adds one field in the `session` table:

```

<DatabaseTable
  fields={[
    {
      name: "impersonatedBy",
      type: "string",
      description: "The ID of the admin that is impersonating this session.",
      isOptional: true,
    },
  ]
}

```

### ## Options

#### #### Default Role

The default role for a user. Defaults to `user`.

```

```ts title="auth.ts"
admin({
  defaultRole: "regular",
});
```

```

#### #### Admin Roles

The roles that are considered admin roles. Defaults to `["admin"]`.

```

```ts title="auth.ts"
admin({
  adminRoles: ["admin", "superadmin"],
});
```

```

<Callout type="warning">

Any role that isn't in the `adminRoles` list, even if they have the permission, will not be considered an admin.

</Callout>

#### #### Admin userIds

You can pass an array of userIds that should be considered as admin. Default to `[]`

```

```ts title="auth.ts"
admin({
  adminUserIds: ["user_id_1", "user_id_2"]
})
```

```

If a user is in the `adminUserIds` list, they will be able to perform any admin operation.

#### #### impersonationSessionDuration

The duration of the impersonation session in seconds. Defaults to 1 hour.

```
```ts title="auth.ts"
admin({
  impersonationSessionDuration: 60 * 60 * 24, // 1 day
});
```
```

### #### Default Ban Reason

The default ban reason for a user created by the admin. Defaults to `No reason`.

```
```ts title="auth.ts"
admin({
  defaultBanReason: "Spamming",
});
```
```

### #### Default Ban Expires In

The default ban expires in for a user created by the admin in seconds. Defaults to `undefined` (meaning the ban never expires).

```
```ts title="auth.ts"
admin({
  defaultBanExpiresIn: 60 * 60 * 24, // 1 day
});
```
```

### #### bannedUserMessage

The message to show when a banned user tries to sign in. Defaults to "You have been banned from this application. Please contact support if you believe this is an error."

```
```ts title="auth.ts"
admin({
  bannedUserMessage: "Custom banned user message",
});
```
```

```
file: ./content/docs/plugins/anonymous.mdx
meta: {
  "title": "Anonymous",
  "description": "Anonymous plugin for Better Auth."
}
```

The Anonymous plugin allows users to have an authenticated experience without requiring them to provide an email address, password, OAuth provider, or any other Personally Identifiable Information (PII). Users can later link an authentication method to their account when ready.

## ## Installation

### <Steps>

#### <Step>

#### #### Add the plugin to your auth config

To enable anonymous authentication, add the anonymous plugin to your authentication configuration.

```
```ts title="auth.ts"
import { betterAuth } from "better-auth"
import { anonymous } from "better-auth/plugins" // [!code highlight]

export const auth = betterAuth({
  // ... other config options
  plugins: [
```

```

    anonymous() // [!code highlight]
  ]
})
```
</Step>

```

```

<Step>
  ### Migrate the database

```

Run the migration or generate the schema to add the necessary fields and tables to the database.

```

<Tabs items={["migrate", "generate"]} >
  <Tab value="migrate">
    ``` bash
    npx @better-auth/cli migrate
    ```
  </Tab>

  <Tab value="generate">
    ``` bash
    npx @better-auth/cli generate
    ```
  </Tab>
</Tabs>

```

See the [Schema](#schema) section to add the fields manually.

</Step>

```

<Step>
  ### Add the client plugin

```

Next, include the anonymous client plugin in your authentication client instance.

```

``` ts title="auth-client.ts"
import { createAuthClient } from "better-auth/client"
import { anonymousClient } from "better-auth/client/plugins"

export const authClient = createAuthClient({
  plugins: [
    anonymousClient()
  ]
})
```
</Step>
</Steps>

```

## ## Usage

### ### Sign In

To sign in a user anonymously, use the `signIn.anonymous()` method.

```

``` ts title="example.ts"
const user = await authClient.signIn.anonymous()
```

```

### ### Link Account

If a user is already signed in anonymously and tries to `signIn` or `signUp` with another method, their anonymous activities can be linked to the new account.

To do that you first need to provide `onLinkAccount` callback to the plugin.

```

``` ts title="auth.ts"
import { betterAuth } from "better-auth"

```

```
export const auth = betterAuth({
  plugins: [
    anonymous({
      onLinkAccount: async ({ anonymousUser, newUser }) => {
        // perform actions like moving the cart items from anonymous user to the new user
      }
    })
  ]
})
````
```

Then when you call `signIn` or `signUp` with another method, the `onLinkAccount` callback will be called. And the `anonymousUser` will be deleted by default.

```
````ts title="example.ts"
const user = await authClient.signIn.email({
  email,
})
````
```

### ### Options

\* `emailDomainName`: The domain name to use when generating an email address for anonymous users. Defaults to the domain name of the current site.

```
````ts title="auth.ts"
import { betterAuth } from "better-auth"

export const auth = betterAuth({
  plugins: [
    anonymous({
      emailDomainName: "example.com"
    })
  ]
})
````
```

\* `onLinkAccount`: A callback function that is called when an anonymous user links their account to a new authentication method. The callback receives an object with the `anonymousUser` and the `newUser`.

\* `disableDeleteAnonymousUser`: By default, the anonymous user is deleted when the account is linked to a new authentication method. Set this option to `true` to disable this behavior.

\* `generateName`: A callback function that is called to generate a name for the anonymous user. Useful if you want to have random names for anonymous users, or if `name` is unique in your database.

### ### Schema

The anonymous plugin requires an additional field in the user table:

```
<DatabaseTable
  fields={[
    { name: "isAnonymous", type: "boolean", description: "Indicates whether the user is anonymous.", isOptional: true },
  ]}
/>
```

```
file: ./content/docs/plugins/api-key.mdx
meta: {
  "title": "API Key",
  "description": "API Key plugin for Better Auth."
}
```

The API Key plugin allows you to create and manage API keys for your application. It provides a way to authenticate and authorize API requests by verifying API keys.



## ## Features

- \* Create, manage, and verify API keys
- \* [Built-in rate limiting](/docs/plugins/api-key#rate-limiting)
- \* [Custom expiration times, remaining count, and refill systems](/docs/plugins/api-key#remaining-refill-and-expiration)
- \* [Metadata for API keys](/docs/plugins/api-key#metadata)
- \* Custom prefix
- \* [Sessions from API keys](/docs/plugins/api-key#sessions-from-api-keys)

## ## Installation

### <Steps>

#### <Step>

#### Add Plugin to the server

```
```ts title="auth.ts"
import { betterAuth } from "better-auth"
import { apiKey } from "better-auth/plugins"

export const auth = betterAuth({
  plugins: [ // [!code highlight]
    apiKey() // [!code highlight]
  ] // [!code highlight]
})
```
```

#### </Step>

#### <Step>

#### Migrate the database

Run the migration or generate the schema to add the necessary fields and tables to the database.

```
<Tabs items=["migrate", "generate"]>
```

```
<Tab value="migrate">
```

```
```bash
npx @better-auth/cli migrate
```
```

```
</Tab>
```

```
<Tab value="generate">
```

```
```bash
npx @better-auth/cli generate
```
```

```
</Tab>
```

```
</Tabs>
```

See the [Schema](#schema) section to add the fields manually.

#### </Step>

#### <Step>

#### Add the client plugin

```
```ts title="auth-client.ts"
import { createAuthClient } from "better-auth/client"
import { apiKeyClient } from "better-auth/client/plugins"

export const authClient = createAuthClient({
  plugins: [ // [!code highlight]
    apiKeyClient() // [!code highlight]
  ] // [!code highlight]
})
```
```

#### </Step>

### </Steps>

## ## Usage

You can view the list of API Key plugin options [here](/docs/plugins/api-key#api-key-plugin-options).

### #### Create an API key

<Endpoint path="/api-key/create" method="POST" />

<Tabs items={['Client', 'Server']>

<Tab value="Client">

```ts

```
const { data: apiKey, error } = await authClient.apiKey.create({
  name: "My API Key",
  expiresIn: 60 * 60 * 24 * 7, // 7 days
  prefix: "my_app",
  metadata: {
    tier: "premium",
  },
});
```

</Tab>

<Tab value="Server">

on the server, you can create an API key for a user by passing the `userId` property in the body. And allows you to add any properties you want to the API key.

```ts

```
const apiKey = await auth.api.createApiKey({
  body: {
    name: "My API Key",
    expiresIn: 60 * 60 * 24 * 365, // 1 year
    prefix: "my_app",
    remaining: 100,
    refillAmount: 100,
    refillInterval: 60 * 60 * 24 * 7, // 7 days
    metadata: {
      tier: "premium",
    },
    rateLimitTimeWindow: 1000 * 60 * 60 * 24, // everyday
    rateLimitMax: 100, // every day, they can use up to 100 requests
    rateLimitEnabled: true,
    userId: user.id, // the user ID to create the API key for
  },
});
```

</Tab>

</Tabs>

All API keys are assigned to a user. If you're creating an API key on the server, without access to headers, you must pass the `userId` property. This is the ID of the user that the API key is associated with.

### #### Properties

All properties are optional. However if you pass a `refillAmount`, you must also pass a `refillInterval`, and vice versa.

\* `name`?: The name of the API key.

\* `expiresIn`?: The expiration time of the API key in seconds. If not provided, the API key will never expire.

\* `prefix`?: The prefix of the API key. This is used to identify the API key in the database.

\* `metadata`?: The metadata of the API key. This is used to store additional information about the API key.

<DividerText>Server Only Properties</DividerText>

\* `remaining`?: The remaining number of requests for the API key. If `null`, then there is no cap to key usage.

\* `refillAmount`?: The amount to refill the `remaining` count of the API key.

- \* ``refillInterval``: The interval to refill the API key in milliseconds.
- \* ``rateLimitTimeWindow``: The duration in milliseconds where each request is counted. Once the ``rateLimitMax`` is reached, the request will be rejected until the ``timeWindow`` has passed, at which point the time window will be reset.
- \* ``rateLimitMax``: The maximum number of requests allowed within the ``rateLimitTimeWindow``.
- \* ``rateLimitEnabled``: Whether rate limiting is enabled for the API key.
- \* ``permissions``: Permissions for the API key, structured as a record mapping resource types to arrays of allowed actions.

```
```ts
const example = {
  projects: ["read", "read-write"],
};
```
```

\* ``userId``: The ID of the user associated with the API key. When creating an API Key, you must pass the headers of the user who will own the key. However if you do not have the headers, you can pass this field, which will allow you to bypass the need for headers.

#### #### Result

It'll return the ``ApiKey`` object which includes the ``key`` value for you to use. Otherwise if it throws, it will throw an ``APIError``.

\*\*\*

#### ### Verify an API key

<Endpoint path="/api-key/verify" method="POST" isServerOnly />

```
```ts
const { valid, error, key } = await auth.api.verifyApiKey({
  body: {
    key: "your_api_key_here",
  },
});
```
```

```
//with permissions check
const { valid, error, key } = await auth.api.verifyApiKey({
  body: {
    key: "your_api_key_here",
    permissions: {
      projects: ["read", "read-write"],
    },
  },
});
```
```

#### #### Properties

- \* ``key``: The API Key to validate
- \* ``permissions``: The permissions to check against the API key.

#### #### Result

```
```ts
type Result = {
  valid: boolean;
  error: { message: string; code: string } | null;
  key: Omit<ApiKey, "key"> | null;
};
```
```

\*\*\*

#### ### Get an API key

```
<Endpoint method="GET" path="/api-key/get" isServerOnly />
```

```
```ts
const key = await auth.api.getApiKey({
  body: {
    keyId: "your_api_key_id_here",
  },
});
```
```

#### #### Properties

\* `keyId`: The API key ID to get information on.

#### #### Result

You'll receive everything about the API key details, except for the `key` value itself. If it fails, it will throw an `APIError`.

```
```ts
type Result = Omit<ApiKey, "key">;
```
```

\*\*\*

#### ### Update an API key

```
<Endpoint method="POST" path="/api-key/update" isServerOnly />
```

```
<Tabs items={['Client', 'Server']}>
<Tab value="Client">
  ```ts
  const { data: apiKey, error } = await authClient.apiKey.update({
    keyId: "your_api_key_id_here",
    name: "New API Key Name",
    enabled: false,
  });
  ```
```

```
</Tab>
```

```
<Tab value="Server">
```

You can update an API key on the server by passing the `keyId` and any other properties you want to update.

```
```ts
const apiKey = await auth.api.updateApiKey({
  body: {
    keyId: "your_api_key_id_here",
    name: "New API Key Name",
    userId: "userId",
    enabled: false,
    remaining: 100,
    refillAmount: null,
    refillInterval: null,
    metadata: null,
    expiresIn: 60 * 60 * 24 * 7,
    rateLimitEnabled: false,
    rateLimitTimeWindow: 1000 * 60 * 60 * 24,
    rateLimitMax: 100,
  },
});
```
```

```
</Tab>
```

```
</Tabs>
```

#### #### Properties

<DividerText>Client</DividerText>- `keyId`: The API key ID to update on. -  
 `name`: Update the key name.

<DividerText>Server Only</DividerText>- `userId`: Update the user ID who owns this key. - `name`: Update the key name. - `enabled`: Update whether the API key is enabled or not. - `remaining`: Update the remaining count. - `refillAmount`: Update the amount to refill the `remaining` count every interval. - `refillInterval`: Update the interval to refill the `remaining` count. - `metadata`: Update the metadata of the API key. - `expiresIn`: Update the expiration time of the API key. In seconds. - `rateLimitEnabled`: Update whether the rate-limiter is enabled or not. - `rateLimitTimeWindow`: Update the time window for the rate-limiter. - `rateLimitMax`: Update the maximum number of requests they can make during the rate-limit-time-window.

#### #### Result

If fails, throws `APIError`.

Otherwise, you'll receive the API Key details, except for the `key` value itself.

\*\*\*

#### ### Delete an API Key

<Endpoint method="POST" path="/api-key/delete" />

```
<Tabs items={['Client', 'Server']}>
  <Tab value="Client">
    `` `ts
    const { data: result, error } = await authClient.apiKey.delete({
      keyId: "your_api_key_id_here",
    });
    `` `
  </Tab>

  <Tab value="Server">
    `` `ts
    const apiKey = await auth.api.deleteApiKey({
      body: {
        keyId: "your_api_key_id_here",
        userId: "userId",
      },
    });
    `` `
  </Tab>
</Tabs>
```

#### #### Properties

\* `keyId`: The API key ID to delete.

#### #### Result

If fails, throws `APIError`.

Otherwise, you'll receive:

```
`` `ts
type Result = {
  success: boolean;
};
`` `
```

\*\*\*

#### ### List API keys

```

<Endpoint method="GET" path="/api-key/list" />

<Tabs items={['Client', 'Server']}>
  <Tab value="Client">
    `` `ts
    const { data: apiKeys, error } = await authClient.apiKey.list();
    `` `
  </Tab>

  <Tab value="Server">
    `` `ts
    const apiKeys = await auth.api.listApiKeys({
      headers: user_headers,
    });
    `` `
  </Tab>
</Tabs>

```

#### #### Result

If fails, throws `APIError`.  
Otherwise, you'll receive:

```

`` `ts
type Result = ApiKey[];
`` `

```

\*\*\*

#### ### Delete all expired API keys

This function will delete all API keys that have an expired expiration date.

```

<Endpoint method="DELETE" path="/api-key/delete-all-expired-api-keys" isServerOnly />

```

```

` ts await auth.api.deleteAllExpiredApiKeys(); `

```

<Callout>

We automatically delete expired API keys every time any apiKey plugin endpoints were called, however they are rate-limited to a 10 second cool down each call to prevent multiple calls to the database.

</Callout>

\*\*\*

#### ## Sessions from API keys

Any time an endpoint in Better Auth is called that has a valid API key in the headers, we will automatically create a mock session to represent the user.

The default header key is `x-api-key`, but this can be changed by setting the `apiKeyHeaders` option in the plugin options.

```

`` `ts
export const auth = betterAuth({
  plugins: [
    apiKey({
      apiKeyHeaders: ["x-api-key", "xyz-api-key"], // or you can pass just a string, eg: "x-api-key"
    }),
  ],
});
`` `

```

Or optionally, you can pass an `apiKeyGetter` function to the plugin options, which will be called with the `GenericEndpointContext`, and from there, you should return the API key, or `null` if the request is invalid.

```

```ts
export const auth = betterAuth({
  plugins: [
    apiKey({
      apiKeyGetter: (ctx) => {
        const has = ctx.request.headers.has("x-api-key");
        if (!has) return null;
        return ctx.request.headers.get("x-api-key");
      },
    }),
  ],
});
```

```

### ### Rate Limiting

Every API key can have its own rate limit settings, however, the built-in rate-limiting only applies to the verification process for a given API key.

For every other endpoint/method, you should utilize Better Auth's [built-in rate-limiting](/docs/concepts/rate-limit).

You can refer to the rate-limit default configurations below in the API Key plugin options.

An example default value:

```

```ts
export const auth = betterAuth({
  plugins: [
    apiKey({
      rateLimit: {
        enabled: true,
        timeWindow: 1000 * 60 * 60 * 24, // 1 day
        maxRequests: 10, // 10 requests per day
      },
    }),
  ],
});
```

```

For each API key, you can customize the rate-limit options on create.

<Callout>

You can only customize the rate-limit options on the server auth instance.

</Callout>

```

```ts
const apiKey = await auth.api.createApiKey({
  body: {
    rateLimitEnabled: true,
    rateLimitTimeWindow: 1000 * 60 * 60 * 24, // 1 day
    rateLimitMax: 10, // 10 requests per day
  },
  headers: user_headers,
});
```

```

### #### How does it work?

For each request, a counter (internally called `requestCount`) is incremented.\

If the `rateLimitMax` is reached, the request will be rejected until the `timeWindow` has passed, at which point the `timeWindow` will be reset.

### ### Remaining, refill, and expiration

The remaining count is the number of requests left before the API key is disabled.\

The refill interval is the interval in milliseconds where the `remaining` count is refilled by day.\n  
The expiration time is the expiration date of the API key.

#### How does it work?

##### Remaining:

Whenever an API key is used, the `remaining` count is updated.\n  
If the `remaining` count is `null`, then there is no cap to key usage.\n  
Otherwise, the `remaining` count is decremented by 1.\n  
If the `remaining` count is 0, then the API key is disabled & removed.

##### refillInterval & refillAmount:

Whenever an API key is created, the `refillInterval` and `refillAmount` are set to `null`.\n  
This means that the API key will not be refilled automatically.\n  
However, if `refillInterval` & `refillAmount` are set, then the API key will be refilled accordingly.

##### Expiration:

Whenever an API key is created, the `expiresAt` is set to `null`.\n  
This means that the API key will never expire.\n  
However, if the `expiresIn` is set, then the API key will expire after the `expiresIn` time.

## Custom Key generation & verification

You can customize the key generation and verification process straight from the plugin options.

Here's an example:

```
```ts
export const auth = betterAuth({
  plugins: [
    apiKey({
      customKeyGenerator: (options: {
        length: number;
        prefix: string | undefined;
      }) => {
        const apiKey = mySuperSecretApiKeyGenerator(
          options.length,
          options.prefix
        );
        return apiKey;
      },
      customAPIKeyValidator: ({ ctx, key }) => {
        if (key.endsWith("_super_secret_api_key")) {
          return true;
        } else {
          return false;
        }
      },
    }),
  ],
});
```
```

<Callout>

If you're **not** using the `length` property provided by `customKeyGenerator`, you **must** set the `defaultKeyLength` property to how long generated keys will be.

```
```ts
export const auth = betterAuth({
  plugins: [
    apiKey({
      customKeyGenerator: () => {
```



```

    return crypto.randomUUID();
  },
  defaultKeyLength: 36 // Or whatever the length is
})
]
});
```
</Callout>

```

If an API key is validated from your `customAPIKeyValidator`, we still must match that against the database's key. However, by providing this custom function, you can improve the performance of the API key verification process, as all failed keys can be invalidated without having to query your database.

### ### Metadata

We allow you to store metadata alongside your API keys. This is useful for storing information about the key, such as a subscription plan for example.

To store metadata, make sure you haven't disabled the metadata feature in the plugin options.

```

```ts
export const auth = betterAuth({
  plugins: [
    apiKey({
      enableMetadata: true,
    }),
  ],
});
```

```

Then, you can store metadata in the `metadata` field of the API key object.

```

```ts
const apiKey = await auth.api.createApiKey({
  body: {
    metadata: {
      plan: "premium",
    },
  },
});
```

```

You can then retrieve the metadata from the API key object.

```

```ts
const apiKey = await auth.api.getApiKey({
  body: {
    keyId: "your_api_key_id_here",
  },
});

console.log(apiKey.metadata.plan); // "premium"
```

```

### ### API Key plugin options

`apiKeyHeaders` <span className="opacity-70">`string | string[]`</span>

The header name to check for API key. Default is `x-api-key`.

`customAPIKeyGetter` <span className="opacity-70">`(ctx: GenericEndpointContext) => string | null`</span>

A custom function to get the API key from the context.

`customAPIKeyValidator` <span className="opacity-70">`(options: { ctx: GenericEndpointContext; key: string; }) =>

boolean`</span>

A custom function to validate the API key.

`customKeyGenerator` <span className="opacity-70">` (options: { length: number; prefix: string | undefined; }) => string | Promise<string>`</span>

A custom function to generate the API key.

`startingCharactersConfig` <span className="opacity-70">{ shouldStore?: boolean; charactersLength?: number; }`</span>

Customize the starting characters configuration.

<Accordions>

<Accordion title="startingCharactersConfig Options">

`shouldStore` <span className="opacity-70">` boolean`</span>

Whether to store the starting characters in the database.

If false, we will set `start` to `null`.

Default is `true`.

`charactersLength` <span className="opacity-70">` number`</span>

The length of the starting characters to store in the database.

This includes the prefix length.

Default is `6`.

</Accordion>

</Accordions>

`defaultKeyLength` <span className="opacity-70">` number`</span>

The length of the API key. Longer is better. Default is 64. (Doesn't include the prefix length)

`defaultPrefix` <span className="opacity-70">` string`</span>

The prefix of the API key.

Note: We recommend you append an underscore to the prefix to make the prefix more identifiable. (eg `hello\_`)

`maximumPrefixLength` <span className="opacity-70">` number`</span>

The maximum length of the prefix.

`minimumPrefixLength` <span className="opacity-70">` number`</span>

The minimum length of the prefix.

`maximumNameLength` <span className="opacity-70">` number`</span>

The maximum length of the name.

`minimumNameLength` <span className="opacity-70">` number`</span>

The minimum length of the name.

`enableMetadata` <span className="opacity-70">` boolean`</span>

Whether to enable metadata for an API key.

`keyExpiration` <span className="opacity-70">{ defaultExpiresIn?: number | null; disableCustomExpiresTime?: boolean; minExpiresIn?: number; maxExpiresIn?: number; }`</span>

Customize the key expiration.

<Accordions>

```
<Accordion title="keyExpiration options">
  `defaultExpiresIn` <span className="opacity-70">` number | null` </span>
```

The default expires time in milliseconds.

If `null`, then there will be no expiration time.

Default is `null`.

```
`disableCustomExpiresTime` <span className="opacity-70">` boolean` </span>
```

Whether to disable the expires time passed from the client.

If `true`, the expires time will be based on the default values.

Default is `false`.

```
`minExpiresIn` <span className="opacity-70">` number` </span>
```

The minimum expiresIn value allowed to be set from the client. in days.

Default is `1`.

```
`maxExpiresIn` <span className="opacity-70">` number` </span>
```

The maximum expiresIn value allowed to be set from the client. in days.

Default is `365`.

```
</Accordion>
```

```
</Accordions>
```

```
`rateLimit` <span className="opacity-70">` { enabled?: boolean; timeWindow?: number; maxRequests?: number; }` </span>
```

Customize the rate-limiting.

```
<Accordions>
```

```
<Accordion title="rateLimit options">
```

```
`enabled` <span className="opacity-70">` boolean` </span>
```

Whether to enable rate limiting. (Default true)

```
`timeWindow` <span className="opacity-70">` number` </span>
```

The duration in milliseconds where each request is counted.

Once the `maxRequests` is reached, the request will be rejected until the `timeWindow` has passed, at which point the `timeWindow` will be reset.

```
`maxRequests` <span className="opacity-70">` number` </span>
```

Maximum amount of requests allowed within a window.

Once the `maxRequests` is reached, the request will be rejected until the `timeWindow` has passed, at which point the `timeWindow` will be reset.

```
</Accordion>
```

```
</Accordions>
```

```
`schema` <span className="opacity-70">` InferOptionSchema<ReturnType<typeof apiKeySchema>>` </span>
```

Custom schema for the API key plugin.

```
`disableSessionForAPIKeys` <span className="opacity-70">` boolean` </span>
```

An API Key can represent a valid session, so we automatically mock a session for the user if we find a valid API key in the request headers.

```
`permissions` <span className="opacity-70">` { defaultPermissions?: Statements | ((userId: string, ctx:
GenericEndpointContext) => Statements | Promise<Statements>)} ` </span>
```

Permissions for the API key.

Read more about permissions [here](/docs/plugins/api-key#permissions).

```
<Accordions>
  <Accordion title="permissions Options">
    `defaultPermissions` <span className="opacity-70">` Statements | ((userId: string, ctx: GenericEndpointContext) =>
Statements | Promise<Statements>)` </span>
```

The default permissions for the API key.

```
</Accordion>
</Accordions>
```

```
`disableKeyHashing` <span className="opacity-70">` boolean` </span>
```

Disable hashing of the API key.

⚠ Security Warning: It's strongly recommended to not disable hashing.  
Storing API keys in plaintext makes them vulnerable to database breaches, potentially exposing all your users' API keys.

\*\*\*

## ## Schema

Table: `apiKey`

```
<DatabaseTable
  fields=[
    {
      name: "id",
      type: "string",
      description: "The ID of the API key.",
      isUnique: true,
      isPrimaryKey: true,
    },
    {
      name: "name",
      type: "string",
      description: "The name of the API key.",
      isOptional: true,
    },
    {
      name: "start",
      type: "string",
      description:
        "The starting characters of the API key. Useful for showing the first few characters of the API key in the UI for the
users to easily identify.",
      isOptional: true,
    },
    {
      name: "prefix",
      type: "string",
      description: "The API Key prefix. Stored as plain text.",
      isOptional: true,
    },
    {
      name: "key",
      type: "string",
      description: "The hashed API key itself.",
    },
    {
      name: "userId",
      type: "string",
      description: "The ID of the user who created the API key.",
      isForeignKey: true,
    },
    {
      name: "refillInterval",
      type: "number",
```

```
description: "The interval to refill the key in milliseconds.",
isOptional: true,
},
{
  name: "refillAmount",
  type: "number",
  description: "The amount to refill the remaining count of the key.",
  isOptional: true,
},
{
  name: "lastRefillAt",
  type: "Date",
  description: "The date and time when the key was last refilled.",
  isOptional: true,
},
{
  name: "enabled",
  type: "boolean",
  description: "Whether the API key is enabled.",
},
{
  name: "rateLimitEnabled",
  type: "boolean",
  description: "Whether the API key has rate limiting enabled.",
},
{
  name: "rateLimitTimeWindow",
  type: "number",
  description: "The time window in milliseconds for the rate limit.",
  isOptional: true,
},
{
  name: "rateLimitMax",
  type: "number",
  description:
    "The maximum number of requests allowed within the `rateLimitTimeWindow`.",
  isOptional: true,
},
{
  name: "requestCount",
  type: "number",
  description:
    "The number of requests made within the rate limit time window.",
},
{
  name: "remaining",
  type: "number",
  description: "The number of requests remaining.",
  isOptional: true,
},
{
  name: "lastRequest",
  type: "Date",
  description: "The date and time of the last request made to the key.",
  isOptional: true,
},
{
  name: "expiresAt",
  type: "Date",
  description: "The date and time when the key will expire.",
  isOptional: true,
},
{
  name: "createdAt",
  type: "Date",
```

```

    description: "The date and time the API key was created.",
  },
  {
    name: "updatedAt",
    type: "Date",
    description: "The date and time the API key was updated.",
  },
  {
    name: "permissions",
    type: "string",
    description: "The permissions of the key.",
    isOptional: true,
  },
  {
    name: "metadata",
    type: "Object",
    isOptional: true,
    description: "Any additional metadata you want to store with the key.",
  },
]}
/>

```

### ### Permissions

API keys can have permissions associated with them, allowing you to control access at a granular level. Permissions are structured as a record of resource types to arrays of allowed actions.

#### #### Setting Default Permissions

You can configure default permissions that will be applied to all newly created API keys:

```

```ts
export const auth = betterAuth({
  plugins: [
    apiKey({
      permissions: {
        defaultPermissions: {
          files: ["read"],
          users: ["read"],
        },
      },
    }),
  ],
});
```

```

You can also provide a function that returns permissions dynamically:

```

```ts
export const auth = betterAuth({
  plugins: [
    apiKey({
      permissions: {
        defaultPermissions: async (userId, ctx) => {
          // Fetch user role or other data to determine permissions
          return {
            files: ["read"],
            users: ["read"],
          };
        },
      },
    }),
  ],
});
```

```

### ### Creating API Keys with Permissions

When creating an API key, you can specify custom permissions:

```
```ts
const apiKey = await auth.api.createApiKey({
  body: {
    name: "My API Key",
    permissions: {
      files: ["read", "write"],
      users: ["read"],
    },
    userId: "userId",
  },
});
```
```

### ### Verifying API Keys with Required Permissions

When verifying an API key, you can check if it has the required permissions:

```
```ts
const result = await auth.api.verifyApiKey({
  body: {
    key: "your_api_key_here",
    permissions: {
      files: ["read"],
    },
  },
});

if (result.valid) {
  // API key is valid and has the required permissions
} else {
  // API key is invalid or doesn't have the required permissions
}
```
```

### ### Updating API Key Permissions

You can update the permissions of an existing API key:

```
```ts
const apiKey = await auth.api.updateApiKey({
  body: {
    keyId: existingApiKeyId,
    permissions: {
      files: ["read", "write", "delete"],
      users: ["read", "write"],
    },
  },
  headers: user_headers,
});
```
```

### ### Permissions Structure

Permissions follow a resource-based structure:

```
```ts
type Permissions = {
  [resourceType: string]: string[];
};
```
```

```
// Example:
const permissions = {
  files: ["read", "write", "delete"],
  users: ["read"],
  projects: ["read", "write"],
};
```,
```

When verifying an API key, all required permissions must be present in the API key's permissions for validation to succeed.

```
file: ./content/docs/plugins/bearer.mdx
meta: {
  "title": "Bearer Token Authentication",
  "description": "Authenticate API requests using Bearer tokens instead of browser cookies"
}
```

The Bearer plugin enables authentication using Bearer tokens as an alternative to browser cookies. It intercepts requests, adding the Bearer token to the Authorization header before forwarding them to your API.

<Callout type="warn">

Use this cautiously; it is intended only for APIs that don't support cookies or require Bearer tokens for authentication. Improper implementation could easily lead to security vulnerabilities.

</Callout>

## ## Installing the Bearer Plugin

Add the Bearer plugin to your authentication setup:

```
```ts title="auth.ts"
import { betterAuth } from "better-auth";
import { bearer } from "better-auth/plugins";

export const auth = betterAuth({
  plugins: [bearer()]
});
```,
```

## ## How to Use Bearer Tokens

### ### 1. Obtain the Bearer Token

After a successful sign-in, you'll receive a session token in the response headers. Store this token securely (e.g., in `localStorage`):

```
```ts title="auth-client.ts"
const { data } = await authClient.signIn.email({
  email: "user@example.com",
  password: "securepassword"
}, {
  onSuccess: (ctx) => {
    const authToken = ctx.response.headers.get("set-auth-token") // get the token from the response headers
    // Store the token securely (e.g., in localStorage)
    localStorage.setItem("bearer_token", authToken);
  }
});
```,
```

You can also set this up globally in your auth client:

```
```ts title="auth-client.ts"
export const authClient = createAuthClient({
  fetchOptions: {
    onSuccess: (ctx) => {
      const authToken = ctx.response.headers.get("set-auth-token") // get the token from the response headers
```



```

    // Store the token securely (e.g., in localStorage)
    if(authToken){
      localStorage.setItem("bearer_token", authToken);
    }
  }
}
});
```

```

You may want to clear the token based on the response status code or other conditions:

### ### 2. Configure the Auth Client

Set up your auth client to include the Bearer token in all requests:

```

```ts title="auth-client.ts"
export const authClient = createAuthClient({
  fetchOptions: {
    auth: {
      type: "Bearer",
      token: () => localStorage.getItem("bearer_token") || "" // get the token from localStorage
    }
  }
});
```

```

### ### 3. Make Authenticated Requests

Now you can make authenticated API calls:

```

```ts title="auth-client.ts"
// This request is automatically authenticated
const { data } = await authClient.listSessions();
```

```

### ### 4. Per-Request Token (Optional)

You can also provide the token for individual requests:

```

```ts title="auth-client.ts"
const { data } = await authClient.listSessions({
  fetchOptions: {
    headers: {
      Authorization: `Bearer ${token}`
    }
  }
});
```

```

### ### 5. Using Bearer Tokens Outside the Auth Client

The Bearer token can be used to authenticate any request to your API, even when not using the auth client:

```

```ts title="api-call.ts"
const token = localStorage.getItem("bearer_token");

const response = await fetch("https://api.example.com/data", {
  headers: {
    Authorization: `Bearer ${token}`
  }
});

const data = await response.json();
```

```

And in the server, you can use the `auth.api.getSession` function to authenticate requests:

```
```ts title="server.ts"
import { auth } from "@auth";

export async function handler(req, res) {
  const session = await auth.api.getSession({
    headers: req.headers
  });

  if (!session) {
    return res.status(401).json({ error: "Unauthorized" });
  }

  // Process authenticated request
  // ...
}
```
```

### ## Options

**requireSignature** (boolean): Require the token to be signed. Default: `false`.

```
file: ./content/docs/plugins/captcha.mdx
meta: {
  "title": "Captcha",
  "description": "Captcha plugin"
}
```

The **Captcha Plugin** integrates bot protection into your Better Auth system by adding captcha verification for key endpoints. This plugin ensures that only human users can perform actions like signing up, signing in, or resetting passwords. The following providers are currently supported:

- \* [Google reCAPTCHA](https://developers.google.com/recaptcha)
- \* [Cloudflare Turnstile](https://www.cloudflare.com/application-services/products/turnstile/)
- \* [hCaptcha](https://www.hcaptcha.com/)

<Callout type="info">

This plugin works out of the box with <Link href="/docs/authentication/email-password">Email & Password</Link> authentication. To use it with other authentication methods, you will need to configure the <Link href="/docs/plugins/captcha#plugin-options">endpoints</Link> array in the plugin options.

</Callout>

### ## Installation

<Steps>

<Step>

### Add the plugin to your **auth** config

```
```ts title="auth.ts"
import { betterAuth } from "better-auth";
import { captcha } from "better-auth/plugins";

export const auth = betterAuth({
  plugins: [ // [!code highlight]
    captcha({ // [!code highlight]
      provider: "cloudflare-turnstile", // or google-recaptcha, hcaptcha // [!code highlight]
      secretKey: process.env.TURNSTILE_SECRET_KEY!, // [!code highlight]
    }), // [!code highlight]
  ], // [!code highlight]
});
```
```

</Step>

<Step>

#### Add the captcha token to your request headers

Add the captcha token to your request headers for all protected endpoints. This example shows how to include it in a `signIn` request:

```

```ts
await authClient.signIn.email({
  email: "user@example.com",
  password: "secure-password",
  fetchOptions: { // [!code highlight]
    headers: { // [!code highlight]
      "x-captcha-response": turnstileToken, // [!code highlight]
      "x-captcha-user-remote-ip": userIp, // optional: forwards the user's IP address to the captcha service // [!code highlight]
    }, // [!code highlight]
  }, // [!code highlight]
});
```

```

\* To implement Cloudflare Turnstile on the client side, follow the official [Cloudflare Turnstile documentation] (<https://developers.cloudflare.com/turnstile/>) or use a library like [react-turnstile] (<https://www.npmjs.com/package/@marsidev/react-turnstile>).

\* To implement Google reCAPTCHA on the client side, follow the official [Google reCAPTCHA documentation] (<https://developers.google.com/recaptcha/intro>) or use libraries like [react-google-recaptcha] (<https://www.npmjs.com/package/react-google-recaptcha>) (v2) and [react-google-recaptcha-v3] (<https://www.npmjs.com/package/react-google-recaptcha-v3>) (v3).

\* To implement hCaptcha on the client side, follow the official [hCaptcha documentation] (<https://docs.hcaptcha.com/#add-the-hcaptcha-widget-to-your-webpage>) or use libraries like [@hcaptcha/react-hcaptcha] (<https://www.npmjs.com/package/@hcaptcha/react-hcaptcha>)

</Step>

</Steps>

## ## How it works

<Steps>

<Step>

The plugin acts as a middleware: it intercepts all `POST` requests to configured endpoints (see `endpoints` in the [Plugin Options](#plugin-options) section).

</Step>

<Step>

it validates the captcha token on the server, by calling the captcha provider's `/siteverify`.

</Step>

<Step>

\* if the token is missing, gets rejected by the captcha provider, or if the `/siteverify` endpoint is unavailable, the plugin returns an error and interrupts the request.

\* if the token is accepted by the captcha provider, the middleware returns `undefined`, meaning the request is allowed to proceed.

</Step>

</Steps>

## ## Plugin Options

\* \*\*`provider` (required)\*\*: your captcha provider.

\* \*\*`secretKey` (required)\*\*: your provider's secret key used for the server-side validation.

\* `endpoints` (optional): overrides the default array of paths where captcha validation is enforced. Default is: `["/sign-up/email", "/sign-in/email", "/forget-password"]`.

\* `minScore` (optional - only \*Google ReCAPTCHA v3\*): minimum score threshold. Default is `0.5`.

\* `siteKey` (optional - only \*hCaptcha\*): prevents tokens issued on one sitekey from being redeemed elsewhere.

\* `siteVerifyURLOverride` (optional): overrides endpoint URL for the captcha verification request.

file: ./content/docs/plugins/community-plugins.mdx

```
meta: {
  "title": "Community Plugins",
  "description": "A list of recommended community plugins."
}
```

This page showcases a list of recommended community made plugins.

We encourage you to create custom plugins and maybe get added to the list!

To create your own custom plugin, get started by reading our [plugins documentation](https://www.better-auth.com/docs/concepts/plugins). And if you want to share your plugin with the community, please open a pull request to add it to this list.

```
| <div className="w-[200px]">Plugin</div> | Description
| <div className="w-[150px]">Author</div> |
-----	-----
[better-auth-harmony](https://github.com/gekorm/better-auth-harmony/)	Email & phone normalization and
additional validation, blocking over 55,000 temporary email domains.	[GeKorm](https://github.com/GeKorm)
[validation-better-auth](https://github.com/Daanish2003/validation-better-auth)	Validate API request using any
validation library (e.g., Zod, Yup) |  [Daanish2003](https://github.com/Daanish2003) |
```

```
file: ./content/docs/plugins/dub.mdx
```

```
meta: {
  "title": "Dub",
  "description": "Better Auth Plugin for Lead Tracking using Dub links and OAuth Linking"
}
```

[Dub](https://dub.co/) is an open source modern link management platform for entrepreneurs, creators, and growth teams.

This plugins allows you to track leads when a user signs up using a Dub link. It also adds OAuth linking support to allow you to build integrations extending Dub's linking management infrastructure.

## ## Installation

### <Steps>

#### <Step>

##### #### Install the plugin

First, install the plugin:

```
<Tabs groupId="package-manager" persist items={}>
```

```
<Tab value="npm">
```

```
  `` bash
```

```
  npm install @dub/better-auth
```

```
  ``
```

```
</Tab>
```

```
<Tab value="pnpm">
```

```
  `` bash
```

```
  pnpm add @dub/better-auth
```

```
  ``
```

```
</Tab>
```

```
<Tab value="yarn">
```

```
  `` bash
```

```
  yarn add @dub/better-auth
```

```
  ``
```

```
</Tab>
```

```
<Tab value="bun">
```

```

    ```bash
    bun add @dub/better-auth
    ```

  </Tab>
</Tabs>
</Step>

```

```

<Step>
  #### Install the Dub SDK

```

Next, install the Dub SDK on your server:

```

<Tabs groupId="package-manager" persist items={}>
  <Tab value="npm">
    ```bash
    npm install dub
    ```

  </Tab>

  <Tab value="pnpm">
    ```bash
    pnpm add dub
    ```

  </Tab>

  <Tab value="yarn">
    ```bash
    yarn add dub
    ```

  </Tab>

  <Tab value="bun">
    ```bash
    bun add dub
    ```

  </Tab>
</Tabs>
</Step>

```

```

<Step>
  #### Configure the plugin

```

Add the plugin to your auth config:

```

```ts title="auth.ts"
import { betterAuth } from "better-auth"
import { dubAnalytics } from "@dub/better-auth"
import { dub } from "dub"

export const auth = betterAuth({
  plugins: [
    dubAnalytics({
      dubClient: new Dub()
    })
  ]
})
```

</Step>
</Steps>

```

## ## Usage

### #### Lead Tracking

By default, the plugin will track sign up events as leads. You can disable this by setting `disableLeadTracking` to `true`.

```

```ts
import { dubAnalytics } from "@dub/better-auth";
import { betterAuth } from "better-auth";
import { Dub } from "dub";

const dub = new Dub();

const betterAuth = betterAuth({
  plugins: [
    dubAnalytics({
      dubClient: dub,
      disableLeadTracking: true, // Disable lead tracking
    }),
  ],
});
```

```

### ### OAuth Linking

The plugin supports OAuth for account linking.

First, you need to setup OAuth app in Dub. Dub supports OAuth 2.0 authentication, which is recommended if you build integrations extending Dub's functionality [Learn more about OAuth] (<https://dub.co/docs/integrations/quickstart#integrating-via-oauth-2-0-recommended>).

Once you get the client ID and client secret, you can configure the plugin.

```

```ts
dubAnalytics({
  dubClient: dub,
  oauth: {
    clientId: "your-client-id",
    clientSecret: "your-client-secret",
  },
});
```

```

And in the client, you need to use the `dubAnalyticsClient` plugin.

```

```ts
import { createAuthClient } from "better-auth/client";
import { dubAnalyticsClient } from "@dub/better-auth/client";

const authClient = createAuthClient({
  plugins: [dubAnalyticsClient()],
});
```

```

To link account with Dub, you need to use the `dub.link`.

```

```ts
const response = await authClient.dub.link({
  callbackURL: "/dashboard", // URL to redirect to after linking
});
```

```

### ## Options

You can pass the following options to the plugin:

#### ### `dubClient`

The Dub client instance.

#### `disableLeadTracking`

Disable lead tracking for sign up events.

#### `leadEventName`

Event name for sign up leads.

#### `customLeadTrack`

Custom lead track function.

#### `oauth`

Dub OAuth configuration.

#### `oauth.clientId`

Client ID for Dub OAuth.

#### `oauth.clientSecret`

Client secret for Dub OAuth.

#### `oauth.pkce`

Enable PKCE for Dub OAuth.

file: ./content/docs/plugins/email-otp.mdx

```
meta: {
  "title": "Email OTP",
  "description": "Email OTP plugin for Better Auth."
}
```

The Email OTP plugin allows user to sign in, verify their email, or reset their password using a one-time password (OTP) sent to their email address.

## ## Installation

<Steps>

<Step>

#### Add the plugin to your auth config

To enable email otp in your app, you need to add the `emailOTP` plugin to your auth config.

```
```ts title="auth.ts"
import { betterAuth } from "better-auth"
import { emailOTP } from "better-auth/plugins" // [!code highlight]

export const auth = betterAuth({
  // ... other config options
  plugins: [
    emailOTP({ // [!code highlight]
      async sendVerificationOTP({ email, otp, type }) { // [!code highlight]
        // Implement the sendVerificationOTP method to send the OTP to the user's email address //
        [!code highlight]
      }, // [!code highlight]
    }) // [!code highlight]
  ]
})
```
```

</Step>

<Step>

### ### Add the client plugin

```
```ts title="auth-client.ts"
import { createAuthClient } from "better-auth/client"
import { emailOTPClient } from "better-auth/client/plugins"

export const authClient = createAuthClient({
  plugins: [
    emailOTPClient()
  ]
})
```
</Step>
</Steps>
```

## ## Usage

### ### Send OTP

First, send an OTP to the user's email address.

```
```ts title="example.ts"
const { data, error } = await authClient.emailOtp.sendVerificationOtp({
  email: "user-email@email.com",
  type: "sign-in" // or "email-verification", "forget-password"
})
```
```

### ### Sign in with OTP

Once the user provides the OTP, you can sign in the user using the `signIn.emailOtp()` method.

```
```ts title="example.ts"
const { data, error } = await authClient.signIn.emailOtp({
  email: "user-email@email.com",
  otp: "123456"
})
```
```

If the user is not registered, they'll be automatically registered. If you want to prevent this, you can pass `disableSignUp` as `true` in the options.

### ### Verify Email

To verify the user's email address, use the `verifyEmail()` method.

```
```ts title="example.ts"
const { data, error } = await authClient.emailOtp.verifyEmail({
  email: "user-email@email.com",
  otp: "123456"
})
```
```

### ### Reset Password

To reset the user's password, use the `resetPassword()` method.

```
```ts title="example.ts"
const { data, error } = await authClient.emailOtp.resetPassword({
  email: "user-email@email.com",
  otp: "123456",
  password: "password"
})
```
```



## ## Options

\* `sendVerificationOTP`: A function that sends the OTP to the user's email address. The function receives an object with the following properties:

\* `email`: The user's email address.

\* `otp`: The OTP to send.

\* `type`: The type of OTP to send. Can be "sign-in", "email-verification", or "forget-password".

## ### Example

```
```ts title="auth.ts"
import { betterAuth } from "better-auth"

export const auth = betterAuth({
  plugins: [
    emailOTP({
      async sendVerificationOTP({
        email,
        otp,
        type
      }) {
        if (type === "sign-in") {
          // Send the OTP for sign-in
        } else if (type === "email-verification") {
          // Send the OTP for email verification
        } else {
          // Send the OTP for password reset
        }
      }
    })
  ]
})
```
```

\* `otpLength`: The length of the OTP. Defaults to `6`.

\* `expiresIn`: The expiry time of the OTP in seconds. Defaults to `300` seconds.

```
```ts title="auth.ts"
import { betterAuth } from "better-auth"

export const auth = betterAuth({
  plugins: [
    emailOTP({
      otpLength: 8,
      expiresIn: 600
    })
  ]
})
```
```

\* `sendVerificationOnSignUp`: A boolean value that determines whether to send the OTP when a user signs up. Defaults to `false`.

\* `disableSignUp`: A boolean value that determines whether to prevent automatic sign-up when the user is not registered. Defaults to `false`.

\* `generateOTP`: A function that generates the OTP. Defaults to a random 6-digit number.

\* `allowedAttempts`: The maximum number of attempts allowed for verifying an OTP. Defaults to `3`. After exceeding this limit, the OTP becomes invalid and the user needs to request a new one.

```
```ts title="auth.ts"
import { betterAuth } from "better-auth"

export const auth = betterAuth({
```

```

plugins: [
  emailOTP({
    allowedAttempts: 5, // Allow 5 attempts before invalidating the OTP
    expiresIn: 300
  })
]
})
```

```

When the maximum attempts are exceeded, the `verifyOTP`, `signIn.emailOtp`, `verifyEmail`, and `resetPassword` methods will return an error with code `MAX\_ATTEMPTS\_EXCEEDED`.

```

file: ./content/docs/plugins/generic-oauth.mdx
meta: {
  "title": "Generic OAuth",
  "description": "Authenticate users with any OAuth provider"
}

```

The Generic OAuth plugin provides a flexible way to integrate authentication with any OAuth provider. It supports both OAuth 2.0 and OpenID Connect (OIDC) flows, allowing you to easily add social login or custom OAuth authentication to your application.

## ## Installation

### <Steps>

#### <Step>

#### Add the plugin to your auth config

To use the Generic OAuth plugin, add it to your auth config.

```

```ts title="auth.ts"
import { betterAuth } from "better-auth"
import { genericOAuth } from "better-auth/plugins" // [!code highlight]

export const auth = betterAuth({
  // ... other config options
  plugins: [
    genericOAuth({ // [!code highlight]
      config: [ // [!code highlight]
        { // [!code highlight]
          providerId: "provider-id", // [!code highlight]
          clientId: "test-client-id", // [!code highlight]
          clientSecret: "test-client-secret", // [!code highlight]
          discoveryUrl: "https://auth.example.com/.well-known/openid-configuration", // [!code highlight]
          // ... other config options // [!code highlight]
        }, // [!code highlight]
        // Add more providers as needed // [!code highlight]
      ] // [!code highlight]
    }) // [!code highlight]
  ]
})
```

```

#### </Step>

#### <Step>

#### Add the client plugin

Include the Generic OAuth client plugin in your authentication client instance.

```

```ts title="auth-client.ts"
import { createAuthClient } from "better-auth/client"
import { genericOAuthClient } from "better-auth/client/plugins"

export const authClient = createAuthClient({

```

```

    plugins: [
      genericOAuthClient()
    ]
  })
  ...
</Step>
</Steps>

```

## ## Usage

The Generic OAuth plugin provides endpoints for initiating the OAuth flow and handling the callback. Here's how to use them:

### ### Initiate OAuth Sign-In

To start the OAuth sign-in process:

```

```ts title="sign-in.ts"
const response = await authClient.signIn.oauth2({
  providerId: "provider-id",
  callbackURL: "/dashboard" // the path to redirect to after the user is authenticated
});
```

```

### ### Linking OAuth Accounts

To link an OAuth account to an existing user:

```

```ts title="link-account.ts"
const response = await authClient.oauth2.link({
  providerId: "provider-id",
  callbackURL: "/dashboard" // the path to redirect to after the account is linked
});
```

```

### ### Handle OAuth Callback

The plugin mounts a route to handle the OAuth callback `/oauth2/callback/:providerId`. This means by default ``${baseURL}/api/auth/oauth2/callback/:providerId`` will be used as the callback URL. Make sure your OAuth provider is configured to use this URL.

## ## Configuration

When adding the plugin to your auth config, you can configure multiple OAuth providers. Each provider configuration object supports the following options:

```

```ts
interface GenericOAuthConfig {
  providerId: string;
  discoveryUrl?: string;
  authorizationUrl?: string;
  tokenUrl?: string;
  userInfoUrl?: string;
  clientId: string;
  clientSecret: string;
  scopes?: string[];
  redirectURI?: string;
  responseType?: string;
  prompt?: string;
  pkce?: boolean;
  accessType?: string;
  getUserInfo?: (tokens: OAuth2Tokens) => Promise<User | null>;
}
```

```

### ### Other Provider Configurations

**\*\*providerId\*\***: A unique string to identify the OAuth provider configuration.

**\*\*discoveryUrl\*\***: (Optional) URL to fetch the provider's OAuth 2.0/OIDC configuration. If provided, endpoints like ``authorizationUrl``, ``tokenUrl``, and ``userInfoUrl`` can be auto-discovered.

**\*\*authorizationUrl\*\***: (Optional) The OAuth provider's authorization endpoint. Not required if using ``discoveryUrl``.

**\*\*tokenUrl\*\***: (Optional) The OAuth provider's token endpoint. Not required if using ``discoveryUrl``.

**\*\*userInfoUrl\*\***: (Optional) The endpoint to fetch user profile information. Not required if using ``discoveryUrl``.

**\*\*clientId\*\***: The OAuth client ID issued by your provider.

**\*\*clientSecret\*\***: The OAuth client secret issued by your provider.

**\*\*scopes\*\***: (Optional) An array of scopes to request from the provider (e.g., ``["openid", "email", "profile"]``).

**\*\*redirectURI\*\***: (Optional) The redirect URI to use for the OAuth flow. If not set, a default is constructed based on your app's base URL.

**\*\*responseType\*\***: (Optional) The OAuth response type. Defaults to ``"code"`` for authorization code flow.

**\*\*responseMode\*\***: (Optional) The response mode for the authorization code request, such as ``"query"`` or ``"form_post"``.

**\*\*prompt\*\***: (Optional) Controls the authentication experience (e.g., force login, consent, etc.).

**\*\*pkce\*\***: (Optional) If true, enables PKCE (Proof Key for Code Exchange) for enhanced security. Defaults to ``false``.

**\*\*accessType\*\***: (Optional) The access type for the authorization request. Use ``"offline"`` to request a refresh token.

**\*\*getUserInfo\*\***: (Optional) A custom function to fetch user info from the provider, given the OAuth tokens. If not provided, a default fetch is used.

**\*\*mapProfileToUser\*\***: (Optional) A function to map the provider's user profile to your app's user object. Useful for custom field mapping or transformations.

**\*\*authorizationUrlParams\*\***: (Optional) Additional query parameters to add to the authorization URL. These can override default parameters.

**\*\*disableImplicitSignUp\*\***: (Optional) If true, disables automatic sign-up for new users. Sign-in must be explicitly requested with sign-up intent.

**\*\*disableSignUp\*\***: (Optional) If true, disables sign-up for new users entirely. Only existing users can sign in.

**\*\*authentication\*\***: (Optional) The authentication method for token requests. Can be ``"basic"`` or ``"post"``. Defaults to ``"post"``.

**\*\*discoveryHeaders\*\***: (Optional) Custom headers to include in the discovery request. Useful for providers that require special headers.

**\*\*authorizationHeaders\*\***: (Optional) Custom headers to include in the authorization request. Useful for providers that require special headers.

**\*\*overrideUserInfo\*\***: (Optional) If true, the user's info in your database will be updated with the provider's info every time they sign in. Defaults to ``false``.

### ## Advanced Usage

#### ### Custom User Info Fetching

You can provide a custom ``getUserInfo`` function to handle specific provider requirements:

```
```ts
```

```
genericOAuth({
  config: [
    {
      providerId: "custom-provider",
      // ... other config options
      getUserInfo: async (tokens) => {
        // Custom logic to fetch and return user info
        const userInfo = await fetchUserInfoFromCustomProvider(tokens);
        return {
          id: userInfo.sub,
          email: userInfo.email,
          name: userInfo.name,
          // ... map other fields as needed
        };
      }
    }
  ]
})
```,
```

### #### Map User Info Fields

If the user info returned by the provider does not match the expected format, or you need to map additional fields, you can use the `mapProfileToUser`:

```
```ts
genericOAuth({
  config: [
    {
      providerId: "custom-provider",
      // ... other config options
      mapProfileToUser: async (profile) => {
        return {
          firstName: profile.given_name,
          // ... map other fields as needed
        };
      }
    }
  ]
})
```,
```

### #### Error Handling

The plugin includes built-in error handling for common OAuth issues. Errors are typically redirected to your application's error page with an appropriate error message in the URL parameters. If the callback URL is not provided, the user will be redirected to Better Auth's default error page.

```
file: ./content/docs/plugins/have-i-been-pwned.mdx
meta: {
  "title": "Have I Been Pwned",
  "description": "A plugin to check if a password has been compromised"
}
```

The Have I Been Pwned plugin helps protect user accounts by preventing the use of passwords that have been exposed in known data breaches. It uses the [Have I Been Pwned](https://haveibeenpwned.com/) API to check if a password has been compromised.

### ## Installation

```
```ts
import { betterAuth } from "better-auth"
import { haveIBeenPwned } from "better-auth/plugins" // [!code highlight]
```

```
export const auth = betterAuth({
  plugins: [
    haveIBeenPwned()
  ]
})
```,
```

### ## Usage

When a user attempts to create an account or update their password with a compromised password, they'll receive the following default error:

```
```json
{
  "code": "PASSWORD_COMPROMISED",
  "message": "Password is compromised"
}
```,
```

### ## Config

You can customize the error message:

```
```ts
haveIBeenPwned({
  customPasswordCompromisedMessage: "Please choose a more secure password."
})
```,
```

### ## Security Notes

- \* Only the first 5 characters of the password hash are sent to the API
- \* The full password is never transmitted
- \* Provides an additional layer of account security

```
file: ./content/docs/plugins/jwt.mdx
meta: {
  "title": "JWT",
  "description": "Authenticate users with JWT tokens in services that can't use the session"
}
```

The JWT plugin provides endpoints to retrieve a JWT token and a JWKS endpoint to verify the token.

<Callout type="info">

This plugin is not meant as a replacement for the session. It's meant to be used for services that require JWT tokens. If you're looking to use JWT tokens for authentication, check out the [Bearer Plugin](/docs/plugins/bearer).

</Callout>

### ## Installation

<Steps>

<Step>

#### Add the plugin to your **auth** config

```
```ts title="auth.ts"
import { betterAuth } from "better-auth"
import { jwt } from "better-auth/plugins"

export const auth = betterAuth({
  plugins: [ // [!code highlight]
    jwt(), // [!code highlight]
  ] // [!code highlight]
})
```,
```

</Step>

<Step>

#### Migrate the database

Run the migration or generate the schema to add the necessary fields and tables to the database.

<Tabs items={["migrate", "generate"]}>

<Tab value="migrate">

```bash

npx @better-auth/cli migrate

```

</Tab>

<Tab value="generate">

```bash

npx @better-auth/cli generate

```

</Tab>

</Tabs>

See the [Schema](#schema) section to add the fields manually.

</Step>

</Steps>

## ## Usage

Once you've installed the plugin, you can start using the JWT & JWKS plugin to get the token and the JWKS through their respective endpoints.

## ## JWT

### #### Retrieve the token

#### 1. Using your session token

To get the token, call the `/token` endpoint. This will return the following:

```
```json
{
  "token": "eyJ..."
}
```

Make sure to include the token in the `Authorization` header of your requests and the `bearer` plugin is added in your auth configuration.

```
```ts
await fetch("/api/auth/token", {
  headers: {
    "Authorization": `Bearer ${token}`
  },
})
```

#### 2. From `set-auth-jwt` header

When you call `getSession` method, a JWT is returned in the `set-auth-jwt` header, which you can use to send to your services directly.

```
```ts
await authClient.getSession({
  fetchOptions: {
    onSuccess: (ctx) => {
      const jwt = ctx.response.headers.get("set-auth-jwt")
    }
  }
})
```

```

    }
  }
})
```

```

### ### Verifying the token

The token can be verified in your own service, without the need for an additional verify call or database check. For this JWKS is used. The public key can be fetched from the `/api/auth/jwks` endpoint.

Since this key is not subject to frequent changes, it can be cached indefinitely.

The key ID (`kid`) that was used to sign a JWT is included in the header of the token.

In case a JWT with a different `kid` is received, it is recommended to fetch the JWKS again.

```

```json
{
  "keys": [
    {
      "crv": "Ed25519",
      "x": "bDHiLTt7u-VIU7rfmcltcFhaHKLvWvWFy-_csKZARUEU",
      "kty": "OKP",
      "kid": "c5c7995d-0037-4553-8aee-b5b620b89b23"
    }
  ]
}
```

```

### #### Example using jose with remote JWKS

```

```ts
import { jwtVerify, createRemoteJWKSet } from 'jose'

async function validateToken(token: string) {
  try {
    const JWKS = createRemoteJWKSet(
      new URL('http://localhost:3000/api/auth/jwks')
    )
    const { payload } = await jwtVerify(token, JWKS, {
      issuer: 'http://localhost:3000', // Should match your JWT issuer, which is the BASE_URL
      audience: 'http://localhost:3000', // Should match your JWT audience, which is the BASE_URL by default
    })
    return payload
  } catch (error) {
    console.error('Token validation failed:', error)
    throw error
  }
}

// Usage example
const token = 'your.jwt.token' // this is the token you get from the /api/auth/token endpoint
const payload = await validateToken(token)
```

```

### #### Example with local JWKS

```

```ts
import { jwtVerify, createLocalJWKSet } from 'jose'

async function validateToken(token: string) {
  try {
    /**
     * This is the JWKS that you get from the /api/auth/
     * jwks endpoint
     */
  }
}

```



```

const storedJWKS = {
  keys: [{
    //...
  }]
};
const JWKS = createLocalJWKSet({
  keys: storedJWKS.data?.keys!,
})
const { payload } = await jwtVerify(token, JWKS, {
  issuer: 'http://localhost:3000', // Should match your JWT issuer, which is the BASE_URL
  audience: 'http://localhost:3000', // Should match your JWT audience, which is the BASE_URL by default
})
return payload
} catch (error) {
  console.error('Token validation failed:', error)
  throw error
}
}

// Usage example
const token = 'your.jwt.token' // this is the token you get from the /api/auth/token endpoint
const payload = await validateToken(token)
```

```

## ## Schema

The JWT plugin adds the following tables to the database:

### ### JWKS

Table Name: `jwks`

```

<DatabaseTable
  fields=[
    {
      name: "id",
      type: "string",
      description: "Unique identifier for each web key",
      isPrimaryKey: true
    },
    {
      name: "publicKey",
      type: "string",
      description: "The public part of the web key"
    },
    {
      name: "privateKey",
      type: "string",
      description: "The private part of the web key"
    },
    {
      name: "createdAt",
      type: "Date",
      description: "Timestamp of when the web key was created"
    },
  ]
/>

```

## ## Options

### ### Algorithm of the Key Pair

The algorithm used for the generation of the key pair. The default is **EdDSA** with the **Ed25519** curve. Below are the available options:

```
```ts title="auth.ts"
jwt({
  jwks: {
    keyPairConfig: {
      alg: "EdDSA",
      crv: "Ed25519"
    }
  }
})
```
```

#### #### EdDSA

\* \*\*Default Curve\*\*: `Ed25519`  
 \* \*\*Optional Property\*\*: `crv`  
 \* Available options: `Ed25519`, `Ed448`  
 \* Default: `Ed25519`

#### #### ES256

\* No additional properties

#### #### RSA256

\* \*\*Optional Property\*\*: `modulusLength`  
 \* Expects a number  
 \* Default: `2048`

#### #### PS256

\* \*\*Optional Property\*\*: `modulusLength`  
 \* Expects a number  
 \* Default: `2048`

#### #### ECDH-ES

\* \*\*Optional Property\*\*: `crv`  
 \* Available options: `P-256`, `P-384`, `P-521`  
 \* Default: `P-256`

#### #### ES512

\* No additional properties

#### ### Disable private key encryption

By default, the private key is encrypted using AES256 GCM. You can disable this by setting the `disablePrivateKeyEncryption` option to `true`.

For security reasons, it's recommended to keep the private key encrypted.

```
```ts title="auth.ts"
jwt({
  jwks: {
    disablePrivateKeyEncryption: true
  }
})
```
```

#### ### Modify JWT payload

By default the entire user object is added to the JWT payload. You can modify the payload by providing a function to the `definePayload` option.

```
```ts title="auth.ts"
```

```

jwt({
  jwt: {
    definePayload: ({user}) => {
      return {
        id: user.id,
        email: user.email,
        role: user.role
      }
    }
  }
})
``

```

### ### Modify Issuer, Audience, Subject or Expiration time

If none is given, the `BASE\_URL` is used as the issuer and the audience is set to the `BASE\_URL`. The expiration time is set to 15 minutes.

```

`` ts title="auth.ts"
jwt({
  jwt: {
    issuer: "https://example.com",
    audience: "https://example.com",
    expirationTime: "1h",
    getSubject: (session) => {
      // by default the subject is the user id
      return session.user.email
    }
  }
})
``

```

```

file: ./content/docs/plugins/magic-link.mdx
meta: {
  "title": "Magic link",
  "description": "Magic link plugin"
}

```

Magic link or email link is a way to authenticate users without a password. When a user enters their email, a link is sent to their email. When the user clicks on the link, they are authenticated.

## ## Installation

### <Steps>

#### <Step>

#### ### Add the server Plugin

Add the magic link plugin to your server:

```

`` ts title="server.ts"
import { betterAuth } from "better-auth";
import { magicLink } from "better-auth/plugins";

export const auth = betterAuth({
  plugins: [
    magicLink({
      sendMagicLink: async ({ email, token, url }, request) => {
        // send email to user
      }
    })
  ]
})
``

```

#### </Step>

```
<Step>
#### Add the client Plugin
```

Add the magic link plugin to your client:

```
```ts title="auth-client.ts"
import { createAuthClient } from "better-auth/client";
import { magicLinkClient } from "better-auth/client/plugins";
export const authClient = createAuthClient({
  plugins: [
    magicLinkClient()
  ]
});
```
```

```
</Step>
</Steps>
```

## ## Usage

### #### Sign In with Magic Link

To sign in with a magic link, you need to call `signIn.magicLink` with the user's email address. The `sendMagicLink` function is called to send the magic link to the user's email.

```
```ts title="magic-link.ts"
const { data, error } = await authClient.signIn.magicLink({
  email: "user@email.com",
  callbackURL: "/dashboard", //redirect after successful login (optional)
});
```
```

If the user has not signed up, unless `disableSignUp` is set to `true`, the user will be signed up automatically.

### #### Verify Magic Link

When you send the URL generated by the `sendMagicLink` function to a user, clicking the link will authenticate them and redirect them to the `callbackURL` specified in the `signIn.magicLink` function. If an error occurs, the user will be redirected to the `callbackURL` with an error query parameter.

```
<Callout type="warn">
  If no callbackURL is provided, the user will be redirected to the root URL.
</Callout>
```

If you want to handle the verification manually, (e.g, if you send the user a different URL), you can use the `verify` function.

```
```ts title="magic-link.ts"
const { data, error } = await authClient.magicLink.verify({
  query: {
    token,
  },
});
```
```

## ## Configuration Options

**`sendMagicLink`**: The `sendMagicLink` function is called when a user requests a magic link. It takes an object with the following properties:

- \* `email`: The email address of the user.
- \* `url`: The URL to be sent to the user. This URL contains the token.
- \* `token`: The token if you want to send the token with custom URL.

and a `request` object as the second parameter.

**\*\*expiresIn\*\***: specifies the time in seconds after which the magic link will expire. The default value is `300` seconds (5 minutes).

**\*\*disableSignUp\*\***: If set to `true`, the user will not be able to sign up using the magic link. The default value is `false`.

**\*\*generateToken\*\***: The `generateToken` function is called to generate a token which is used to uniquely identify the user. The default value is a random string. There is one parameter:

\* `email`: The email address of the user.

<Callout type="warn">

When using `generateToken`, ensure that the returned string is hard to guess because it is used to verify who someone actually is in a confidential way. By default, we return a long and cryptographically secure string.

</Callout>

file: ./content/docs/plugins/mcp.mdx

```
meta: {
  "title": "MCP",
  "description": "MCP provider plugin for Better Auth"
}
```

`OAuth` `MCP`

The **\*\*MCP\*\*** plugin lets your app act as an OAuth provider for MCP clients. It handles authentication and makes it easy to issue and manage access tokens for MCP applications.

## ## Installation

<Steps>

<Step>

#### Add the Plugin

Add the MCP plugin to your auth configuration and specify the login page path.

```
```ts title="auth.ts"
import { betterAuth } from "better-auth";
import { mcp } from "better-auth/plugins";

export const auth = betterAuth({
  plugins: [
    mcp({
      loginPage: "/sign-in" // path to your login page
    })
  ]
});
```
```

<Callout>

This doesn't have a client plugin, so you don't need to make any changes to your authClient.

</Callout>

</Step>

<Step>

#### Generate Schema

Run the migration or generate the schema to add the necessary fields and tables to the database.

```
<Tabs items=["migrate", "generate"]>
<Tab value="migrate">
  ```bash
  npx @better-auth/cli migrate
  ```
</Tab>
```

```

<Tab value="generate">
  ```bash
  npx @better-auth/cli generate
  ```
</Tab>
</Tabs>

```

The MCP plugin uses the same schema as the OIDC Provider plugin. See the [OIDC Provider Schema](#schema) section for details.

```

</Step>
</Steps>

```

## ## Usage

### #### OAuth Discovery Metadata

Add a route to expose OAuth metadata for MCP clients:

```

```ts title=".well-known/oauth-authorization-server/route.ts"
import { OAuthDiscoveryMetadata } from "better-auth/plugins";
import { auth } from "../../lib/auth";

export const GET = OAuthDiscoveryMetadata(auth);
```

```

### #### MCP Session Handling

You can use the helper function `withMcpAuth` to get the session and handle unauthenticated calls automatically.

```

```ts title="api/[transport]/route.ts"
import { auth } from "@lib/auth";
import { createMcpHandler } from "@vercel/mcp-adapter";
import { withMcpAuth } from "better-auth/plugins";
import { z } from "zod";

const handler = withMcpAuth(auth, (req, session) => {
  // session contains the access token record with scopes and user ID
  return createMcpHandler(
    (server) => {
      server.tool(
        "echo",
        "Echo a message",
        { message: z.string() },
        async ({ message }) => {
          return {
            content: [{ type: "text", text: `Tool echo: ${message}` }],
          };
        },
      );
    },
  );
},
{
  capabilities: {
    tools: {
      echo: {
        description: "Echo a message",
      },
    },
  },
},
{
  redisUrl: process.env.REDIS_URL,
  basePath: "/api",
  verboseLogs: true,
  maxDuration: 60,
}

```

```

    },
  )(req);
});

export { handler as GET, handler as POST, handler as DELETE };
``,`

```

You can also use `auth.api.getMcpSession` to get the session using the access token sent from the MCP client:

```

``,`ts title="api/[transport]/route.ts"
import { auth } from "@lib/auth";
import { createMcpHandler } from "@vercel/mcp-adapter";
import { z } from "zod";

const handler = async (req: Request) => {
  // session contains the access token record with scopes and user ID
  const session = await auth.api.getMcpSession({
    headers: req.headers
  })
  if(!session){
    //this is important and you must return 401
    return new Response(null, {
      status: 401
    })
  }
  return createMcpHandler(
    (server) => {
      server.tool(
        "echo",
        "Echo a message",
        { message: z.string() },
        async ({ message }) => {
          return {
            content: [{ type: "text", text: `Tool echo: ${message}` }],
          };
        },
      );
    },
  ),
  {
    capabilities: {
      tools: {
        echo: {
          description: "Echo a message",
        },
      },
    },
  },
  {
    redisUrl: process.env.REDIS_URL,
    basePath: "/api",
    verboseLogs: true,
    maxDuration: 60,
  },
  )(req);
}

export { handler as GET, handler as POST, handler as DELETE };
``,`

```

### ### Configuration

The MCP plugin accepts the following configuration options:

```

<TypeTable
  type={{

```

```

loginPage: {
  description: "Path to the login page where users will be redirected for authentication",
  type: "string",
  required: true
},
oidcConfig: {
  description: "Optional OIDC configuration options",
  type: "object",
  required: false
}
}}
/>

```

### ### OIDC Configuration

The plugin supports additional OIDC configuration options through the `oidcConfig` parameter:

```

<TypeTable
  type={{
    codeExpiresIn: {
      description: "Expiration time for authorization codes in seconds",
      type: "number",
      default: 600
    },
    accessTokenExpiresIn: {
      description: "Expiration time for access tokens in seconds",
      type: "number",
      default: 3600
    },
    refreshTokenExpiresIn: {
      description: "Expiration time for refresh tokens in seconds",
      type: "number",
      default: 604800
    },
    defaultScope: {
      description: "Default scope for OAuth requests",
      type: "string",
      default: "openid"
    },
    scopes: {
      description: "Additional scopes to support",
      type: "string[]",
      default: ["openid", "profile", "email", "offline_access"]
    }
  }}
/>

```

### ## Schema

The MCP plugin uses the same schema as the OIDC Provider plugin. See the [OIDC Provider Schema](#schema) section for details.

file: ./content/docs/plugins/multi-session.mdx

```

meta: {
  "title": "Multi Session",
  "description": "Learn how to use multi-session plugin in Better Auth."
}

```

The multi-session plugin allows users to maintain multiple active sessions across different accounts in the same browser. This plugin is useful for applications that require users to switch between multiple accounts without logging out.

### ## Installation

#### <Steps>



```
<Step>
#### Add the plugin to your **auth** config

```ts title="auth.ts"
import { betterAuth } from "better-auth"
import { multiSession } from "better-auth/plugins"

export const auth = betterAuth({
  plugins: [ // [!code highlight]
    multiSession(), // [!code highlight]
  ] // [!code highlight]
})
```
</Step>
```

```
<Step>
#### Add the client Plugin

Add the client plugin and Specify where the user should be redirected if they need to verify 2nd factor
```

```
```ts title="auth-client.ts"
import { createAuthClient } from "better-auth/client"
import { multiSessionClient } from "better-auth/client/plugins"

export const authClient = createAuthClient({
  plugins: [
    multiSessionClient()
  ]
})
```
</Step>
</Steps>
```

## ## Usage

Whenever a user logs in, the plugin will add additional cookie to the browser. This cookie will be used to maintain multiple sessions across different accounts.

### #### List all device sessions

To list all active sessions for the current user, you can call the `listDeviceSessions` method.

```
```ts
await authClient.multiSession.listDeviceSessions()
```
```

on the server you can call `listDeviceSessions` method.

```
```ts
await auth.api.listDeviceSessions()
```
```

### #### Set active session

To set the active session, you can call the `setActive` method.

```
```ts
await authClient.multiSession.setActive({
  sessionToken: "session-token"
})
```
```

### #### Revoke a session

To revoke a session, you can call the `revoke` method.

```
```ts
await authClient.multiSession.revoke({
  sessionToken: "session-token"
})
```
```

### #### Signout and Revoke all sessions

When a user logs out, the plugin will revoke all active sessions for the user. You can do this by calling the existing `signOut` method, which handles revoking all sessions automatically.

### #### Max Sessions

You can specify the maximum number of sessions a user can have by passing the `maximumSessions` option to the plugin. By default, the plugin allows 5 sessions per device.

```
```ts title="auth.ts"
import { betterAuth } from "better-auth"

export const auth = betterAuth({
  plugins: [
    multiSession({
      maximumSessions: 3
    })
  ]
})
```
```

```
file: ./content/docs/plugins/oauth-proxy.mdx
meta: {
  "title": "OAuth Proxy",
  "description": "OAuth Proxy plugin for Better Auth"
}
```

A proxy plugin, that allows you to proxy OAuth requests. Useful for development and preview deployments where the redirect URL can't be known in advance to add to the OAuth provider.

## ## Installation

### <Steps>

#### <Step>

#### Add the plugin to your **auth** config

```
```ts title="auth.ts"
import { betterAuth } from "better-auth"
import { OAuthProxy } from "better-auth/plugins"

export const auth = betterAuth({
  plugins: [ // [!code highlight]
    OAuthProxy({
      productionURL: "https://my-main-app.com", // Optional - if the URL isn't inferred correctly // [!code highlight]
      currentURL: "http://localhost:3000", // Optional - if the URL isn't inferred correctly // [!code highlight]
    }), // [!code highlight]
  ] // [!code highlight]
})
```
```

#### </Step>

#### <Step>

#### Add redirect URL to your OAuth provider

For the proxy server to work properly, you'll need to pass the redirect URL of your main production app registered with the OAuth provider in your social provider config. This needs to be done for each social provider you want to proxy requests

for.

```

```ts
export const auth = betterAuth({
  plugins: [
    oAuthProxy(),
  ],
  socialProviders: {
    github: {
      clientId: "your-client-id",
      clientSecret: "your-client-secret",
      redirectURI: "https://my-main-app.com/api/auth/callback/github" // [!code highlight]
    }
  }
})
```
</Step>
</Steps>

```

### ## How it works

The plugin adds an endpoint to your server that proxies OAuth requests. When you initiate a social sign-in, it sets the redirect URL to this proxy endpoint. After the OAuth provider redirects back to your server, the plugin then forwards the user to the original callback URL.

```

```ts
await authClient.signIn.social({
  provider: "github",
  callbackURL: "/dashboard" // the plugin will override this to something like "http://localhost:3000/api/auth/oauth-proxy?callbackURL=/dashboard"
})
```

```

When the OAuth provider returns the user to your server, the plugin automatically redirects them to the intended callback URL.

#### <Callout>

To share cookies between the proxy server and your main server it uses URL query parameters to pass the cookies encrypted in the URL. This is secure as the cookies are encrypted and can only be decrypted by the server.

#### </Callout>

### ## Options

**currentURL**: The application's current URL is automatically determined by the plugin. It first it check for the request URL if invoked by a client, then it checks the base URL from popular hosting providers, and finally falls back to the `baseURL` in your auth config. If the URL isn't inferred correctly, you can specify it manually here.

**productionURL**: If this value matches the `baseURL` in your auth config, requests will not be proxied. Defaults to the `BETTER\_AUTH\_URL` environment variable.

file: ./content/docs/plugins/oidc-provider.mdx

```

meta: {
  "title": "OIDC Provider",
  "description": "Open ID Connect plugin for Better Auth that allows you to have your own OIDC provider."
}

```

The **OIDC Provider Plugin** enables you to build and manage your own OpenID Connect (OIDC) provider, granting full control over user authentication without relying on third-party services like Okta or Azure AD. It also allows other services to authenticate users through your OIDC provider.

#### **Key Features**:

- Client Registration**: Register clients to authenticate with your OIDC provider.

- Dynamic Client Registration**: Allow clients to register dynamically.

- \* \*\*Authorization Code Flow\*\*\*: Support the Authorization Code Flow.
- \* \*\*JWKS Endpoint\*\*\*: Publish a JWKS endpoint to allow clients to verify tokens. (Not fully implemented)
- \* \*\*Refresh Tokens\*\*\*: Issue refresh tokens and handle access token renewal using the `refresh\_token` grant.
- \* \*\*OAuth Consent\*\*\*: Implement OAuth consent screens for user authorization, with an option to bypass consent for trusted applications.
- \* \*\*UserInfo Endpoint\*\*\*: Provide a UserInfo endpoint for clients to retrieve user details.

<Callout type="warn">

This plugin is in active development and may not be suitable for production use. Please report any issues or bugs on [GitHub] (<https://github.com/better-auth/better-auth>).

</Callout>

## ## Installation

<Steps>

<Step>

### #### Mount the Plugin

Add the OIDC plugin to your auth config. See [OIDC Configuration](#oidc-configuration) on how to configure the plugin.

```
```ts title="auth.ts"
import { betterAuth } from "better-auth";
import { oidcProvider } from "better-auth/plugins";

const auth = betterAuth({
  plugins: [oidcProvider({
    loginPage: "/sign-in", // path to the login page
    // ...other options
  })]
})
```
```

</Step>

<Step>

### #### Migrate the Database

Run the migration or generate the schema to add the necessary fields and tables to the database.

```
<Tabs items={["migrate", "generate"]} >
  <Tab value="migrate">
    ```bash
    npx @better-auth/cli migrate
    ```
  </Tab>

  <Tab value="generate">
    ```bash
    npx @better-auth/cli generate
    ```
  </Tab>
</Tabs>
```

See the [Schema](#schema) section to add the fields manually.

</Step>

<Step>

### #### Add the Client Plugin

Add the OIDC client plugin to your auth client config.

```
```ts
import { createAuthClient } from "better-auth/client";
import { oidcClient } from "better-auth/client/plugins"
const authClient = createAuthClient({
  plugins: [oidcClient({
```

```

    // Your OIDC configuration
  })]
})
```
</Step>
</Steps>

```

## ## Usage

Once installed, you can utilize the OIDC Provider to manage authentication flows within your application.

### ### Register a New Client

To register a new OIDC client, use the `oauth2.register` method.

```

<Endpoint path="/oauth2/register" method="POST" />

```ts title="client.ts"
const application = await client.oauth2.register({
  client_name: "My Client",
  redirect_uris: ["https://client.example.com/callback"],
});
```

```

Once the application is created, you will receive a `client_id` and `client_secret` that you can display to the user.

This Endpoint support [RFC7591](https://datatracker.ietf.org/doc/html/rfc7591) compliant client registration.

### ### UserInfo Endpoint

The OIDC Provider includes a UserInfo endpoint that allows clients to retrieve information about the authenticated user. This endpoint is available at `/oauth2/userinfo` and requires a valid access token.

```

<Endpoint path="/oauth2/userinfo" method="GET" />

```ts title="client-app.ts"
// Example of how a client would use the UserInfo endpoint
const response = await fetch('https://your-domain.com/api/auth/oauth2/userinfo', {
  headers: {
    'Authorization': 'Bearer ACCESS_TOKEN'
  }
});

const userInfo = await response.json();
// userInfo contains user details based on the scopes granted
```

```

The UserInfo endpoint returns different claims based on the scopes that were granted during authorization:

- \* With `openid` scope: Returns the user's ID (`sub` claim)
- \* With `profile` scope: Returns name, picture, given\_name, family\_name
- \* With `email` scope: Returns email and email\_verified

The `getAdditionalUserInfoClaim` function receives the user object and the requested scopes array, allowing you to conditionally include claims based on the scopes granted during authorization. These additional claims will be included in both the UserInfo endpoint response and the ID token.

### ### Consent Screen

When a user is redirected to the OIDC provider for authentication, they may be prompted to authorize the application to access their data. This is known as the consent screen. By default, Better Auth will display a sample consent screen. You can customize the consent screen by providing a `consentPage` option during initialization.

```

```ts title="auth.ts"
import { betterAuth } from "better-auth";

```

```
export const auth = betterAuth({
  plugins: [oidcProvider({
    consentPage: "/path/to/consent/page"
  })]
})
```,
```

The plugin will redirect the user to the specified path with a `client\_id` and `scope` query parameter. You can use this information to display a custom consent screen. Once the user consents, you can call `oauth2.consent` to complete the authorization.

```
<Endpoint path="/oauth2/consent" method="POST" />
```

```
```ts title="server.ts"
const res = await client.oauth2.consent({
  accept: true, // or false to deny
});
```,
```

The `client\_id` and other necessary information are stored in the browser cookie, so you don't need to pass them in the request. If they don't exist in the cookie, the consent method will return an error.

### ### Handling Login

When a user is redirected to the OIDC provider for authentication, if they are not already logged in, they will be redirected to the login page. You can customize the login page by providing a `loginPage` option during initialization.

```
```ts title="auth.ts"
import { betterAuth } from "better-auth";

export const auth = betterAuth({
  plugins: [oidcProvider({
    loginPage: "/sign-in"
  })]
})
```,
```

You don't need to handle anything from your side; when a new session is created, the plugin will handle continuing the authorization flow.

## ## Configuration

### ### OIDC Metadata

Customize the OIDC metadata by providing a configuration object during initialization.

```
```ts title="auth.ts"
import { betterAuth } from "better-auth";
import { oidcProvider } from "better-auth/plugins";

export const auth = betterAuth({
  plugins: [oidcProvider({
    metadata: {
      issuer: "https://your-domain.com",
      authorization_endpoint: "/custom/oauth2/authorize",
      token_endpoint: "/custom/oauth2/token",
      // ...other custom metadata
    }
  })]
})
```,
```

### ### JWKS Endpoint (Not Fully Implemented)

For JWKS support, you need to use the `jwt` plugin. It exposes the `/jwks` endpoint to provide the public keys.

<Callout type="warn">

Currently, the token is signed with the application's secret key. The JWKS endpoint is not fully implemented yet.

</Callout>

### Dynamic Client Registration

If you want to allow clients to register dynamically, you can enable this feature by setting the `allowDynamicClientRegistration` option to `true`.

```
```ts title="auth.ts"
const auth = betterAuth({
  plugins: [oidcProvider({
    allowDynamicClientRegistration: true,
  })]
})
```
```

This will allow clients to register using the `/register` endpoint to be publicly available.

### Schema

The OIDC Provider plugin adds the following tables to the database:

#### OAuth Application

Table Name: `oauthApplication`

```
<DatabaseTable
  fields=[
    {
      name: "id",
      type: "string",
      description: "Database ID of the OAuth client",
      isPrimaryKey: true
    },
    {
      name: "clientId",
      type: "string",
      description: "Unique identifier for each OAuth client",
      isPrimaryKey: true
    },
    {
      name: "clientSecret",
      type: "string",
      description: "Secret key for the OAuth client",
      isRequired: true
    },
    {
      name: "name",
      type: "string",
      description: "Name of the OAuth client",
      isRequired: true
    },
    {
      name: "redirectURLs",
      type: "string",
      description: "Comma-separated list of redirect URLs",
      isRequired: true
    },
    {
      name: "metadata",
      type: "string",
      description: "Additional metadata for the OAuth client",
    }
  ]
}
```

```

    isOptional: true
  },
  {
    name: "type",
    type: "string",
    description: "Type of OAuth client (e.g., web, mobile)",
    isRequired: true
  },
  {
    name: "disabled",
    type: "boolean",
    description: "Indicates if the client is disabled",
    isRequired: true
  },
  {
    name: "userId",
    type: "string",
    description: "ID of the user who owns the client. (optional)",
    isOptional: true
  },
  {
    name: "createdAt",
    type: "Date",
    description: "Timestamp of when the OAuth client was created"
  },
  {
    name: "updatedAt",
    type: "Date",
    description: "Timestamp of when the OAuth client was last updated"
  }
]
/>

```

### ### OAuth Access Token

Table Name: `oauthAccessToken`

```

<DatabaseTable
  fields=[
    {
      name: "id",
      type: "string",
      description: "Database ID of the access token",
      isPrimaryKey: true
    },
    {
      name: "accessToken",
      type: "string",
      description: "Access token issued to the client",
    },
    {
      name: "refreshToken",
      type: "string",
      description: "Refresh token issued to the client",
      isRequired: true
    },
    {
      name: "accessTokenExpiresAt",
      type: "Date",
      description: "Expiration date of the access token",
      isRequired: true
    },
    {
      name: "refreshTokenExpiresAt",
      type: "Date",
    }
  ]

```



```

    description: "Expiration date of the refresh token",
    isRequired: true
  },
  {
    name: "clientId",
    type: "string",
    description: "ID of the OAuth client",
    isForeignKey: true,
    references: { model: "oauthClient", field: "clientId" }
  },
  {
    name: "userId",
    type: "string",
    description: "ID of the user associated with the token",
    isForeignKey: true,
    references: { model: "user", field: "id" }
  },
  {
    name: "scopes",
    type: "string",
    description: "Comma-separated list of scopes granted",
    isRequired: true
  },
  {
    name: "createdAt",
    type: "Date",
    description: "Timestamp of when the access token was created"
  },
  {
    name: "updatedAt",
    type: "Date",
    description: "Timestamp of when the access token was last updated"
  }
]
/>

```

### ### OAuth Consent

Table Name: `oauthConsent`

```

<DatabaseTable
  fields=[
    {
      name: "id",
      type: "string",
      description: "Database ID of the consent",
      isPrimaryKey: true
    },
    {
      name: "userId",
      type: "string",
      description: "ID of the user who gave consent",
      isForeignKey: true,
      references: { model: "user", field: "id" }
    },
    {
      name: "clientId",
      type: "string",
      description: "ID of the OAuth client",
      isForeignKey: true,
      references: { model: "oauthClient", field: "clientId" }
    },
    {
      name: "scopes",
      type: "string",

```

```

    description: "Comma-separated list of scopes consented to",
    isRequired: true
  },
  {
    name: "consentGiven",
    type: "boolean",
    description: "Indicates if consent was given",
    isRequired: true
  },
  {
    name: "createdAt",
    type: "Date",
    description: "Timestamp of when the consent was given"
  },
  {
    name: "updatedAt",
    type: "Date",
    description: "Timestamp of when the consent was last updated"
  }
]}
/>

```

### ### Options

**\*\*allowDynamicClientRegistration\*\***: `boolean` - Enable or disable dynamic client registration.

**\*\*metadata\*\***: `OIDCMetadata` - Customize the OIDC provider metadata.

**\*\*loginPage\*\***: `string` - Path to the custom login page.

**\*\*consentPage\*\***: `string` - Path to the custom consent page.

**\*\*getAdditionalUserInfoClaim\*\***: `(user: User, scopes: string[]) => Record<string, any>` - Function to get additional user info claims.

```

file: ./content/docs/plugins/one-tap.mdx
meta: {
  "title": "One Tap",
  "description": "One Tap plugin for Better Auth"
}

```

The One Tap plugin allows users to log in with a single tap using Google's One Tap API. The plugin provides a simple way to integrate One Tap into your application, handling the client-side and server-side logic for you.

### ### Installation

#### #### Add the Server Plugin

Add the One Tap plugin to your auth configuration:

```

```ts title="auth.ts"
import { betterAuth } from "better-auth";
import { oneTap } from "better-auth/plugins"; // [!code highlight]

export const auth = betterAuth({
  plugins: [ // [!code highlight]
    oneTap(), // Add the One Tap server plugin // [!code highlight]
  ] // [!code highlight]
});
```

```

#### #### Add the Client Plugin

Add the client plugin and specify where the user should be redirected after sign-in or if additional verification (like 2FA) is

needed.

```
```ts
import { createAuthClient } from "better-auth/client";
import { oneTapClient } from "better-auth/client/plugins";

export const authClient = createAuthClient({
  plugins: [
    oneTapClient({
      clientId: "YOUR_CLIENT_ID",
      // Optional client configuration:
      autoSelect: false,
      cancelOnTapOutside: true,
      context: "signin",
      additionalOptions: {
        // Any extra options for the Google initialize method
      },
      // Configure prompt behavior and exponential backoff:
      promptOptions: {
        baseDelay: 1000, // Base delay in ms (default: 1000)
        maxAttempts: 5 // Maximum number of attempts before triggering onPromptNotification (default: 5)
      }
    })
  ]
});
```
```

### ### Usage

To display the One Tap popup, simply call the `oneTap` method on your auth client:

```
```ts
await authClient.oneTap();
```
```

### ### Customizing Redirect Behavior

By default, after a successful login the plugin will hard redirect the user to ``/``. You can customize this behavior as follows:

#### #### Avoiding a Hard Redirect

Pass `fetchOptions` with an `onSuccess` callback to handle the login response without a page reload:

```
```ts
await authClient.oneTap({
  fetchOptions: {
    onSuccess: () => {
      // For example, use a router to navigate without a full reload:
      router.push("/dashboard");
    }
  }
});
```
```

#### #### Specifying a Custom Callback URL

To perform a hard redirect to a different page after login, use the `callbackURL` option:

```
```ts
await authClient.oneTap({
  callbackURL: "/dashboard"
});
```
```

#### #### Handling Prompt Dismissals with Exponential Backoff

If the user dismisses or skips the prompt, the plugin will retry showing the One Tap prompt using exponential backoff based on your configured promptOptions.

If the maximum number of attempts is reached without a successful sign-in, you can use the onPromptNotification callback to be notified—allowing you to render an alternative UI (e.g., a traditional Google Sign-In button) so users can restart the process manually:

```
```ts
await authClient.oneTap({
  onPromptNotification: (notification) => {
    console.warn("Prompt was dismissed or skipped. Consider displaying an alternative sign-in option.", notification);
    // Render your alternative UI here
  }
});
```
```

### #### Client Options

- \* \*\*clientId\*\*: The client ID for your Google One Tap API.
- \* \*\*autoSelect\*\*: Automatically select the account if the user is already signed in. Default is false.
- \* \*\*context\*\*: The context in which the One Tap API should be used (e.g., "signin"). Default is "signin".
- \* \*\*cancelOnTapOutside\*\*: Cancel the One Tap popup when the user taps outside it. Default is true.
- \* additionalOptions: Extra options to pass to Google's initialize method as per the [Google Identity Services docs] (<https://developers.google.com/identity/gsi/web/reference/js-reference#google.accounts.id.prompt>).
- \* \*\*promptOptions\*\*: Configuration for the prompt behavior and exponential backoff:
- \* \*\*baseDelay\*\*: Base delay in milliseconds for retries. Default is 1000.
- \* \*\*maxAttempts\*\*: Maximum number of prompt attempts before invoking the onPromptNotification callback. Default is 5.

### #### Server Options

- \* \*\*disableSignUp\*\*: Disable the sign-up option, allowing only existing users to sign in. Default is `false`.
- \* \*\*ClientId\*\*: Optionally, pass a client ID here if it is not provided in your social provider configuration.

```
file: ./content/docs/plugins/one-time-token.mdx
meta: {
  "title": "One-Time Token Plugin",
  "description": "Generate and verify single-use token"
}
```

The One-Time Token (OTT) plugin provides functionality to generate and verify secure, single-use session tokens. These are commonly used for across domains authentication.

## ## Installation

```
```ts title="auth.ts"
import { betterAuth } from "better-auth";
import { oneTimeToken } from "better-auth/plugins/one-time-token";

export const auth = betterAuth({
  plugins: [
    oneTimeToken()
  ]
  // ... other auth config
});
```
```

## ## Usage

### #### 1. Generate a Token

Generate a token using `auth.api.generateOneTimeToken` or `authClient.oneTimeToken.generate`

<Tabs items={["Server", "Client"]}>

```

<Tab value="Server">
  `` `ts
  const response = await auth.api.generateOneTimeToken({
    headers: await headers() // pass the request headers
  })
  `` `
</Tab>

<Tab value="Client">
  `` `ts
  const response = await authClient.oneTimeToken.generate()
  `` `
</Tab>
</Tabs>

```

This will return a `token` that is attached to the current session which can be used to verify the one-time token. By default, the token will expire in 3 minutes.

### ### 2. Verify the Token

When the user clicks the link or submits the token, use the `auth.api.verifyOneTimeToken` or `authClient.oneTimeToken.verify` method in another API route to validate it.

```

<Tabs items=[["Server", "Client"]]>
  <Tab value="Server">
    `` `ts
    const token = request.query.token as string; //retrieve a token
    const response = await auth.api.verifyOneTimeToken({
      body: {
        token
      }
    })
    `` `
  </Tab>

  <Tab value="Client">
    `` `ts
    const url = window.location.href;
    const token = url.split("token=")[1]; //retrieve a token
    const response = await authClient.oneTimeToken.verify({
      token
    })
    `` `
  </Tab>
</Tabs>

```

This will return the session that was attached to the token.

### ## Options

These options can be configured when adding the `oneTimeToken` plugin:

**\* \*\*`disableClientRequest`\*\* (boolean):** Optional. If `true`, the token will only be generated on the server side. Default: `false`.

**\* \*\*`expiresIn`\*\* (number):** Optional. The duration for which the token is valid in minutes. Default: `3`.

```

`` `ts
oneTimeToken({
  expiresIn: 10 // 10 minutes
})
`` `

```

**\* \*\*`generateToken`\*\*:** A custom token generator function that takes `session` object and a `ctx` as paramters.

```
file: ./content/docs/plugins/open-api.mdx
meta: {
  "title": "Open API",
  "description": "Open API reference for Better Auth."
}
```

This is a plugin that provides an Open API reference for Better Auth. It shows all endpoints added by plugins and the core. It also provides a way to test the endpoints. It uses [Scalar](https://scalar.com/) to display the Open API reference.

<Callout>

This plugin is still in the early stages of development. We are working on adding more features to it and filling in the gaps.

</Callout>

## ## Installation

<Steps>

<Step>

#### Add the plugin to your **auth** config

```
```ts title="auth.ts"
import { betterAuth } from "better-auth"
import { openAPI } from "better-auth/plugins"

export const auth = betterAuth({
  plugins: [ // [!code highlight]
    openAPI(), // [!code highlight]
  ] // [!code highlight]
})
```
```

</Step>

<Step>

#### Navigate to `/api/auth/reference` to view the Open API reference

Each plugin endpoints are grouped by the plugin name. The core endpoints are grouped under the `Default` group. And Model schemas are grouped under the `Models` group.

![Open API reference](/open-api-reference.png)

</Step>

</Steps>

## ## Usage

The Open API reference is generated using the [OpenAPI 3.0](https://swagger.io/specification/) specification. You can use the reference to generate client libraries, documentation, and more.

The reference is generated using the [Scalar](https://scalar.com/) library. Scalar provides a way to view and test the endpoints. You can test the endpoints by clicking on the `Try it out` button and providing the required parameters.

![Open API reference](/open-api-reference.png)

### #### Generated Schema

To get the generated Open API schema directly as JSON, you can do `auth.api.generateOpenAPISchema()`. This will return the Open API schema as a JSON object.

```
```ts
import { auth } from "~/lib/auth"

const openAPISchema = await auth.api.generateOpenAPISchema()
console.log(openAPISchema)
```
```

## ## Configuration

`path` - The path where the Open API reference is served. Default is `/api/auth/reference`. You can change it to any path you like, but keep in mind that it will be appended to the base path of your auth server.

`disableDefaultReference` - If set to `true`, the default Open API reference UI by Scalar will be disabled. Default is `false`.

file: ./content/docs/plugins/organization.mdx

```
meta: {
  "title": "Organization",
  "description": "The organization plugin allows you to manage your organization's members and teams."
}
```

Organizations simplifies user access and permissions management. Assign roles and permissions to streamline project management, team coordination, and partnerships.

## ## Installation

### <Steps>

#### <Step>

#### Add the plugin to your **auth** config

```
```ts title="auth.ts"
import { betterAuth } from "better-auth"
import { organization } from "better-auth/plugins"

export const auth = betterAuth({
  plugins: [ // [!code highlight]
    organization() // [!code highlight]
  ] // [!code highlight]
})
```
```

#### </Step>

#### <Step>

#### Migrate the database

Run the migration or generate the schema to add the necessary fields and tables to the database.

<Tabs items=["migrate", "generate"]>

<Tab value="migrate">

```
```bash
npx @better-auth/cli migrate
```
```

</Tab>

<Tab value="generate">

```
```bash
npx @better-auth/cli generate
```
```

</Tab>

</Tabs>

See the [Schema](#schema) section to add the fields manually.

#### </Step>

#### <Step>

#### Add the client plugin

```
```ts title="auth-client.ts"
import { createAuthClient } from "better-auth/client"
import { organizationClient } from "better-auth/client/plugins"
```

```
export const authClient = createAuthClient({
  plugins: [ // [!code highlight]
```

```

    organizationClient() // [!code highlight]
  ] // [!code highlight]
})
```
</Step>
</Steps>

```

## ## Usage

Once you've installed the plugin, you can start using the organization plugin to manage your organization's members and teams. The client plugin will provide you methods under the `organization` namespace. And the server `api` will provide you with the necessary endpoints to manage your organization and gives you easier way to call the functions on your own backend.

## ## Organization

### #### Create an organization

To create an organization, you need to provide:

- \* `name`: The name of the organization.
- \* `slug`: The slug of the organization.
- \* `logo`: The logo of the organization. (Optional)

```

```ts title="auth-client.ts"
await authClient.organization.create({
  name: "My Organization",
  slug: "my-org",
  logo: "https://example.com/logo.png"
})
```

```

### #### Restrict who can create an organization

By default, any user can create an organization. To restrict this, set the `allowUserToCreateOrganization` option to a function that returns a boolean, or directly to `true` or `false`.

```

```ts title="auth.ts"
import { betterAuth } from "better-auth"
import { organization } from "better-auth/plugins"

const auth = betterAuth({
  //...
  plugins: [
    organization({
      allowUserToCreateOrganization: async (user) => { // [!code highlight]
        const subscription = await getSubscription(user.id) // [!code highlight]
        return subscription.plan === "pro" // [!code highlight]
      } // [!code highlight]
    })
  ]
})
```

```

### #### Check if organization slug is taken

To check if an organization slug is taken or not you can use the `checkSlug` function provided by the client. The function takes an object with the following properties:

- \* `slug`: The slug of the organization.

```

```ts title="auth-client.ts"
await authClient.organization.checkSlug({
  slug: "my-org",
});
```

```



### #### Organization Creation Hooks

You can customize the organization creation process using hooks that run before and after an organization is created.

```
```ts title="auth.ts"
import { betterAuth } from "better-auth"
import { organization } from "better-auth/plugins"

export const auth = betterAuth({
  plugins: [
    organization({
      organizationCreation: {
        disabled: false, // Set to true to disable organization creation
        beforeCreate: async ({ organization, user }, request) => {
          // Run custom logic before organization is created
          // Optionally modify the organization data
          return {
            data: {
              ...organization,
              metadata: {
                customField: "value"
              }
            }
          }
        },
        afterCreate: async ({ organization, member, user }, request) => {
          // Run custom logic after organization is created
          // e.g., create default resources, send notifications
          await setupDefaultResources(organization.id)
        }
      }
    })
  ]
})
```
```

The `beforeCreate` hook runs before an organization is created. It receives:

- \* `organization`: The organization data (without ID)
- \* `user`: The user creating the organization
- \* `request`: The HTTP request object (optional)

Return an object with `data` property to modify the organization data that will be created.

The `afterCreate` hook runs after an organization is successfully created. It receives:

- \* `organization`: The created organization (with ID)
- \* `member`: The member record for the creator
- \* `user`: The user who created the organization
- \* `request`: The HTTP request object (optional)

### ### List User's Organizations

To list the organizations that a user is a member of, you can use `useListOrganizations` hook. It implements a reactive way to get the organizations that the user is a member of.

```
<Tabs items={["React", "Vue", "Svelte"]} defaultValue="React">
  <Tab value="React">
    ```tsx title="client.tsx"
    import { authClient } from "@/lib/auth-client"

    function App(){
      const { data: organizations } = authClient.useListOrganizations()
      return (
```

```

    <div>
      {organizations.map(org => <p>{org.name}</p>)}
    </div>
  )
}
...
</Tab>

<Tab value="Svelte">
  `` svelte title="page.svelte"
  <script lang="ts">
    import { authClient } from "$lib/auth-client";
    const organizations = authClient.useListOrganizations();
  </script>

  <h1>Organizations</h1>

  {#if $organizations.isPending}
    <p>Loading...</p>
  {:else if $organizations.data === null}
    <p>No organizations found.</p>
  {:else}
    <ul>
      {#each $organizations.data as organization}
        <li>{organization.name}</li>
      {/each}
    </ul>
  {/if}
  ``
</Tab>

<Tab value="Vue">
  `` vue title="organization.vue"
  <script lang="ts">
    export default {
      setup() {
        const organizations = authClient.useListOrganizations()
        return { organizations };
      }
    };
  </script>

  <template>
    <div>
      <h1>Organizations</h1>
      <div v-if="organizations.isPending">Loading...</div>
      <div v-else-if="organizations.data === null">No organizations found.</div>
      <ul v-else>
        <li v-for="organization in organizations.data" :key="organization.id">
          {{ organization.name }}
        </li>
      </ul>
    </div>
  </template>
  ``
</Tab>
</Tabs>

```

### ### Active Organization

Active organization is the workspace the user is currently working on. By default when the user is signed in the active organization is set to `null`. You can set the active organization to the user session.

<Callout type="info">

It's not always you want to persist the active organization in the session. You can manage the active organization in the

client side only. For example, multiple tabs can have different active organizations.  
</Callout>

#### #### Set Active Organization

You can set the active organization by calling the `organization.setActive` function. It'll set the active organization for the user session.

```
<Tabs items={["client", "server"]} defaultValue="client">
  <Tab value="client">
    `` `ts title="auth-client.ts"
    import { authClient } from "@lib/auth-client";

    await authClient.organization.setActive({
      organizationId: "organization-id"
    })

    // you can also use organizationSlug instead of organizationId
    await authClient.organization.setActive({
      organizationSlug: "organization-slug"
    })
    `` `

  </Tab>

  <Tab value="server">
    `` `ts title="api.ts"
    import { auth } from "@lib/auth";

    await auth.api.setActiveOrganization({
      headers: // pass the headers,
      body: {
        organizationSlug: "organization-slug"
      }
    })

    // you can also use organizationId instead of organizationSlug
    await auth.api.setActiveOrganization({
      headers: // pass the headers,
      body: {
        organizationId: "organization-id"
      }
    })
    `` `

  </Tab>
</Tabs>
```

To set active organization when a session is created you can use [database hooks](/docs/concepts/database#database-hooks).

```
`` `ts title="auth.ts"
export const auth = betterAuth({
  databaseHooks: {
    session: {
      create: {
        before: async(session)=>{
          const organization = await getActiveOrganization(session.userId)
          return {
            data: {
              ...session,
              activeOrganizationId: organization.id
            }
          }
        }
      }
    }
  }
})
```

```

    }
  })
  ...

```

#### #### Use Active Organization

To retrieve the active organization for the user, you can call the `useActiveOrganization` hook. It returns the active organization for the user. Whenever the active organization changes, the hook will re-evaluate and return the new active organization.

```

<Tabs items={['React', 'Vue', 'Svelte']}>
  <Tab value="React">
    `` `tsx title="client.tsx"
    import { authClient } from "@lib/auth-client"

    function App(){
      const { data: activeOrganization } = authClient.useActiveOrganization()
      return (
        <div>
          {activeOrganization ? <p>{activeOrganization.name}</p> : null}
        </div>
      )
    }
    ...
  </Tab>

  <Tab value="Svelte">
    `` `tsx title="client.tsx"
    <script lang="ts">
      import { authClient } from "$lib/auth-client";
      const activeOrganization = authClient.useActiveOrganization();
    </script>

    <h2>Active Organization</h2>

    {#if $activeOrganization.isPending}
    <p>Loading...</p>
    {:else if $activeOrganization.data === null}
    <p>No active organization found.</p>
    {:else}
    <p>{$activeOrganization.data.name}</p>
    {/if}
    ...
  </Tab>

  <Tab value="Vue">
    `` `vue title="organization.vue"
    <script lang="ts">
      export default {
        setup() {
          const activeOrganization = authClient.useActiveOrganization();
          return { activeOrganization };
        }
      };
    </script>

    <template>
      <div>
        <h2>Active organization</h2>
        <div v-if="activeOrganization.isPending">Loading...</div>
        <div v-else-if="activeOrganization.data === null">No active organization.</div>
        <div v-else>
          {{ activeOrganization.data.name }}
        </div>
      </div>
    </template>

```

```

</template>
...

</Tab>
</Tabs>

```

### #### Get Full Organization

To get the full details of an organization, you can use the `getFullOrganization` function provided by the client. The function takes an object with the following properties:

- \* `organizationId`: The ID of the organization. (Optional) - By default, it will use the active organization.
- \* `organizationSlug`: The slug of the organization. (Optional) - To get the organization by slug.

```

<Tabs items=[["client", "server"]]>
  <Tab value="client">
    `` ts title="auth-client.ts"
    const organization = await authClient.organization.getFullOrganization({
      query: { organizationId: "organization-id" } // optional, by default it will use the active organization
    })
    // you can also use organizationSlug instead of organizationId
    const organization = await authClient.organization.getFullOrganization({
      query: { organizationSlug: "organization-slug" }
    })
    ...
  </Tab>

  <Tab value="server">
    `` ts title="api.ts"
    import { auth } from "@auth";

    auth.api.getFullOrganization({
      headers: // pass the headers
    })

    // you can also use organizationSlug instead of organizationId
    auth.api.getFullOrganization({
      headers: // pass the headers,
      query: {
        organizationSlug: "organization-slug"
      }
    })
    ...
  </Tab>
</Tabs>

```

### #### Update Organization

To update organization info, you can use `organization.update`

```

`` ts
await authClient.organization.update({
  data: {
    name: "updated-name",
    logo: "new-logo.url",
    metadata: {
      customerId: "test"
    },
    slug: "updated-slug"
  },
  organizationId: 'org-id' //defaults to the current active organization
})
...

```

### #### Delete Organization

To remove user owned organization, you can use `organization.delete`

```
```ts title="org.ts"
await authClient.organization.delete({
  organizationId: "test"
});
```
```

If the user has the necessary permissions (by default: role is owner) in the specified organization, all members, invitations and organization information will be removed.

You can configure how organization deletion is handled through `organizationDeletion` option:

```
```ts
const auth = betterAuth({
  plugins: [
    organization({
      organizationDeletion: {
        disabled: true, //to disable it altogether
        beforeDelete: async (data, request) => {
          // a callback to run before deleting org
        },
        afterDelete: async (data, request) => {
          // a callback to run after deleting org
        },
      },
    },
  ],
});
```
```

## ## Invitations

To add a member to an organization, we first need to send an invitation to the user. The user will receive an email/sms with the invitation link. Once the user accepts the invitation, they will be added to the organization.

### ### Setup Invitation Email

For member invitation to work we first need to provide `sendInvitationEmail` to the `better-auth` instance. This function is responsible for sending the invitation email to the user.

You'll need to construct and send the invitation link to the user. The link should include the invitation ID, which will be used with the `acceptInvitation` function when the user clicks on it.

```
```ts title="auth.ts"
import { betterAuth } from "better-auth"
import { organization } from "better-auth/plugins"
import { sendOrganizationInvitation } from "../email"
export const auth = betterAuth({
  plugins: [
    organization({
      async sendInvitationEmail(data) {
        const inviteLink = `https://example.com/accept-invitation/${data.id}`
        sendOrganizationInvitation({
          email: data.email,
          invitedByUsername: data.inviter.user.name,
          invitedByEmail: data.inviter.user.email,
          teamName: data.organization.name,
          inviteLink
        })
      },
    },
  ],
});
```
```

### #### Send Invitation

To invite users to an organization, you can use the `invite` function provided by the client. The `invite` function takes an object with the following properties:

- \* `email`: The email address of the user.
- \* `role`: The role of the user in the organization. It can be `admin`, `member`, or `guest`.
- \* `organizationId`: The ID of the organization. this is optional by default it will use the active organization. (Optional)

```
```ts title="invitation.ts"
await authClient.organization.inviteMember({
  email: "test@email.com",
  role: "admin", //this can also be an array for multiple roles (e.g. ["admin", "sale"])
})
```
```

#### <Callout>

- \* If the user is already a member of the organization, the invitation will be canceled.
- \* If the user is already invited to the organization, unless `resend` is set to `true`, the invitation will not be sent again.
- \* If `cancelPendingInvitationsOnReInvite` is set to `true`, the invitation will be canceled if the user is already invited to the organization and a new invitation is sent.

#### </Callout>

### #### Accept Invitation

When a user receives an invitation email, they can click on the invitation link to accept the invitation. The invitation link should include the invitation ID, which will be used to accept the invitation.

Make sure to call the `acceptInvitation` function after the user is logged in.

```
```ts title="auth-client.ts"
await authClient.organization.acceptInvitation({
  invitationId: "invitation-id"
})
```
```

### #### Update Invitation Status

To update the status of invitation you can use the `acceptInvitation`, `cancelInvitation`, `rejectInvitation` functions provided by the client. The functions take the invitation ID as an argument.

```
```ts title="auth-client.ts"
//cancel invitation
await authClient.organization.cancelInvitation({
  invitationId: "invitation-id"
})

//reject invitation (needs to be called when the user who received the invitation is logged in)
await authClient.organization.rejectInvitation({
  invitationId: "invitation-id"
})
```
```

### #### Get Invitation

To get an invitation you can use the `getInvitation` function provided by the client. You need to provide the invitation ID as a query parameter.

```
```ts title="auth-client.ts"
await authClient.organization.getInvitation({
  query: {
    id: params.id
  }
})
```
```

```

### #### List Invitations

To list all invitations you can use the `listInvitations` function provided by the client.

```
```ts title="auth-client.ts"
const invitations = await authClient.organization.listInvitations({
  query: {
    organizationId: "organization-id" // optional, by default it will use the active organization
  }
})
```
```

### ## Members

#### #### Remove Member

To remove you can use `organization.removeMember`

```
```ts title="auth-client.ts"
//remove member
await authClient.organization.removeMember({
  memberIdOrEmail: "member-id", // this can also be the email of the member
  organizationId: "organization-id" // optional, by default it will use the active organization
})
```
```

#### #### Update Member Role

To update the role of a member in an organization, you can use the `organization.updateMemberRole`. If the user has the permission to update the role of the member, the role will be updated.

```
```ts title="auth-client.ts"
await authClient.organization.updateMemberRole({
  memberId: "member-id",
  role: "admin" // this can also be an array for multiple roles (e.g. ["admin", "sale"])
})
```
```

#### #### Get Active Member

To get the current member of the organization you can use the `organization.getActiveMember` function. This function will return the current active member.

```
```ts title="auth-client.ts"
const member = await authClient.organization.getActiveMember()
```
```

#### #### Add Member

If you want to add a member directly to an organization without sending an invitation, you can use the `addMember` function which can only be invoked on the server.

```
```ts title="api.ts"
import { auth } from "@auth";

await auth.api.addMember({
  body: {
    userId: "user-id",
    organizationId: "organization-id",
    role: "admin", // this can also be an array for multiple roles (e.g. ["admin", "sale"])
    teamId: "team-id" // Optionally specify a teamId to add the member to a team. (requires teams to be enabled)
  }
})
```
```



```

### ### Leave Organization

To leave organization you can use `organization.leave` function. This function will remove the current user from the organization.

```
```ts title="auth-client.ts"
await authClient.organization.leave({
  organizationId: "organization-id"
})
```
```

## ## Access Control

The organization plugin provides a very flexible access control system. You can control the access of the user based on the role they have in the organization. You can define your own set of permissions based on the role of the user.

### ### Roles

By default, there are three roles in the organization:

`owner`: The user who created the organization by default. The owner has full control over the organization and can perform any action.

`admin`: Users with the admin role have full control over the organization except for deleting the organization or changing the owner.

`member`: Users with the member role have limited control over the organization. They can create projects, invite users, and manage projects they have created.

<Callout>

A user can have multiple roles. Multiple roles are stored as string separated by comma (",").

</Callout>

### ### Permissions

By default, there are three resources, and these have two to three actions.

**\*\*organization\*\*:**

`update` `delete`

**\*\*member\*\*:**

`create` `update` `delete`

**\*\*invitation\*\*:**

`create` `cancel`

The owner has full control over all the resources and actions. The admin has full control over all the resources except for deleting the organization or changing the owner. The member has no control over any of those actions other than reading the data.

### ### Custom Permissions

The plugin provides an easy way to define your own set of permissions for each role.

<Steps>

<Step>

#### ##### Create Access Control

You first need to create access controller by calling `createAccessControl` function and passing the statement object. The statement object should have the resource name as the key and the array of actions as the value.

```

` `` ts title="permissions.ts"
import { createAccessControl } from "better-auth/plugins/access";

/**
 * make sure to use `as const` so typescript can infer the type correctly
 */
const statement = { // [!code highlight]
  project: ["create", "share", "update", "delete"], // [!code highlight]
} as const; // [!code highlight]

const ac = createAccessControl(statement); // [!code highlight]
` ``
</Step>

```

<Step>  
**#### Create Roles**

Once you have created the access controller you can create roles with the permissions you have defined.

```

` `` ts title="permissions.ts"
import { createAccessControl } from "better-auth/plugins/access";

const statement = {
  project: ["create", "share", "update", "delete"],
} as const;

const ac = createAccessControl(statement);

const member = ac.newRole({ // [!code highlight]
  project: ["create"], // [!code highlight]
}); // [!code highlight]

const admin = ac.newRole({ // [!code highlight]
  project: ["create", "update"], // [!code highlight]
}); // [!code highlight]

const owner = ac.newRole({ // [!code highlight]
  project: ["create", "update", "delete"], // [!code highlight]
}); // [!code highlight]

const myCustomRole = ac.newRole({ // [!code highlight]
  project: ["create", "update", "delete"], // [!code highlight]
  organization: ["update"], // [!code highlight]
}); // [!code highlight]
` ``

```

When you create custom roles for existing roles, the predefined permissions for those roles will be overridden. To add the existing permissions to the custom role, you need to import `defaultStatements` and merge it with your new statement, plus merge the roles' permissions set with the default roles.

```

` `` ts title="permissions.ts"
import { createAccessControl } from "better-auth/plugins/access";
import { defaultStatements, adminAc } from 'better-auth/plugins/organization/access'

const statement = {
  ...defaultStatements, // [!code highlight]
  project: ["create", "share", "update", "delete"],
} as const;

const ac = createAccessControl(statement);

const admin = ac.newRole({
  project: ["create", "update"],
  ...adminAc.statements, // [!code highlight]

```

```
});
```
</Step>

```

```
<Step>
#### Pass Roles to the Plugin

```

Once you have created the roles you can pass them to the organization plugin both on the client and the server.

```
```ts title="auth.ts"
import { betterAuth } from "better-auth"
import { organization } from "better-auth/plugins"
import { ac, owner, admin, member } from "@auth/permissions"

export const auth = betterAuth({
  plugins: [
    organization({
      ac,
      roles: {
        owner,
        admin,
        member,
        myCustomRole
      }
    })
  ],
});
```

```

You also need to pass the access controller and the roles to the client plugin.

```
```ts title="auth-client"
import { createAuthClient } from "better-auth/client"
import { organizationClient } from "better-auth/client/plugins"
import { ac, owner, admin, member, myCustomRole } from "@auth/permissions"

export const authClient = createAuthClient({
  plugins: [
    organizationClient({
      ac,
      roles: {
        owner,
        admin,
        member,
        myCustomRole
      }
    })
  ]
})
```
</Step>
</Steps>

```

### ### Access Control Usage

**\*\*Has Permission\*\*:**

You can use the `hasPermission` action provided by the `api` to check the permission of the user.

```
```ts title="api.ts"
import { auth } from "@auth";

await auth.api.hasPermission({
  headers: await headers(),
  body: {

```

```

    permissions: {
      project: ["create"] // This must match the structure in your access control
    }
  }
});

```

```

// You can also check multiple resource permissions at the same time
await auth.api.hasPermission({
  headers: await headers(),
  body: {
    permissions: {
      project: ["create"], // This must match the structure in your access control
      sale: ["create"]
    }
  }
});
```

```

If you want to check the permission of the user on the client from the server you can use the `hasPermission` function provided by the client.

```

```ts title="auth-client.ts"
const canCreateProject = await authClient.organization.hasPermission({
  permissions: {
    project: ["create"]
  }
})

// You can also check multiple resource permissions at the same time
const canCreateProjectAndCreateSale = await authClient.organization.hasPermission({
  permissions: {
    project: ["create"],
    sale: ["create"]
  }
})
```

```

### **\*\*Check Role Permission\*\*:**

Once you have defined the roles and permissions to avoid checking the permission from the server you can use the `checkRolePermission` function provided by the client.

```

```ts title="auth-client.ts"
const canCreateProject = authClient.organization.checkRolePermission({
  permissions: {
    organization: ["delete"],
  },
  role: "admin",
});

// You can also check multiple resource permissions at the same time
const canCreateProjectAndCreateSale = authClient.organization.checkRolePermission({
  permissions: {
    organization: ["delete"],
    member: ["delete"]
  },
  role: "admin",
});
```

```

### **### Teams**

Teams allow you to group members within an organization. The teams feature provides additional organization structure and can be used to manage permissions at a more granular level.

### ### Enabling Teams

To enable teams, pass the `teams` configuration option to both server and client plugins:

```

```ts title="auth.ts"
import { betterAuth } from "better-auth"
import { organization } from "better-auth/plugins"

export const auth = betterAuth({
  plugins: [
    organization({
      teams: {
        enabled: true,
        maximumTeams: 10, // Optional: limit teams per organization
        allowRemovingAllTeams: false // Optional: prevent removing the last team
      }
    })
  ]
})
```

```

```

```ts title="auth-client.ts"
import { createAuthClient } from "better-auth/client"
import { organizationClient } from "better-auth/client/plugins"

export const authClient = createAuthClient({
  plugins: [
    organizationClient({
      teams: {
        enabled: true
      }
    })
  ]
})
```

```

### ### Managing Teams

#### #### Create Team

Create a new team within an organization:

```

```ts
const team = await authClient.organization.createTeam({
  name: "Development Team",
  organizationId: "org-id" // Optional: defaults to active organization
})
```

```

#### #### List Teams

Get all teams in an organization:

```

```ts
const teams = await authClient.organization.listTeams({
  query: {
    organizationId: org.id, // Optional: defaults to active organization
  },
});
```

```

#### #### Update Team

Update a team's details:

```

```ts
const updatedTeam = await authClient.organization.updateTeam({
  teamId: "team-id",
  data: {
    name: "Updated Team Name"
  }
})
```

```

#### #### Remove Team

Delete a team from an organization:

```

```ts
await authClient.organization.removeTeam({
  teamId: "team-id",
  organizationId: "org-id" // Optional: defaults to active organization
})
```

```

#### ### Team Permissions

Teams follow the organization's permission system. To manage teams, users need the following permissions:

- \* `team:create` - Create new teams
- \* `team:update` - Update team details
- \* `team:delete` - Remove teams

By default:

- \* Organization owners and admins can manage teams
- \* Regular members cannot create, update, or delete teams

#### ### Team Configuration Options

The teams feature supports several configuration options:

```

* `maximumTeams`: Limit the number of teams per organization
```ts
teams: {
  enabled: true,
  maximumTeams: 10 // Fixed number
  // OR
  maximumTeams: async ({ organizationId, session }, request) => {
    // Dynamic limit based on organization plan
    const plan = await getPlan(organizationId)
    return plan === 'pro' ? 20 : 5
  },
  maximumMembersPerTeam: 10 // Fixed number
  // OR
  maximumMembersPerTeam: async ({ teamId, session, organizationId }, request) => {
    // Dynamic limit based on team plan
    const plan = await getPlan(organizationId, teamId)
    return plan === 'pro' ? 50 : 10
  },
}
```

```

```

* `allowRemovingAllTeams`: Control whether the last team can be removed
```ts
teams: {
  enabled: true,
  allowRemovingAllTeams: false // Prevent removing the last team
}
```

```

### Team Members

When inviting members to an organization, you can specify a team:

```
```ts
await authClient.organization.inviteMember({
  email: "user@example.com",
  role: "member",
  teamId: "team-id"
})
```
```

The invited member will be added to the specified team upon accepting the invitation.

### Database Schema

When teams are enabled, a new `team` table is added with the following structure:

```
<DatabaseTable
  fields=[
    {
      name: "id",
      type: "string",
      description: "Unique identifier for each team",
      isPrimaryKey: true
    },
    {
      name: "name",
      type: "string",
      description: "The name of the team"
    },
    {
      name: "organizationId",
      type: "string",
      description: "The ID of the organization",
      isForeignKey: true
    },
    {
      name: "createdAt",
      type: "Date",
      description: "Timestamp of when the team was created"
    },
    {
      name: "updatedAt",
      type: "Date",
      description: "Timestamp of when the team was last updated"
    }
  ]
/>
```

### Schema

The organization plugin adds the following tables to the database:

#### Organization

Table Name: `organization`

```
<DatabaseTable
  fields=[
    {
      name: "id",
      type: "string",
      description: "Unique identifier for each organization",
```

```

    isPrimaryKey: true
  },
  {
    name: "name",
    type: "string",
    description: "The name of the organization"
  },
  {
    name: "slug",
    type: "string",
    description: "The slug of the organization"
  },
  {
    name: "logo",
    type: "string",
    description: "The logo of the organization",
    isOptional: true
  },
  {
    name: "metadata",
    type: "string",
    description: "Additional metadata for the organization",
    isOptional: true
  },
  {
    name: "createdAt",
    type: "Date",
    description: "Timestamp of when the organization was created"
  },
  ]
}

```

#### ### Member

Table Name: `member`

```

<DatabaseTable
  fields={[
    {
      name: "id",
      type: "string",
      description: "Unique identifier for each member",
      isPrimaryKey: true
    },
    {
      name: "userId",
      type: "string",
      description: "The ID of the user",
      isForeignKey: true
    },
    {
      name: "organizationId",
      type: "string",
      description: "The ID of the organization",
      isForeignKey: true
    },
    {
      name: "role",
      type: "string",
      description: "The role of the user in the organization"
    },
    {
      name: "createdAt",
      type: "Date",
      description: "Timestamp of when the member was added to the organization"
    }
  ]
}

```



```

    },
  ]}
/>

```

### #### Invitation

Table Name: `invitation`

```

<DatabaseTable
  fields=[
    {
      name: "id",
      type: "string",
      description: "Unique identifier for each invitation",
      isPrimaryKey: true
    },
    {
      name: "email",
      type: "string",
      description: "The email address of the user"
    },
    {
      name: "inviterId",
      type: "string",
      description: "The ID of the inviter",
      isForeignKey: true
    },
    {
      name: "organizationId",
      type: "string",
      description: "The ID of the organization",
      isForeignKey: true
    },
    {
      name: "role",
      type: "string",
      description: "The role of the user in the organization"
    },
    {
      name: "status",
      type: "string",
      description: "The status of the invitation"
    },
    {
      name: "expiresAt",
      type: "Date",
      description: "Timestamp of when the invitation expires"
    },
    {
      name: "createdAt",
      type: "Date",
      description: "Timestamp of when the invitation was created"
    },
  ]}
/>

```

### #### Session

Table Name: `session`

You need to add one more field to the session table to store the active organization ID.

```

<DatabaseTable
  fields=[
    {

```

```

    name: "activeOrganizationId",
    type: "string",
    description: "The ID of the active organization",
    isOptional: true
  },
]}
/>

```

### #### Teams (optional)

Table Name: `team`

```

<DatabaseTable
  fields={[
    {
      name: "id",
      type: "string",
      description: "Unique identifier for each team",
      isPrimaryKey: true
    },
    {
      name: "name",
      type: "string",
      description: "The name of the team"
    },
    {
      name: "organizationId",
      type: "string",
      description: "The ID of the organization",
      isForeignKey: true
    },
    {
      name: "createdAt",
      type: "Date",
      description: "Timestamp of when the team was created"
    },
    {
      name: "updatedAt",
      type: "Date",
      isOptional: true,
      description: "Timestamp of when the team was created"
    },
  ]}
/>

```

Table Name: `member`

```

<DatabaseTable
  fields={[
    {
      name: "teamId",
      type: "string",
      description: "The ID of the team",
      isOptional: true
    },
  ]}
/>

```

Table Name: `invitation`

```

<DatabaseTable
  fields={[
    {
      name: "teamId",
      type: "string",

```

```

    description: "The ID of the team",
    isOptional: true
  },
]}
/>

```

### ### Customizing the Schema

To change the schema table name or fields, you can pass `schema` option to the organization plugin.

```

```ts title="auth.ts"
const auth = betterAuth({
  plugins: [organization({
    schema: {
      organization: {
        modelName: "organizations", //map the organization table to organizations
        fields: {
          name: "title" //map the name field to title
        }
      }
    }
  })]
})
```

```

### ## Options

**\*\*allowUserToCreateOrganization\*\***: `boolean` | `((user: User) => Promise<boolean> | boolean)` - A function that determines whether a user can create an organization. By default, it's `true`. You can set it to `false` to restrict users from creating organizations.

**\*\*organizationLimit\*\***: `number` | `((user: User) => Promise<boolean> | boolean)` - The maximum number of organizations allowed for a user. By default, it's `5`. You can set it to any number you want or a function that returns a boolean.

**\*\*creatorRole\*\***: `admin | owner` - The role of the user who creates the organization. By default, it's `owner`. You can set it to `admin`.

**\*\*membershipLimit\*\***: `number` - The maximum number of members allowed in an organization. By default, it's `100`. You can set it to any number you want.

**\*\*sendInvitationEmail\*\***: `async (data) => Promise<void>` - A function that sends an invitation email to the user.

**\*\*invitationExpiresIn\*\***: `number` - How long the invitation link is valid for in seconds. By default, it's 48 hours (2 days).

**\*\*cancelPendingInvitationsOnReInvite\*\***: `boolean` - Whether to cancel pending invitations if the user is already invited to the organization. By default, it's `true`.

**\*\*invitationLimit\*\***: `number` | `((user: User) => Promise<boolean> | boolean)` - The maximum number of invitations allowed for a user. By default, it's `100`. You can set it to any number you want or a function that returns a boolean.

```

file: ./content/docs/plugins/passkey.mdx
meta: {
  "title": "Passkey",
  "description": "Passkey"
}

```

Passkeys are a secure, passwordless authentication method using cryptographic key pairs, supported by WebAuthn and FIDO2 standards in web browsers. They replace passwords with unique key pairs: a private key stored on the user's device and a public key shared with the website. Users can log in using biometrics, PINs, or security keys, providing strong, phishing-resistant authentication without traditional passwords.

The passkey plugin implementation is powered by [SimpleWebAuthn](https://simplewebauthn.dev/) behind the scenes.

### ## Installation

<Steps>

<Step>

#### Add the plugin to your auth config

To add the passkey plugin to your auth config, you need to import the plugin and pass it to the `plugins` option of the auth instance.

**\*\*Options\*\***

`rpID`: A unique identifier for your website. 'localhost' is okay for local dev

`rpName`: Human-readable title for your website

`origin`: The URL at which registrations and authentications should occur. `http://localhost` and `http://localhost:PORT` are also valid. Do **\*\*NOT\*\*** include any trailing /

`authenticatorSelection`: Allows customization of WebAuthn authenticator selection criteria. Leave unspecified for default settings.

\* `authenticatorAttachment`: Specifies the type of authenticator

\* `platform`: Authenticator is attached to the platform (e.g., fingerprint reader)

\* `cross-platform`: Authenticator is not attached to the platform (e.g., security key)

\* Default: `not set` (both platform and cross-platform allowed, with platform preferred)

\* `residentKey`: Determines credential storage behavior.

\* `required`: User **MUST** store credentials on the authenticator (highest security)

\* `preferred`: Encourages credential storage but not mandatory

\* `discouraged`: No credential storage required (fastest experience)

\* Default: `preferred`

\* `userVerification`: Controls biometric/PIN verification during authentication:

\* `required`: User **MUST** verify identity (highest security)

\* `preferred`: Verification encouraged but not mandatory

\* `discouraged`: No verification required (fastest experience)

\* Default: `preferred`

```
```ts title="auth.ts"
```

```
import { betterAuth } from "better-auth"
```

```
import { passkey } from "better-auth/plugins/passkey" // [!code highlight]
```

```
export const auth = betterAuth({
  plugins: [ // [!code highlight]
    passkey(), // [!code highlight]
  ], // [!code highlight]
})
```
```

</Step>

<Step>

#### Migrate the database

Run the migration or generate the schema to add the necessary fields and tables to the database.

```
<Tabs items=["migrate", "generate"]>
```

```
<Tab value="migrate">
```

```
```bash
```

```
npx @better-auth/cli migrate
```

```
```
```

```
</Tab>
```

```
<Tab value="generate">
```

```
```bash
```

```
npx @better-auth/cli generate
```

```
```
```

```
</Tab>
```

```
</Tabs>
```

See the [Schema](#schema) section to add the fields manually.

</Step>

<Step>  
#### Add the client plugin

```
```ts title="auth-client.ts"
import { createAuthClient } from "better-auth/client"
import { passkeyClient } from "better-auth/client/plugins"

export const authClient = createAuthClient({
  plugins: [ // [!code highlight]
    passkeyClient() // [!code highlight]
  ] // [!code highlight]
})
```
```

</Step>

</Steps>

## Usage

#### Add/Register a passkey

To add or register a passkey make sure a user is authenticated and then call the `passkey.addPasskey` function provided by the client.

```
```ts
// Default behavior allows both platform and cross-platform passkeys
const { data, error } = await authClient.passkey.addPasskey();
```
```

This will prompt the user to register a passkey. And it'll add the passkey to the user's account.

You can also specify the type of authenticator you want to register. The `authenticatorAttachment` can be either `platform` or `cross-platform`.

```
```ts
// Register a cross-platform passkey showing only a QR code
// for the user to scan as well as the option to plug in a security key
const { data, error } = await authClient.passkey.addPasskey({
  authenticatorAttachment: 'cross-platform'
});
```
```

#### Sign in with a passkey

To sign in with a passkey you can use the passkeySignIn method. This will prompt the user to sign in with their passkey.

Signin method accepts:

`autoFill`: Browser autofill, a.k.a. Conditional UI. [read more]  
(<https://simplewebauthn.dev/docs/packages/browser#browser-autofill-aka-conditional-ui>)

`email`: The email of the user to sign in.

`fetchOptions`: Fetch options to pass to the fetch request.

```
```ts
const data = await authClient.signIn.passkey();
```
```

#### Conditional UI

The plugin supports conditional UI, which allows the browser to autofill the passkey if the user has already registered a

passkey.

There are two requirements for conditional UI to work:

<Steps>

<Step>

#### Update input fields

Add the `autocomplete` attribute with the value `webauthn` to your input fields. You can add this attribute to multiple input fields, but at least one is required for conditional UI to work.

The `webauthn` value should also be the last entry of the `autocomplete` attribute.

```
```html
```

```
<label for="name">Username:</label>
```

```
<input type="text" name="name" autocomplete="username webauthn">
```

```
<label for="password">Password:</label>
```

```
<input type="password" name="password" autocomplete="current-password webauthn">
```

```
```
```

</Step>

<Step>

#### Preload the passkeys

When your component mounts, you can preload the user's passkeys by calling the `authClient.signIn.passkey` method with the `autoFill` option set to `true`.

To prevent unnecessary calls, we will also add a check to see if the browser supports conditional UI.

```
<Tabs items={["React"]}>
```

```
<Tab value="React">
```

```
```ts
```

```
useEffect(() => {
```

```
  if (!PublicKeyCredential.isConditionalMediationAvailable ||
```

```
    !PublicKeyCredential.isConditionalMediationAvailable()) {
```

```
    return;
```

```
  }
```

```
  void authClient.signIn.passkey({ autoFill: true })
```

```
}, [])
```

```
```
```

```
</Tab>
```

```
</Tabs>
```

</Step>

</Steps>

Depending on the browser, a prompt will appear to autofill the passkey. If the user has multiple passkeys, they can select the one they want to use.

Some browsers also require the user to first interact with the input field before the autofill prompt appears.

### ### Debugging

To test your passkey implementation you can use [emulated authenticators]

(<https://developer.chrome.com/docs/devtools/webauthn>). This way you can test the registration and sign-in process without even owning a physical device.

### ## Schema

The plugin require a new table in the database to store passkey data.

Table Name: `passkey`

<DatabaseTable

fields={

```

{
  name: "id",
  type: "string",
  description: "Unique identifier for each passkey",
  isPrimaryKey: true
},
{
  name: "name",
  type: "string",
  description: "The name of the passkey",
  isOptional: true
},
{
  name: "publicKey",
  type: "string",
  description: "The public key of the passkey",
},
{
  name: "userId",
  type: "string",
  description: "The ID of the user",
  isForeignKey: true
},
{
  name: "credentialID",
  type: "string",
  description: "The unique identifier of the registered credential",
},
{
  name: "counter",
  type: "number",
  description: "The counter of the passkey",
},
{
  name: "deviceType",
  type: "string",
  description: "The type of device used to register the passkey",
},
{
  name: "backedUp",
  type: "boolean",
  description: "Whether the passkey is backed up",
},
{
  name: "transports",
  type: "string",
  description: "The transports used to register the passkey",
},
{
  name: "createdAt",
  type: "Date",
  description: "The time when the passkey was created",
},
{
  name: "aaguid",
  type: "string",
  description: "Authenticator's Attestation GUID indicating the type of the authenticator",
  isOptional: true
},
}
]
/>

```

### ## Options

**\*\*rpID\*\***: A unique identifier for your website. 'localhost' is okay for local dev.

**\*\*rpName\*\***: Human-readable title for your website.

**\*\*origin\*\***: The URL at which registrations and authentications should occur. `http://localhost` and `http://localhost:PORT` are also valid. Do NOT include any trailing `/`.

**\*\*authenticatorSelection\*\***: Allows customization of WebAuthn authenticator selection criteria. When unspecified, both platform and cross-platform authenticators are allowed with `preferred` settings for `residentKey` and `userVerification`.

**\*\*aaguid\*\***: (optional) Authenticator Attestation GUID. This is a unique identifier for the passkey provider (device or authenticator type) and can be used to identify the type of passkey device used during registration or authentication.

file: ./content/docs/plugins/phone-number.mdx

```
meta: {
  "title": "Phone Number",
  "description": "Phone number plugin"
}
```

The phone number plugin extends the authentication system by allowing users to sign in and sign up using their phone number. It includes OTP (One-Time Password) functionality to verify phone numbers.

## ## Installation

### <Steps>

#### <Step>

#### Add Plugin to the server

```
```ts title="auth.ts"
import { betterAuth } from "better-auth"
import { phoneNumber } from "better-auth/plugins"

const auth = betterAuth({
  plugins: [
    phoneNumber({ // [!code highlight]
      sendOTP: ({ phoneNumber, code }, request) => { // [!code highlight]
        // Implement sending OTP code via SMS // [!code highlight]
      } // [!code highlight]
    }) // [!code highlight]
  ]
})
```
```

#### </Step>

#### <Step>

#### Migrate the database

Run the migration or generate the schema to add the necessary fields and tables to the database.

```
<Tabs items=["migrate", "generate"]>
<Tab value="migrate">
  ```bash
  npx @better-auth/cli migrate
  ```
</Tab>

<Tab value="generate">
  ```bash
  npx @better-auth/cli generate
  ```
</Tab>
</Tabs>
```

See the [Schema](#schema) section to add the fields manually.



</Step>

<Step>

#### Add the client plugin

```
```ts title="auth-client.ts"
import { createAuthClient } from "better-auth/client"
import { phoneNumberClient } from "better-auth/client/plugins"
```

```
const authClient = createAuthClient({
  plugins: [ // [!code highlight]
    phoneNumberClient() // [!code highlight]
  ] // [!code highlight]
})
```
```

</Step>

</Steps>

## ## Usage

### #### Send OTP for Verification

To send an OTP to a user's phone number for verification, you can use the `sendVerificationCode` endpoint.

```
```ts title="auth-client.ts"
await authClient.phoneNumber.sendOtp({
  phoneNumber: "+1234567890"
})
```
```

### #### Verify Phone Number

After the OTP is sent, users can verify their phone number by providing the code.

```
```ts title="auth-client.ts"
const isVerified = await authClient.phoneNumber.verify({
  phoneNumber: "+1234567890",
  code: "123456"
})
```
```

When the phone number is verified, the `phoneNumberVerified` field in the user table is set to `true`. If `disableSession` is not set to `true`, a session is created for the user. Additionally, if `callbackOnVerification` is provided, it will be called.

### #### Allow Sign-Up with Phone Number

To allow users to sign up using their phone number, you can pass `signUpOnVerification` option to your plugin configuration. It requires you to pass `getTempEmail` function to generate a temporary email for the user.

```
```ts title="auth.ts"
export const auth = betterAuth({
  plugins: [
    phoneNumber({
      sendOTP: ({ phoneNumber, code }, request) => {
        // Implement sending OTP code via SMS
      },
      signUpOnVerification: {
        getTempEmail: (phoneNumber) => {
          return `${phoneNumber}@my-site.com`
        },
        //optionally, you can also pass `getTempName` function to generate a temporary name for the user
        getTempName: (phoneNumber) => {
          return phoneNumber //by default, it will use the phone number as the name
        }
      }
    })
  ]
})
```
```

```

    })
  ]
})
```

```

### #### Sign In with Phone Number

In addition to signing in a user using send-verify flow, you can also use phone number as an identifier and sign in a user using phone number and password.

```

```ts
await authClient.signIn.phoneNumber({
  phoneNumber: "+123456789",
  password: "password",
  rememberMe: true //optional defaults to true
})
```

```

### #### Update Phone Number

Updating phone number uses the same process as verifying a phone number. The user will receive an OTP code to verify the new phone number.

```

```ts title="auth-client.ts"
await authClient.phoneNumber.sendOtp({
  phoneNumber: "+1234567890" // New phone number
})
```

```

Then verify the new phone number with the OTP code.

```

```ts title="auth-client.ts"
const isVerified = await authClient.phoneNumber.verify({
  phoneNumber: "+1234567890",
  code: "123456",
  updatePhoneNumber: true // Set to true to update the phone number
})
```

```

If a user session exist the phone number will be updated automatically.

### #### Disable Session Creation

By default, the plugin creates a session for the user after verifying the phone number. You can disable this behavior by passing `disableSession: true` to the `verify` method.

```

```ts title="auth-client.ts"
const isVerified = await authClient.phoneNumber.verify({
  phoneNumber: "+1234567890",
  code: "123456",
  disableSession: true
})
```

```

### #### Request Password Reset

To initiate a request password reset flow using `phoneNumber`, you can start by calling `requestPasswordReset` on the client to send an OTP code to the user's phone number.

```

```ts title="auth-client.ts"
await authClient.phoneNumber.requestPasswordReset({
  phoneNumber: "+1234567890"
})
```

```

Then, you can reset the password by calling `resetPassword` on the client with the OTP code and the new password.

```
```ts title="auth-client.ts"
const isVerified = await authClient.phoneNumber.resetPassword({
  otp: "123456", // OTP code sent to the user's phone number
  phoneNumber: "+1234567890",
  newPassword: "newPassword"
})
```
```

### ## Options

- \* `otpLength`: The length of the OTP code to be generated. Default is `6`.
- \* `sendOTP`: A function that sends the OTP code to the user's phone number. It takes the phone number and the OTP code as arguments.
- \* `expiresIn`: The time in seconds after which the OTP code expires. Default is `300` seconds.
- \* `callbackOnVerification`: A function that is called after the phone number is verified. It takes the phone number and the user object as the first argument and a request object as the second argument.

```
```ts
export const auth = betterAuth({
  plugins: [
    phoneNumber({
      sendOTP: ({ phoneNumber, code }, request) => {
        // Implement sending OTP code via SMS
      },
      callbackOnVerification: async ({ phoneNumber, user }, request) => {
        // Implement callback after phone number verification
      }
    })
  ]
})
```
```

\* `sendPasswordResetOTP`: A function that sends the OTP code to the user's phone number for password reset. It takes the phone number and the OTP code as arguments.

\* `phoneNumberValidator`: A custom function to validate the phone number. It takes the phone number as an argument and returns a boolean indicating whether the phone number is valid.

\* `signUpOnVerification`: An object with the following properties:

- \* `getTempEmail`: A function that generates a temporary email for the user. It takes the phone number as an argument and returns the temporary email.
- \* `getTempName`: A function that generates a temporary name for the user. It takes the phone number as an argument and returns the temporary name.

\* `requireVerification`: When enabled, users cannot sign in with their phone number until it has been verified. If an unverified user attempts to sign in, the server will respond with a 401 error (PHONE\\_NUMBER\\_NOT\\_VERIFIED) and automatically trigger an OTP send to start the verification process.

### ## Schema

The plugin requires 2 fields to be added to the user table

#### ### User Table

```
<DatabaseTable
  fields={[
    {
      name: "phoneNumber",
      type: "string",
      description: "The phone number of the user",
      isUnique: true,
      isOptional: true
    },
  ]}
```

```
{
  name: "phoneNumberVerified",
  type: "boolean",
  description: "Whether the phone number is verified or not",
  defaultValue: false,
  isOptional: true
},
]]
/>
```

### ### OTP Verification Attempts

The phone number plugin includes a built-in protection against brute force attacks by limiting the number of verification attempts for each OTP code.

```
```typescript
phoneNumber({
  allowedAttempts: 3, // default is 3
  // ... other options
})
```
```

When a user exceeds the allowed number of verification attempts:

- \* The OTP code is automatically deleted
- \* Further verification attempts will return a 403 (Forbidden) status with "Too many attempts" message
- \* The user will need to request a new OTP code to continue

Example error response after exceeding attempts:

```
```json
{
  "error": {
    "status": 403,
    "message": "Too many attempts"
  }
}
```
```

<Callout type="warning">

When receiving a 403 status, prompt the user to request a new OTP code

</Callout>

file: ./content/docs/plugins/polar.mdx

```
meta: {
  "title": "Polar",
  "description": "Better Auth Plugin for Payment and Checkouts using Polar"
}
```

[Polar](https://polar.sh) is a developer first payment infrastructure. Out of the box it provides a lot of developer first integrations for payments, checkouts and more. This plugin helps you integrate Polar with Better Auth to make your auth + payments flow seamless.

<Callout>

This plugin is maintained by Polar team. For bugs, issues or feature requests, please visit the [Polar GitHub repo](https://github.com/polarsource/polar-adapters).

</Callout>

### ## Features

- \* Checkout Integration
- \* Customer Portal
- \* Automatic Customer creation on signup

- \* Event Ingestion & Customer Meters for flexible Usage Based Billing
- \* Handle Polar Webhooks securely with signature verification
- \* Reference System to associate purchases with organizations

## ## Installation

```
```bash
pnpm add better-auth @polar-sh/better-auth @polar-sh/sdk
```
```

## ## Preparation

Go to your Polar Organization Settings, and create an Organization Access Token. Add it to your environment.

```
```bash
# .env
POLAR_ACCESS_TOKEN=...
```
```

## ### Configuring BetterAuth Server

The Polar plugin comes with a handful additional plugins which adds functionality to your stack.

- \* Checkout - Enables a seamless checkout integration
- \* Portal - Makes it possible for your customers to manage their orders, subscriptions & granted benefits
- \* Usage - Simple extension for listing customer meters & ingesting events for Usage Based Billing
- \* Webhooks - Listen for relevant Polar webhooks

```
```typescript
import { betterAuth } from "better-auth";
import { polar, checkout, portal, usage, webhooks } from "@polar-sh/better-auth";
import { Polar } from "@polar-sh/sdk";

const polarClient = new Polar({
  accessToken: process.env.POLAR_ACCESS_TOKEN,
  // Use 'sandbox' if you're using the Polar Sandbox environment
  // Remember that access tokens, products, etc. are completely separated between environments.
  // Access tokens obtained in Production are for instance not usable in the Sandbox environment.
  server: 'sandbox'
});

const auth = betterAuth({
  // ... Better Auth config
  plugins: [
    polar({
      client: polarClient,
      createCustomerOnSignUp: true,
      use: [
        checkout({
          products: [
            {
              productId: "123-456-789", // ID of Product from Polar Dashboard
              slug: "pro" // Custom slug for easy reference in Checkout URL, e.g. /checkout/pro
            }
          ],
          successUrl: "/success?checkout_id={CHECKOUT_ID}",
          authenticatedUsersOnly: true
        }),
        portal(),
        usage(),
        webhooks({
          secret: process.env.POLAR_WEBHOOK_SECRET,
          onCustomerStateChanged: (payload) => // Triggered when anything regarding a customer changes
          onOrderPaid: (payload) => // Triggered when an order was paid (purchase, subscription renewal, etc.)
          ... // Over 25 granular webhook handlers
        })
      ]
    })
  ]
});
```

```

    onPayload: (payload) => // Catch-all for all events
  })
},
})
]
});
```

```

### ### Configuring BetterAuth Client

You will be using the BetterAuth Client to interact with the Polar functionalities.

```

```typescript
import { createAuthClient } from "better-auth/react";
import { polarClient } from "@polar-sh/better-auth";
import { organizationClient } from "better-auth/client/plugins";

// This is all that is needed
// All Polar plugins, etc. should be attached to the server-side BetterAuth config
export const authClient = createAuthClient({
  plugins: [polarClient()],
});
```

```

### ## Configuration Options

```

```typescript
import { betterAuth } from "better-auth";
import {
  polar,
  checkout,
  portal,
  usage,
  webhooks,
} from "@polar-sh/better-auth";
import { Polar } from "@polar-sh/sdk";

const polarClient = new Polar({
  accessToken: process.env.POLAR_ACCESS_TOKEN,
  // Use 'sandbox' if you're using the Polar Sandbox environment
  // Remember that access tokens, products, etc. are completely separated between environments.
  // Access tokens obtained in Production are for instance not usable in the Sandbox environment.
  server: "sandbox",
});

const auth = betterAuth({
  // ... Better Auth config
  plugins: [
    polar({
      client: polarClient,
      createCustomerOnSignUp: true,
      getCustomerCreateParams: ({ user }, request) => ({
        metadata: {
          myCustomProperty: 123,
        },
      }),
    }),
    use: [
      // This is where you add Polar plugins
    ],
  ]),
],
});
```

```

### ### Required Options

\* `client` : Polar SDK client instance

### ### Optional Options

- \* `createCustomerOnSignUp` : Automatically create a Polar customer when a user signs up
- \* `getCustomerCreateParams` : Custom function to provide additional customer creation metadata

### ### Customers

When `createCustomerOnSignUp` is enabled, a new Polar Customer is automatically created when a new User is added in the Better-Auth Database.

All new customers are created with an associated `externalId`, which is the ID of your User in the Database. This allows us to skip any Polar to User mapping in your Database.

### ## Checkout Plugin

To support checkouts in your app, simply pass the Checkout plugin to the use-property.

```
```typescript
import { polar, checkout } from "@polar-sh/better-auth";

const auth = betterAuth({
  // ... Better Auth config
  plugins: [
    polar({
      // ...
      use: [
        checkout({
          // Optional field - will make it possible to pass a slug to checkout instead of Product ID
          products: [ { productId: "123-456-789", slug: "pro" } ],
          // Relative URL to return to when checkout is successfully completed
          successUrl: "/success?checkout_id={CHECKOUT_ID}",
          // Whether you want to allow unauthenticated checkout sessions or not
          authenticatedUsersOnly: true
        })
      ],
    })
  ],
});
```
```

When checkouts are enabled, you're able to initialize Checkout Sessions using the checkout-method on the BetterAuth Client. This will redirect the user to the Product Checkout.

```
```typescript
await authClient.checkout({
  // Any Polar Product ID can be passed here
  products: ["e651f46d-ac20-4f26-b769-ad088b123df2"],
  // Or, if you setup "products" in the Checkout Config, you can pass the slug
  slug: "pro",
});
```
```

Checkouts will automatically carry the authenticated User as the customer to the checkout. Email-address will be "locked-in".

If `authenticatedUsersOnly` is `false` - then it will be possible to trigger checkout sessions without any associated customer.

### ### Organization Support

This plugin supports the Organization plugin. If you pass the organization ID to the Checkout referenceId, you will be able to keep track of purchases made from organization members.

```

```typescript
const organizationId = (await authClient.organization.list())?.data?.[0]?.id,

await authClient.checkout({
  // Any Polar Product ID can be passed here
  products: ["e651f46d-ac20-4f26-b769-ad088b123df2"],
  // Or, if you setup "products" in the Checkout Config, you can pass the slug
  slug: 'pro',
  // Reference ID will be saved as `referenceId` in the metadata of the checkout, order & subscription object
  referenceId: organizationId
});
```

```

### ### Portal Plugin

A plugin which enables customer management of their purchases, orders and subscriptions.

```

```typescript
import { polar, checkout, portal } from "@polar-sh/better-auth";

const auth = betterAuth({
  // ... Better Auth config
  plugins: [
    polar({
      ...
      use: [
        checkout(...),
        portal()
      ],
    })
  ]
});
```

```

The portal-plugin gives the BetterAuth Client a set of customer management methods, scoped under `authClient.customer`.

### #### Customer Portal Management

The following method will redirect the user to the Polar Customer Portal, where they can see orders, purchases, subscriptions, benefits, etc.

```

```typescript
await authClient.customer.portal();
```

```

### #### Customer State

The portal plugin also adds a convenient state-method for retrieving the general Customer State.

```

```typescript
const { data: customerState } = await authClient.customer.state();
```

```

The customer state object contains:

- \* All the data about the customer.
- \* The list of their active subscriptions
  - \* Note: This does not include subscriptions done by a parent organization. See the subscription list-method below for more information.
- \* The list of their granted benefits.
- \* The list of their active meters, with their current balance.

Thus, with that single object, you have all the required information to check if you should provision access to your service or not.



[You can learn more about the Polar Customer State in the Polar Docs](https://docs.polar.sh/integrate/customer-state).

### #### Benefits, Orders & Subscriptions

The portal plugin adds 3 convenient methods for listing benefits, orders & subscriptions relevant to the authenticated user/customer.

[All of these methods use the Polar CustomerPortal APIs](https://docs.polar.sh/api-reference/customer-portal)

#### ##### Benefits

This method only lists granted benefits for the authenticated user/customer.

```
```typescript
const { data: benefits } = await authClient.customer.benefits.list({
  query: {
    page: 1,
    limit: 10,
  },
});
```
```

#### ##### Orders

This method lists orders like purchases and subscription renewals for the authenticated user/customer.

```
```typescript
const { data: orders } = await authClient.customer.orders.list({
  query: {
    page: 1,
    limit: 10,
    productBillingType: "one_time", // or 'recurring'
  },
});
```
```

#### ##### Subscriptions

This method lists the subscriptions associated with authenticated user/customer.

```
```typescript
const { data: subscriptions } = await authClient.customer.subscriptions.list({
  query: {
    page: 1,
    limit: 10,
    active: true,
  },
});
```
```

### \*\*Important\*\* - Organization Support

This will **not** return subscriptions made by a parent organization to the authenticated user.

However, you can pass a `referenceId` to this method. This will return all subscriptions associated with that referenceId instead of subscriptions associated with the user.

So in order to figure out if a user should have access, pass the user's organization ID to see if there is an active subscription for that organization.

```
```typescript
const organizationId = (await authClient.organization.list())?.data?.[0]?.id,

const { data: subscriptions } = await authClient.customer.orders.list({
```

```

    query: {
      page: 1,
      limit: 10,
      active: true,
      referenceId: organizationId
    },
  });

```

```

const userShouldHaveAccess = subscriptions.some(
  sub => // Your logic to check subscription product or whatever.
)
```

```

### ## Usage Plugin

A simple plugin for Usage Based Billing.

```

```typescript
import { polar, checkout, portal, usage } from "@polar-sh/better-auth";

const auth = betterAuth({
  // ... Better Auth config
  plugins: [
    polar({
      ...
      use: [
        checkout(...),
        portal(),
        usage()
      ],
    })
  ]
});
```

```

### ### Event Ingestion

Polar's Usage Based Billing builds entirely on event ingestion. Ingest events from your application, create Meters to represent that usage, and add metered prices to Products to charge for it.

[Learn more about Usage Based Billing in the Polar Docs.](<https://docs.polar.sh/features/usage-based-billing/introduction>)

```

```typescript
const { data: ingested } = await authClient.usage.ingest({
  event: "file-uploads",
  metadata: {
    uploadedFiles: 12,
  },
});
```

```

The authenticated user is automatically associated with the ingested event.

### ### Customer Meters

A simple method for listing the authenticated user's Usage Meters, or as we call them, Customer Meters.

Customer Meter's contains all information about their consumption on your defined meters.

- \* Customer Information
- \* Meter Information
- \* Customer Meter Information
  - \* Consumed Units
  - \* Credited Units
  - \* Balance

```
```typescript
const { data: customerMeters } = await authClient.usage.meters.list({
  query: {
    page: 1,
    limit: 10,
  },
});
```
```

### ## Webhooks Plugin

The Webhooks plugin can be used to capture incoming events from your Polar organization.

```
```typescript
import { polar, webhooks } from "@polar-sh/better-auth";

const auth = betterAuth({
  // ... Better Auth config
  plugins: [
    polar({
      ...
      use: [
        webhooks({
          secret: process.env.POLAR_WEBHOOK_SECRET,
          onCustomerStateChanged: (payload) => // Triggered when anything regarding a customer changes
          onOrderPaid: (payload) => // Triggered when an order was paid (purchase, subscription renewal, etc.)
          ... // Over 25 granular webhook handlers
          onPayload: (payload) => // Catch-all for all events
        })
      ],
    })
  ],
});
```
```

Configure a Webhook endpoint in your Polar Organization Settings page. Webhook endpoint is configured at /polar/webhooks.

Add the secret to your environment.

```
```bash
# .env
POLAR_WEBHOOK_SECRET=...
```
```

The plugin supports handlers for all Polar webhook events:

- \* `onPayload` - Catch-all handler for any incoming Webhook event
- \* `onCheckoutCreated` - Triggered when a checkout is created
- \* `onCheckoutUpdated` - Triggered when a checkout is updated
- \* `onOrderCreated` - Triggered when an order is created
- \* `onOrderPaid` - Triggered when an order is paid
- \* `onOrderRefunded` - Triggered when an order is refunded
- \* `onRefundCreated` - Triggered when a refund is created
- \* `onRefundUpdated` - Triggered when a refund is updated
- \* `onSubscriptionCreated` - Triggered when a subscription is created
- \* `onSubscriptionUpdated` - Triggered when a subscription is updated
- \* `onSubscriptionActive` - Triggered when a subscription becomes active
- \* `onSubscriptionCanceled` - Triggered when a subscription is canceled
- \* `onSubscriptionRevoked` - Triggered when a subscription is revoked
- \* `onSubscriptionUncanceled` - Triggered when a subscription cancellation is reversed
- \* `onProductCreated` - Triggered when a product is created
- \* `onProductUpdated` - Triggered when a product is updated
- \* `onOrganizationUpdated` - Triggered when an organization is updated
- \* `onBenefitCreated` - Triggered when a benefit is created

- \* `onBenefitUpdated` - Triggered when a benefit is updated
- \* `onBenefitGrantCreated` - Triggered when a benefit grant is created
- \* `onBenefitGrantUpdated` - Triggered when a benefit grant is updated
- \* `onBenefitGrantRevoked` - Triggered when a benefit grant is revoked
- \* `onCustomerCreated` - Triggered when a customer is created
- \* `onCustomerUpdated` - Triggered when a customer is updated
- \* `onCustomerDeleted` - Triggered when a customer is deleted
- \* `onCustomerStateChanged` - Triggered when a customer is created

file: ./content/docs/plugins/sso.mdx

```
meta: {
  "title": "Single Sign-On (SSO)",
  "description": "Integrate Single Sign-On (SSO) with your application."
}
```

`OIDC` `OAuth2` `SSO`

Single Sign-On (SSO) allows users to authenticate with multiple applications using a single set of credentials. This plugin supports OpenID Connect (OIDC) and OAuth2 providers.

<Callout>

SAML support is coming soon. Upvote the feature request on our [GitHub](https://github.com/better-auth/better-auth/issues/96)

</Callout>

## ## Installation

<Steps>

<Step>

#### Add Plugin to the server

```
```ts title="auth.ts"
import { betterAuth } from "better-auth"
import { sso } from "better-auth/plugins/sso";

const auth = betterAuth({
  plugins: [ // [!code highlight]
    sso() // [!code highlight]
  ] // [!code highlight]
})
```
```

</Step>

<Step>

#### Migrate the database

Run the migration or generate the schema to add the necessary fields and tables to the database.

<Tabs items=["migrate", "generate"]>

<Tab value="migrate">

```
```bash
npx @better-auth/cli migrate
```
```

</Tab>

<Tab value="generate">

```
```bash
npx @better-auth/cli generate
```
```

</Tab>

</Tabs>

See the [Schema](#schema) section to add the fields manually.

</Step>

```

<Step>
  ### Add the client plugin

  ```ts title="auth-client.ts"
import { createAuthClient } from "better-auth/client"
import { ssoClient } from "better-auth/client/plugins"

const authClient = createAuthClient({
  plugins: [ // [!code highlight]
    ssoClient() // [!code highlight]
  ] // [!code highlight]
})
  ```
</Step>
</Steps>

```

## ## Usage

### ### Register an OIDC Provider

To register an OIDC provider, use the `createOIDCProvider` endpoint and provide the necessary configuration details for the provider.

A redirect URL will be automatically generated using the provider ID. For instance, if the provider ID is `hydra`, the redirect URL would be `{baseURL}/api/auth/sso/callback/hydra`. Note that `/api/auth` may vary depending on your base path configuration.

```

<Tabs items=[["client", "server"]]>
  <Tab value="client">
    ```ts title="register-provider.ts"
import { authClient } from "@lib/auth-client";

// only with issuer if the provider supports discovery
await authClient.sso.register({
  issuer: "https://idp.example.com",
  providerId: "example-provider",
});

// with all fields
await authClient.sso.register({
  issuer: "https://idp.example.com",
  domain: "example.com",
  clientId: "client-id",
  clientSecret: "client-secret",
  authorizationEndpoint: "https://idp.example.com/authorize",
  tokenEndpoint: "https://idp.example.com/token",
  jwksEndpoint: "https://idp.example.com/jwks",
  mapping: {
    id: "sub",
    email: "email",
    emailVerified: "email_verified",
    name: "name",
    image: "picture",
  },
  providerId: "example-provider",
});
    ```
  </Tab>

  <Tab value="server">
    ```ts title="register-provider.ts"
const { headers } = await signInWithTestUser();
await auth.api.createOIDCProvider({
  body: {

```

```

    issuer: "https://idp.example.com",
    domain: "example.com",
    clientId: "your-client-id",
    clientSecret: "your-client-secret",
    authorizationEndpoint: "https://idp.example.com/authorize",
    tokenEndpoint: "https://idp.example.com/token",
    jwksEndpoint: "https://idp.example.com/jwks",
    mapping: {
      id: "sub",
      email: "email",
      emailVerified: "email_verified",
      name: "name",
      image: "picture",
    },
    providerId: "example-provider",
  },
  headers,
});
`,
</Tab>
</Tabs>

```

### ### Sign In with SSO

To sign in with an SSO provider, you can call `signIn.sso`

You can sign in using the email with domain matching:

```

```ts title="sign-in.ts"
const res = await authClient.signIn.sso({
  email: "user@example.com",
  callbackURL: "/dashboard",
});
```

```

or you can specify the domain:

```

```ts title="sign-in-domain.ts"
const res = await authClient.signIn.sso({
  domain: "example.com",
  callbackURL: "/dashboard",
});
```

```

You can also sign in using the organization slug if a provider is associated with an organization:

```

```ts title="sign-in-org.ts"
const res = await authClient.signIn.sso({
  organizationSlug: "example-org",
  callbackURL: "/dashboard",
});
```

```

Alternatively, you can sign in using the provider's ID:

```

```ts title="sign-in-provider-id.ts"
const res = await authClient.signIn.sso({
  providerId: "example-provider-id",
  callbackURL: "/dashboard",
});
```

```

To use the server API you can use `signInSSO`

```

```ts title="sign-in-org.ts"

```

```
const res = await auth.api.signInSSO({
  body: {
    organizationSlug: "example-org",
    callbackURL: "/dashboard",
  }
});
```
```

When a user is authenticated, if the user does not exist, the user will be provisioned using the `provisionUser` function. If the organization provisioning is enabled and a provider is associated with an organization, the user will be added to the organization.

```
```ts title="auth.ts"
const auth = betterAuth({
  plugins: [
    sso({
      provisionUser: async (user) => {
        // provision user
      },
      organizationProvisioning: {
        disabled: false,
        defaultRole: "member",
        getRole: async (user) => {
          // get role if needed
        },
      },
    }),
  ],
});
```
```

### ## Schema

The plugin requires additional fields in the `ssoProvider` table to store the provider's configuration.

```
<DatabaseTable
  fields=[
    {
      name: "id", type: "string", description: "A database identifier", isRequired: true, isPrimaryKey: true,
    },
    { name: "issuer", type: "string", description: "The issuer identifier", isRequired: true },
    { name: "domain", type: "string", description: "The domain of the provider", isRequired: true },
    { name: "oidcConfig", type: "string", description: "The OIDC configuration", isRequired: false },
    { name: "userId", type: "string", description: "The user ID", isRequired: true, references: { model: "user", field: "id" } },
    { name: "providerId", type: "string", description: "The provider ID. Used to identify a provider and to generate a redirect URL.", isRequired: true, isUnique: true },
    { name: "organizationId", type: "string", description: "The organization Id. If provider is linked to an organization.", isRequired: false },
  ]
/>
```

### ## Options

#### ### Server

**\*\*provisionUser\*\***: A custom function to provision a user when they sign in with an SSO provider.

**\*\*organizationProvisioning\*\***: Options for provisioning users to an organization.

```
<TypeTable
  type={{
    provisionUser: {
      description: "A custom function to provision a user when they sign in with an SSO provider.",
      type: "function",
    },
  },
```

```

organizationProvisioning: {
  description: "Options for provisioning users to an organization.",
  type: "object",
  properties: {
    disabled: {
      description: "Disable organization provisioning.",
      type: "boolean",
      default: false,
    },
    defaultRole: {
      description: "The default role for new users.",
      type: "string",
      enum: ["member", "admin"],
      default: "member",
    },
    getRole: {
      description: "A custom function to determine the role for new users.",
      type: "function",
    },
  },
},
},
}}
/>

```

```
file: ./content/docs/plugins/stripe.mdx
```

```

meta: {
  "title": "Stripe",
  "description": "Stripe plugin for Better Auth to manage subscriptions and payments."
}

```

The Stripe plugin integrates Stripe's payment and subscription functionality with Better Auth. Since payment and authentication are often tightly coupled, this plugin simplifies the integration of stripe into your application, handling customer creation, subscription management, and webhook processing.

<Callout type="warn">

This plugin is currently in beta. We're actively collecting feedback and exploring additional features. If you have feature requests or suggestions, please join our [Discord community](https://discord.gg/better-auth) to discuss them.

</Callout>

## ## Features

- \* Create Stripe Customers automatically when users sign up
- \* Manage subscription plans and pricing
- \* Process subscription lifecycle events (creation, updates, cancellations)
- \* Handle Stripe webhooks securely with signature verification
- \* Expose subscription data to your application
- \* Support for trial periods and subscription upgrades
- \* Flexible reference system to associate subscriptions with users or organizations
- \* Team subscription support with seats management

## ## Installation

<Steps>

<Step>

#### Install the plugin

First, install the plugin:

<Tabs groupId="package-manager" persist items={}>

<Tab value="npm">

```
```bash
```

```
npm install @better-auth/stripe
```

```
```
```

</Tab>



```

<Tab value="pnpm">
  ```bash
  pnpm add @better-auth/stripe
  ```
</Tab>

<Tab value="yarn">
  ```bash
  yarn add @better-auth/stripe
  ```
</Tab>

<Tab value="bun">
  ```bash
  bun add @better-auth/stripe
  ```
</Tab>
</Tabs>

<Callout>
  If you're using a separate client and server setup, make sure to install the plugin in both parts of your project.
</Callout>
</Step>

```

```

<Step>
  #### Install the Stripe SDK

```

Next, install the Stripe SDK on your server:

```

<Tabs groupId="package-manager" persist items={}>
  <Tab value="npm">
    ```bash
    npm install stripe@^18.0.0
    ```
  </Tab>

  <Tab value="pnpm">
    ```bash
    pnpm add stripe@^18.0.0
    ```
  </Tab>

  <Tab value="yarn">
    ```bash
    yarn add stripe@^18.0.0
    ```
  </Tab>

  <Tab value="bun">
    ```bash
    bun add stripe@^18.0.0
    ```
  </Tab>
</Tabs>
</Step>

```

```

<Step>
  #### Add the plugin to your auth config

  ```ts title="auth.ts"
  import { betterAuth } from "better-auth"
  import { stripe } from "@better-auth/stripe"
  import Stripe from "stripe"

```

```
const stripeClient = new Stripe(process.env.STRIPE_SECRET_KEY!, {
  apiVersion: "2025-02-24.acacia",
})

export const auth = betterAuth({
  // ... your existing config
  plugins: [
    stripe({
      stripeClient,
      stripeWebhookSecret: process.env.STRIPE_WEBHOOK_SECRET!,
      createCustomerOnSignUp: true,
    })
  ]
})
````
```

&lt;/Step&gt;

&lt;Step&gt;

#### Add the client plugin

```
````ts title="auth-client.ts"
import { createAuthClient } from "better-auth/client"
import { stripeClient } from "@better-auth/stripe/client"

export const client = createAuthClient({
  // ... your existing config
  plugins: [
    stripeClient({
      subscription: true //if you want to enable subscription management
    })
  ]
})
````
```

&lt;/Step&gt;

&lt;Step&gt;

#### Migrate the database

Run the migration or generate the schema to add the necessary tables to the database.

```
<Tabs items=["migrate", "generate"]>
<Tab value="migrate">
  ````bash
  npx @better-auth/cli migrate
  ````
```

&lt;/Tab&gt;

```
<Tab value="generate">
  ````bash
  npx @better-auth/cli generate
  ````
```

&lt;/Tab&gt;

&lt;/Tabs&gt;

See the [Schema](#schema) section to add the tables manually.

&lt;/Step&gt;

&lt;Step&gt;

#### Set up Stripe webhooks

Create a webhook endpoint in your Stripe dashboard pointing to:

```
````
https://your-domain.com/api/auth/stripe/webhook
````
```

`/api/auth` is the default path for the auth server.

Make sure to select at least these events:

- \* `checkout.session.completed`
- \* `customer.subscription.updated`
- \* `customer.subscription.deleted`

Save the webhook signing secret provided by Stripe and add it to your environment variables as `STRIPE\_WEBHOOK\_SECRET`.

</Step>

</Steps>

## ## Usage

### ### Customer Management

You can use this plugin solely for customer management without enabling subscriptions. This is useful if you just want to link Stripe customers to your users.

By default, when a user signs up, a Stripe customer is automatically created if you set `createCustomerOnSignUp: true`. This customer is linked to the user in your database. You can customize the customer creation process:

```
```ts title="auth.ts"
stripe({
  // ... other options
  createCustomerOnSignUp: true,
  onCustomerCreate: async ({ customer, stripeCustomer, user }, request) => {
    // Do something with the newly created customer
    console.log(` Customer ${customer.id} created for user ${user.id}`);
  },
  getCustomerCreateParams: async ({ user, session }, request) => {
    // Customize the Stripe customer creation parameters
    return {
      metadata: {
        referralSource: user.metadata?.referralSource
      }
    };
  }
})
```
```

### ### Subscription Management

#### #### Defining Plans

You can define your subscription plans either statically or dynamically:

```
```ts title="auth.ts"
// Static plans
subscription: {
  enabled: true,
  plans: [
    {
      name: "basic", // the name of the plan, it'll be automatically lower cased when stored in the database
      priceId: "price_1234567890", // the price ID from stripe
      annualDiscountPriceId: "price_1234567890", // (optional) the price ID for annual billing with a discount
      limits: {
        projects: 5,
        storage: 10
      }
    },
    {

```

```

    name: "pro",
    priceId: "price_0987654321",
    limits: {
      projects: 20,
      storage: 50
    },
    freeTrial: {
      days: 14,
    }
  }
]
}

// Dynamic plans (fetched from database or API)
subscription: {
  enabled: true,
  plans: async () => {
    const plans = await db.query("SELECT * FROM plans");
    return plans.map(plan => ({
      name: plan.name,
      priceId: plan.stripe_price_id,
      limits: JSON.parse(plan.limits)
    }));
  }
}
``,`

```

see [plan configuration](#plan-configuration) for more.

#### #### Creating a Subscription

To create a subscription, use the `subscription.upgrade` method:

```

` `` ts title="client.ts"
await client.subscription.upgrade({
  plan: "pro",
  successUrl: "/dashboard",
  cancelUrl: "/pricing",
  annual: true, // Optional: upgrade to an annual plan
  referenceId: "org_123" // Optional: defaults to the current logged in user ID
  seats: 5 // Optional: for team plans
});
``,`

```

This will create a Checkout Session and redirect the user to the Stripe Checkout page.

<Callout type="warn">

If the user already has an active subscription, you *must* provide the `subscriptionId` parameter. Otherwise, the user will be subscribed to (and pay for) both plans.

</Callout>

> **Important:** The `successUrl` parameter will be internally modified to handle race conditions between checkout completion and webhook processing. The plugin creates an intermediate redirect that ensures subscription status is properly updated before redirecting to your success page.

```

` `` ts
const { error } = await client.subscription.upgrade({
  plan: "pro",
  successUrl: "/dashboard",
  cancelUrl: "/pricing",
});
if(error) {
  alert(error.message);
}
``,`

```

<Callout type="warn">

For each reference ID (user or organization), only one active or trialing subscription is supported at a time. The plugin doesn't currently support multiple concurrent active subscriptions for the same reference ID.

</Callout>

#### ##### Switching Plans

To switch a subscription to a different plan, use the `subscription.upgrade` method:

```
```ts title="client.ts"
await client.subscription.upgrade({
  plan: "pro",
  successUrl: "/dashboard",
  cancelUrl: "/pricing",
  subscriptionId: "sub_123", // the Stripe subscription ID of the user's current plan
});
```
```

This ensures that the user only pays for the new plan, and not both.

#### ##### Listing Active Subscriptions

To get the user's active subscriptions:

```
```ts title="client.ts"
const { data: subscriptions } = await client.subscription.list();

// get the active subscription
const activeSubscription = subscriptions.find(
  sub => sub.status === "active" || sub.status === "trialing"
);

// Check subscription limits
const projectLimit = subscriptions?.limits?.projects || 0;
```
```

#### ##### Canceling a Subscription

To cancel a subscription:

```
```ts title="client.ts"
const { data } = await client.subscription.cancel({
  returnUrl: "/account",
  referenceId: "org_123" // optional defaults to userId
});
```
```

This will redirect the user to the Stripe Billing Portal where they can cancel their subscription.

#### ##### Restoring a Canceled Subscription

If a user changes their mind after canceling a subscription (but before the subscription period ends), you can restore the subscription:

```
```ts title="client.ts"
const { data } = await client.subscription.restore({
  referenceId: "org_123" // optional, defaults to userId
});
```
```

This will reactivate a subscription that was previously set to cancel at the end of the billing period (`cancelAtPeriodEnd: true`). The subscription will continue to renew automatically.

> **Note:** This only works for subscriptions that are still active but marked to cancel at the end of the period. It cannot

restore subscriptions that have already ended.

### ### Reference System

By default, subscriptions are associated with the user ID. However, you can use a custom reference ID to associate subscriptions with other entities, such as organizations:

```
```ts title="client.ts"
// Create a subscription for an organization
await client.subscription.upgrade({
  plan: "pro",
  referenceId: "org_123456",
  successUrl: "/dashboard",
  cancelUrl: "/pricing",
  seats: 5 // Number of seats for team plans
});

// List subscriptions for an organization
const { data: subscriptions } = await client.subscription.list({
  query: {
    referenceId: "org_123456"
  }
});
```
```

### #### Team Subscriptions with Seats

For team or organization plans, you can specify the number of seats:

```
```ts
await client.subscription.upgrade({
  plan: "team",
  referenceId: "org_123456",
  seats: 10, // 10 team members
  successUrl: "/org/billing/success",
  cancelUrl: "/org/billing"
});
```
```

The `seats` parameter is passed to Stripe as the quantity for the subscription item. You can use this value in your application logic to limit the number of members in a team or organization.

To authorize reference IDs, implement the `authorizeReference` function:

```
```ts title="auth.ts"
subscription: {
  // ... other options
  authorizeReference: async ({ user, session, referenceId, action }) => {
    // Check if the user has permission to manage subscriptions for this reference
    if (action === "upgrade-subscription" || action === "cancel-subscription" || action === "restore-subscription") {
      const org = await db.member.findFirst({
        where: {
          organizationId: referenceId,
          userId: user.id
        }
      });
    }
    return org?.role === "owner"
  }
  return true;
}
}
```
```

### ### Webhook Handling

The plugin automatically handles common webhook events:

- \* `checkout.session.completed` : Updates subscription status after checkout
- \* `customer.subscription.updated` : Updates subscription details when changed
- \* `customer.subscription.deleted` : Marks subscription as canceled

You can also handle custom events:

```
```ts title="auth.ts"
stripe({
  // ... other options
  onEvent: async (event) => {
    // Handle any Stripe event
    switch (event.type) {
      case "invoice.paid":
        // Handle paid invoice
        break;
      case "payment_intent.succeeded":
        // Handle successful payment
        break;
    }
  }
})
```
```

### Subscription Lifecycle Hooks

You can hook into various subscription lifecycle events:

```
```ts title="auth.ts"
subscription: {
  // ... other options
  onSubscriptionComplete: async ({ event, subscription, stripeSubscription, plan }) => {
    // Called when a subscription is successfully created
    await sendWelcomeEmail(subscription.referenceId, plan.name);
  },
  onSubscriptionUpdate: async ({ event, subscription }) => {
    // Called when a subscription is updated
    console.log(` Subscription ${subscription.id} updated` );
  },
  onSubscriptionCancel: async ({ event, subscription, stripeSubscription, cancellationDetails }) => {
    // Called when a subscription is canceled
    await sendCancellationEmail(subscription.referenceId);
  },
  onSubscriptionDeleted: async ({ event, subscription, stripeSubscription }) => {
    // Called when a subscription is deleted
    console.log(` Subscription ${subscription.id} deleted` );
  }
}
```
```

### Trial Periods

You can configure trial periods for your plans:

```
```ts title="auth.ts"
{
  name: "pro",
  priceId: "price_0987654321",
  freeTrial: {
    days: 14,
    onTrialStart: async (subscription) => {
      // Called when a trial starts
      await sendTrialStartEmail(subscription.referenceId);
    },
  },
}
```

```

    onTrialEnd: async ({ subscription, user }, request) => {
      // Called when a trial ends
      await sendTrialEndEmail(user.email);
    },
    onTrialExpired: async (subscription) => {
      // Called when a trial expires without conversion
      await sendTrialExpiredEmail(subscription.referenceId);
    }
  }
}
}
`

```

### ### Schema

The Stripe plugin adds the following tables to your database:

#### #### User

Table Name: `user`

```

<DatabaseTable
  fields=[
    {
      name: "stripeCustomerId",
      type: "string",
      description: "The Stripe customer ID",
      isOptional: true
    },
  ]
/>

```

#### #### Subscription

Table Name: `subscription`

```

<DatabaseTable
  fields=[
    {
      name: "id",
      type: "string",
      description: "Unique identifier for each subscription",
      isPrimaryKey: true
    },
    {
      name: "plan",
      type: "string",
      description: "The name of the subscription plan"
    },
    {
      name: "referenceId",
      type: "string",
      description: "The ID this subscription is associated with (user ID by default)",
      isUnique: true
    },
    {
      name: "stripeCustomerId",
      type: "string",
      description: "The Stripe customer ID",
      isOptional: true
    },
    {
      name: "stripeSubscriptionId",
      type: "string",
      description: "The Stripe subscription ID",
      isOptional: true
    }
  ]
/>

```



```

    },
    {
      name: "status",
      type: "string",
      description: "The status of the subscription (active, canceled, etc.)",
      defaultValue: "incomplete"
    },
    {
      name: "periodStart",
      type: "Date",
      description: "Start date of the current billing period",
      isOptional: true
    },
    {
      name: "periodEnd",
      type: "Date",
      description: "End date of the current billing period",
      isOptional: true
    },
    {
      name: "cancelAtPeriodEnd",
      type: "boolean",
      description: "Whether the subscription will be canceled at the end of the period",
      defaultValue: false,
      isOptional: true
    },
    {
      name: "seats",
      type: "number",
      description: "Number of seats for team plans",
      isOptional: true
    },
    {
      name: "trialStart",
      type: "Date",
      description: "Start date of the trial period",
      isOptional: true
    },
    {
      name: "trialEnd",
      type: "Date",
      description: "End date of the trial period",
      isOptional: true
    }
  ]
}
/>

```

### ### Customizing the Schema

To change the schema table names or fields, you can pass a `schema` option to the Stripe plugin:

```

```ts title="auth.ts"
stripe({
  // ... other options
  schema: {
    subscription: {
      modelName: "stripeSubscriptions", // map the subscription table to stripeSubscriptions
      fields: {
        plan: "planName" // map the plan field to planName
      }
    }
  }
})
```

```

## ## Options

### #### Main Options

**\*\*stripeClient\*\***: `Stripe` - The Stripe client instance. Required.

**\*\*stripeWebhookSecret\*\***: `string` - The webhook signing secret from Stripe. Required.

**\*\*createCustomerOnSignUp\*\***: `boolean` - Whether to automatically create a Stripe customer when a user signs up. Default: `false`.

**\*\*onCustomerCreate\*\***: `(data: { customer: Customer, stripeCustomer: Stripe.Customer, user: User }, request?: Request) => Promise<void>` - A function called after a customer is created.

**\*\*getCustomerCreateParams\*\***: `(data: { user: User, session: Session }, request?: Request) => Promise<{}>` - A function to customize the Stripe customer creation parameters.

**\*\*onEvent\*\***: `(event: Stripe.Event) => Promise<void>` - A function called for any Stripe webhook event.

### #### Subscription Options

**\*\*enabled\*\***: `boolean` - Whether to enable subscription functionality. Required.

**\*\*plans\*\***: `Plan[] | (() => Promise<Plan[]>)` - An array of subscription plans or a function that returns plans. Required if subscriptions are enabled.

**\*\*requireEmailVerification\*\***: `boolean` - Whether to require email verification before allowing subscription upgrades. Default: `false`.

**\*\*authorizeReference\*\***: `(data: { user: User, session: Session, referenceId: string, action: "upgrade-subscription" | "list-subscription" | "cancel-subscription" | "restore-subscription"}, request?: Request) => Promise<boolean>` - A function to authorize reference IDs.

### #### Plan Configuration

Each plan can have the following properties:

**\*\*name\*\***: `string` - The name of the plan. Required.

**\*\*priceId\*\***: `string` - The Stripe price ID. Required unless using `lookupKey`.

**\*\*lookupKey\*\***: `string` - The Stripe price lookup key. Alternative to `priceId`.

**\*\*annualDiscountPriceId\*\***: `string` - A price ID for annual billing.

**\*\*annualDiscountLookupKey\*\***: `string` - The Stripe price lookup key for annual billing. Alternative to `annualDiscountPriceId`.

**\*\*limits\*\***: `Record<string, number>` - Limits associated with the plan (e.g., `{ projects: 10, storage: 5 }`).

**\*\*group\*\***: `string` - A group name for the plan, useful for categorizing plans.

**\*\*freeTrial\*\***: Object containing trial configuration:

**\*\*days\*\***: `number` - Number of trial days.

**\*\*onTrialStart\*\***: `(subscription: Subscription) => Promise<void>` - Called when a trial starts.

**\*\*onTrialEnd\*\***: `(data: { subscription: Subscription, user: User }, request?: Request) => Promise<void>` - Called when a trial ends.

**\*\*onTrialExpired\*\***: `(subscription: Subscription) => Promise<void>` - Called when a trial expires without conversion.

## ## Advanced Usage

### #### Using with Organizations

The Stripe plugin works well with the organization plugin. You can associate subscriptions with organizations instead of

individual users:

```
```ts title="client.ts"
// Get the active organization
const { data: activeOrg } = client.useActiveOrganization();

// Create a subscription for the organization
await client.subscription.upgrade({
  plan: "team",
  referenceId: activeOrg.id,
  seats: 10,
  annual: true, // upgrade to an annual plan (optional)
  successUrl: "/org/billing/success",
  cancelUrl: "/org/billing"
});
```
```

Make sure to implement the `authorizeReference` function to verify that the user has permission to manage subscriptions for the organization:

```
```ts title="auth.ts"
authorizeReference: async ({ user, referenceId, action }) => {
  const member = await db.members.findFirst({
    where: {
      userId: user.id,
      organizationId: referenceId
    }
  });

  return member?.role === "owner" || member?.role === "admin";
}
```
```

### ### Custom Checkout Session Parameters

You can customize the Stripe Checkout session with additional parameters:

```
```ts title="auth.ts"
getCheckoutSessionParams: async ({ user, session, plan, subscription }, request) => {
  return {
    params: {
      allow_promotion_codes: true,
      tax_id_collection: {
        enabled: true
      },
      billing_address_collection: "required",
      custom_text: {
        submit: {
          message: "We'll start your subscription right away"
        }
      },
      metadata: {
        planType: "business",
        referralCode: user.metadata?.referralCode
      }
    },
    options: {
      idempotencyKey: `sub_${user.id}_${plan.name}_${Date.now()}`
    }
  };
}
```
```

### ### Tax Collection

To enable tax collection:

```
```ts title="auth.ts"
subscription: {
  // ... other options
  getCheckoutSessionParams: async ({ user, session, plan, subscription }, request) => {
    return {
      params: {
        tax_id_collection: {
          enabled: true
        }
      }
    };
  }
}
}
```
```

## ## Troubleshooting

### #### Webhook Issues

If webhooks aren't being processed correctly:

1. Check that your webhook URL is correctly configured in the Stripe dashboard
2. Verify that the webhook signing secret is correct
3. Ensure you've selected all the necessary events in the Stripe dashboard
4. Check your server logs for any errors during webhook processing

### #### Subscription Status Issues

If subscription statuses aren't updating correctly:

1. Make sure the webhook events are being received and processed
2. Check that the `stripeCustomerId` and `stripeSubscriptionId` fields are correctly populated
3. Verify that the reference IDs match between your application and Stripe

### #### Testing Webhooks Locally

For local development, you can use the Stripe CLI to forward webhooks to your local environment:

```
```bash
stripe listen --forward-to localhost:3000/api/auth/stripe/webhook
```
```

This will provide you with a webhook signing secret that you can use in your local environment.

```
file: ./content/docs/plugins/username.mdx
meta: {
  "title": "Username",
  "description": "Username plugin"
}
```

The username plugin wraps the email and password authenticator and adds username support. This allows users to sign in and sign up with their username instead of their email.

## ## Installation

<Steps>

<Step>

#### Add Plugin to the server

```
```ts title="auth.ts"
import { betterAuth } from "better-auth"
import { username } from "better-auth/plugins"
```

```
export const auth = betterAuth({
  plugins: [ // [!code highlight]
    username() // [!code highlight]
  ] // [!code highlight]
})
```,
```

</Step>

<Step>

#### Migrate the database

Run the migration or generate the schema to add the necessary fields and tables to the database.

<Tabs items={["migrate", "generate"]}>

<Tab value="migrate">

```
```bash
npx @better-auth/cli migrate
```
```

</Tab>

<Tab value="generate">

```
```bash
npx @better-auth/cli generate
```
```

</Tab>

</Tabs>

See the [Schema](#schema) section to add the fields manually.

</Step>

<Step>

#### Add the client plugin

```
```ts title="auth-client.ts"
```

```
import { createAuthClient } from "better-auth/client"
```

```
import { usernameClient } from "better-auth/client/plugins"
```

```
export const authClient = createAuthClient({
```

```
  plugins: [ // [!code highlight]
    usernameClient() // [!code highlight]
  ] // [!code highlight]
})
```

```
```,
```

</Step>

</Steps>

## Usage

#### Sign up with username

To sign up a user with username, you can use the existing `signIn.email` function provided by the client. The `signIn` function should take a new `username` property in the object.

```
```ts title="auth-client.ts"
```

```
const data = await authClient.signIn.email({
```

```
  email: "email@domain.com",
  name: "Test User",
  password: "password1234",
  username: "test"
```

```
})
```,
```

#### Sign in with username

To sign in a user with username, you can use the `signIn.username` function provided by the client. The `signIn` function takes an object with the following properties:

- \* `username`: The username of the user.
- \* `password`: The password of the user.

```
```ts title="auth-client.ts"
const data = await authClient.signIn.username({
  username: "test",
  password: "password1234",
})
```
```

### ### Update username

To update the username of a user, you can use the `updateUser` function provided by the client.

```
```ts title="auth-client.ts"
const data = await authClient.updateUser({
  username: "new-username"
})
```
```

### ## Schema

The plugin requires 1 field to be added to the user table:

```
<DatabaseTable
  fields=[
    {
      name: "username",
      type: "string",
      description: "The username of the user",
      isUnique: true
    },
    {
      name: "displayUsername",
      type: "string",
      description: "Non normalized username of the user",
      isUnique: true
    },
  ]
/>
```

### ## Options

#### ### Min Username Length

The minimum length of the username. Default is `3`.

```
```ts title="auth.ts"
import { betterAuth } from "better-auth"
import { username } from "better-auth/plugins"

const auth = betterAuth({
  plugins: [
    username({
      minUsernameLength: 5
    })
  ]
})
```
```

#### ### Max Username Length

The maximum length of the username. Default is `30`.

```
```ts title="auth.ts"
import { betterAuth } from "better-auth"
import { username } from "better-auth/plugins"

const auth = betterAuth({
  plugins: [
    username({
      maxUsernameLength: 100
    })
  ]
})
```
```

### ### Username Validator

A function that validates the username. The function should return false if the username is invalid. By default, the username should only contain alphanumeric characters, underscores, and dots.

```
```ts title="auth.ts"
import { betterAuth } from "better-auth"
import { username } from "better-auth/plugins"

const auth = betterAuth({
  plugins: [
    username({
      usernameValidator: (username) => {
        if (username === "admin") {
          return false
        }
      }
    })
  ]
})
```
```

```
file: ./content/docs/reference/contributing.mdx
meta: {
  "title": "Contributing to BetterAuth",
  "description": "A concise guide to contributing to BetterAuth"
}
```

Thank you for your interest in contributing to Better Auth! This guide is a concise guide to contributing to Better Auth.

## ## Getting Started

Before diving in, here are a few important resources:

- \* Take a look at our existing [issues](https://github.com/better-auth/better-auth/issues) and [pull requests](https://github.com/better-auth/better-auth/pulls)
- \* Join our community discussions in [Discord](https://discord.gg/better-auth)

## ## Development Setup

To get started with development:

```
<Callout type="warn">
  Make sure you have Node.JS
  installed, preferably on LTS.
</Callout>
```

```
<Steps>
  <Step>
```

### 1. Fork the repository

Visit [<https://github.com/better-auth/better-auth>](<https://github.com/better-auth/better-auth>)

Click the "Fork" button in the top right.

</Step>

<Step>

### 2. Clone your fork

```
```bash
# Replace YOUR-USERNAME with your GitHub username
git clone https://github.com/YOUR-USERNAME/better-auth.git
cd better-auth
```
```

</Step>

<Step>

### 3. Install dependencies

Make sure you have [pnpm](https://pnpm.io/installation) installed!

```
```bash
pnpm install
```
```

</Step>

<Step>

### 4. Prepare ENV files

Copy the example env file to create your new `.env` file.

```
```bash
cp -n ./docs/.env.example ./docs/.env
```
```

</Step>

</Steps>

## ## Making changes

Once you have an idea of what you want to contribute, you can start making changes. Here are some steps to get started:

<Steps>

<Step>

### 1. Create a new branch

```
```bash
# Make sure you're on main
git checkout main

# Pull latest changes
git pull upstream main

# Create and switch to a new branch
git checkout -b feature/your-feature-name
```
```

</Step>

<Step>

### 2. Start development server

Start the development server:

```
```bash
pnpm dev
```
```



```
```
```

To start the docs server:

```
```bash
pnpm -F docs dev
```
```

</Step>

<Step>

### ### 3. Make Your Changes

- \* Make your changes to the codebase.

- \* Write tests if needed. (Read more about testing <Link href="/docs/reference/contribute/testing">here</Link>)

- \* Update documentation. (Read more about documenting <Link href="/docs/reference/contribute/documenting">here</Link>)

</Step>

</Steps>

### ### Issues and Bug Fixes

- \* Check our [GitHub issues](https://github.com/better-auth/better-auth/issues) for tasks labeled `good first issue`
- \* When reporting bugs, include steps to reproduce and expected behavior
- \* Comment on issues you'd like to work on to avoid duplicate efforts

### ### Framework Integrations

We welcome contributions to support more frameworks:

- \* Focus on framework-agnostic solutions where possible
- \* Keep integrations minimal and maintainable
- \* All integrations currently live in the main package

### ### Plugin Development

- \* For core plugins: Open an issue first to discuss your idea
- \* For community plugins: Feel free to develop independently
- \* Follow our plugin architecture guidelines

### ### Documentation

- \* Fix typos and errors
- \* Add examples and clarify existing content
- \* Ensure documentation is up to date with code changes

## ## Testing

We use Vitest for testing. Place test files next to the source files they test:

```
```ts
import { describe, it, expect } from "vitest";
import { getTestInstance } from "../test-utils/test-instance";

describe("Feature", () => {
  it("should work as expected", async () => {
    const { client } = getTestInstance();
    // Test code here
    expect(result).toBeDefined();
  });
});
```
```

### ### Testing Best Practices

- \* Write clear commit messages
- \* Update documentation to reflect your changes
- \* Add tests for new features
- \* Follow our coding standards
- \* Keep pull requests focused on a single change

### ## Need Help?

Don't hesitate to ask for help! You can:

- \* Open an [issue](https://github.com/better-auth/better-auth/issues) with questions
- \* Join our [community discussions](https://discord.gg/better-auth)
- \* Reach out to project maintainers

Thank you for contributing to Better Auth!

file: ./content/docs/reference/faq.mdx

```
meta: {
  "title": "FAQ",
  "description": "Frequently asked questions about Better Auth."
}
```

This page contains frequently asked questions, common issues, and other helpful information about Better Auth.

<Accordions>

<Accordion title="Auth client not working">

When encountering `createAuthClient` related errors, make sure to have the correct import path as it varies based on environment.

If you're using the auth client on react front-end, you'll need to import it from `/react`:

```
```ts title="component.ts"
import { createAuthClient } from "better-auth/react";
```
```

Where as if you're using the auth client in Next.js middleware, server-actions, server-components or anything server-related, you'll likely need to import it from `/client`:

```
```ts title="server.ts"
import { createAuthClient } from "better-auth/client";
```
```

</Accordion>

<Accordion title="getSession not working">

If you try to call `authClient.getSession` on a server environment (e.g, a Next.js server component), it doesn't work since it can't access the cookies. You can use the `auth.api.getSession` instead and pass the request headers to it.

```
```tsx title="server.tsx"
import { auth } from "./auth";
import { headers } from "next/headers";

const session = await auth.api.getSession({
  headers: await headers()
})
```
```

if you need to use the auth client on the server for different purposes, you still can pass the request headers to it:

```
```tsx title="server.tsx"
import { authClient } from "./auth-client";
import { headers } from "next/headers";

const session = await authClient.getSession({
```

```

    fetchOptions:{
      headers: await headers()
    }
  })
  ...
</Accordion>

<Accordion title="Adding custom fields to the users table">
  Better Auth provides a type-safe way to extend the user and session schemas, take a look at our docs on <Link
href="/docs/concepts/database#extending-core-schema">extending core schema</Link>.
</Accordion>

<Accordion title="Difference between getSession and useSession">
  Both `useSession` and `getSession` instances are used fundamentally different based on the situation.

  `useSession` is a hook, meaning it can trigger re-renders whenever session data changes.

  If you have UI you need to change based on user or session data, you can use this hook.

  <Callout type="warn">
    For performance reasons, do not use this hook on your `layout.tsx` file. We
    recommend using RSC and use your server auth instance to get the session data
    via `auth.api.getSession`.
  </Callout>
</Accordion>

  `getSession` returns a promise containing data and error.

  For all other situations where you shouldn't use `useSession`, is when you should be using `getSession`.

  <Callout type="info">
    `getSession` is available on both server and client auth instances.
    Not just the latter.
  </Callout>
</Accordion>

<Accordion title="Common TypeScript Errors">
  If you're facing typescript errors, make sure your tsconfig has `strict` set to `true`:

  ```json title="tsconfig.json"
  {
    "compilerOptions": {
      "strict": true,
    }
  }
  ```

  if you can't set strict to true, you can enable strictNullChecks:

  ```json title="tsconfig.json"
  {
    "compilerOptions": {
      "strictNullChecks": true,
    }
  }
  ```

  You can learn more in our <Link href="/docs/concepts/typescript#typescript-config">TypeScript docs</Link>.
</Accordion>
</Accordions>

```

```

file: ./content/docs/reference/options.mdx
meta: {
  "title": "Options",
  "description": "Better Auth configuration options reference."
}

```

```
}
```

List of all the available options for configuring Better Auth. See [Better Auth Options](https://github.com/better-auth/better-auth/blob/main/packages/better-auth/src/types/options.ts#L13).

### ## `appName`

The name of the application.

```
```ts
import { betterAuth } from "better-auth";
export const auth = betterAuth({
  appName: "My App",
})
```
```

### ## `baseURL`

Base URL for Better Auth. This is typically the root URL where your application server is hosted. Note: If you include a path in the baseURL, it will take precedence over the default path.

```
```ts
import { betterAuth } from "better-auth";
export const auth = betterAuth({
  baseURL: "https://example.com",
})
```
```

If not explicitly set, the system will check for the environment variable `process.env.BETTER\_AUTH\_URL`

### ## `basePath`

Base path for Better Auth. This is typically the path where the Better Auth routes are mounted. It will be overridden if there is a path component within `baseURL`.

```
```ts
import { betterAuth } from "better-auth";
export const auth = betterAuth({
  basePath: "/api/auth",
})
```
```

Default: `/api/auth`

### ## `trustedOrigins`

List of trusted origins.

```
```ts
import { betterAuth } from "better-auth";
export const auth = betterAuth({
  trustedOrigins: ["http://localhost:3000", "https://example.com"],
})
```
```

### ## `secret`

The secret used for encryption, signing, and hashing.

```
```ts
import { betterAuth } from "better-auth";
export const auth = betterAuth({
  secret: "your-secret-key",
})
```
```

By default, Better Auth will look for the following environment variables:

```
* `process.env.BETTER_AUTH_SECRET`
* `process.env.AUTH_SECRET`
```

If none of these environment variables are set, it will default to ``"better-auth-secret-123456789"``. In production, if it's not set, it will throw an error.

You can generate a good secret using the following command:

```
` `` bash
openssl rand -base64 32
` ``
```

```
### `database`
```

Database configuration for Better Auth.

```
` `` ts
import { betterAuth } from "better-auth";
export const auth = betterAuth({
  database: {
    dialect: "postgres",
    type: "postgres",
    casing: "camel"
  },
})
` ``
```

Better Auth supports various database configurations including [PostgreSQL](/docs/adapters/postgresql), [MySQL](/docs/adapters/mysql), and [SQLite](/docs/adapters/sqlite).

Read more about databases [here](/docs/concepts/database).

```
### `secondaryStorage`
```

Secondary storage configuration used to store session and rate limit data.

```
` `` ts
import { betterAuth } from "better-auth";

export const auth = betterAuth({
  // ... other options
  secondaryStorage: {
    // Your implementation here
  },
})
` ``
```

Read more about secondary storage [here](/docs/concepts/database#secondary-storage).

```
### `emailVerification`
```

Email verification configuration.

```
` `` ts
import { betterAuth } from "better-auth";
export const auth = betterAuth({
  emailVerification: {
    sendVerificationEmail: async ({ user, url, token }) => {
      // Send verification email to user
    },
    sendOnSignUp: true,
    autoSignInAfterVerification: true,
  },
})
` ``
```

```

    expiresIn: 3600 // 1 hour
  },
})
```

```

\* `sendVerificationEmail`: Function to send verification email  
 \* `sendOnSignUp`: Send verification email automatically after sign up (default: `false`)  
 \* `autoSignInAfterVerification`: Auto sign in the user after they verify their email  
 \* `expiresIn`: Number of seconds the verification token is valid for (default: `3600` seconds)

## `emailAndPassword`

Email and password authentication configuration.

```

```ts
import { betterAuth } from "better-auth";
export const auth = betterAuth({
  emailAndPassword: {
    enabled: true,
    disableSignUp: false,
    requireEmailVerification: true,
    minPasswordLength: 8,
    maxPasswordLength: 128,
    autoSignIn: true,
    sendResetPassword: async ({ user, url, token }) => {
      // Send reset password email
    },
    resetPasswordTokenExpiresIn: 3600, // 1 hour
    password: {
      hash: async (password) => {
        // Custom password hashing
        return hashedPassword;
      },
      verify: async ({ hash, password }) => {
        // Custom password verification
        return isValid;
      }
    }
  },
})
```

```

\* `enabled`: Enable email and password authentication (default: `false`)  
 \* `disableSignUp`: Disable email and password sign up (default: `false`)  
 \* `requireEmailVerification`: Require email verification before a session can be created  
 \* `minPasswordLength`: Minimum password length (default: `8`)  
 \* `maxPasswordLength`: Maximum password length (default: `128`)  
 \* `autoSignIn`: Automatically sign in the user after sign up  
 \* `sendResetPassword`: Function to send reset password email  
 \* `resetPasswordTokenExpiresIn`: Number of seconds the reset password token is valid for (default: `3600` seconds)  
 \* `password`: Custom password hashing and verification functions

## `socialProviders`

Configure social login providers.

```

```ts
import { betterAuth } from "better-auth";
export const auth = betterAuth({
  socialProviders: {
    google: {
      clientId: "your-client-id",
      clientSecret: "your-client-secret",
      redirectUri: "https://example.com/api/auth/callback/google"
    },
  },
})
```

```

```

    github: {
      clientId: "your-client-id",
      clientSecret: "your-client-secret",
      redirectUri: "https://example.com/api/auth/callback/github"
    },
  },
})
```

```

### ## `plugins`

List of Better Auth plugins.

```

```ts
import { betterAuth } from "better-auth";
import { emailOTP } from "better-auth/plugins";

export const auth = betterAuth({
  plugins: [
    emailOTP({
      sendVerificationOTP: async ({ email, otp, type }) => {
        // Send OTP to user's email
      }
    })
  ],
})
```

```

### ## `user`

User configuration options.

```

```ts
import { betterAuth } from "better-auth";
export const auth = betterAuth({
  user: {
    modelName: "users",
    fields: {
      email: "emailAddress",
      name: "fullName"
    },
    additionalFields: {
      customField: {
        type: "string",
      }
    },
    changeEmail: {
      enabled: true,
      sendChangeEmailVerification: async ({ user, newEmail, url, token }) => {
        // Send change email verification
      }
    },
    deleteUser: {
      enabled: true,
      sendDeleteAccountVerification: async ({ user, url, token }) => {
        // Send delete account verification
      },
      beforeDelete: async (user) => {
        // Perform actions before user deletion
      },
      afterDelete: async (user) => {
        // Perform cleanup after user deletion
      }
    }
  },
})
```

```

```

  })
  ``

```

```

* `modelName`: The model name for the user (default: `"user"`)
* `fields`: Map fields to different column names
* `additionalFields`: Additional fields for the user table
* `changeEmail`: Configuration for changing email
* `deleteUser`: Configuration for user deletion

```

```

### `session`

```

Session configuration options.

```

````ts
import { betterAuth } from "better-auth";
export const auth = betterAuth({
  session: {
    modelName: "sessions",
    fields: {
      userId: "user_id"
    },
    expiresIn: 604800, // 7 days
    updateAge: 86400, // 1 day
    disableSessionRefresh: true, // Disable session refresh so that the session is not updated regardless of the
    `updateAge` option. (default: `false`)
    additionalFields: { // Additional fields for the session table
      customField: {
        type: "string",
      }
    },
    storeSessionInDatabase: true, // Store session in database when secondary storage is provided (default: `false`)
    preserveSessionInDatabase: false, // Preserve session records in database when deleted from secondary storage
    (default: `false`)
    cookieCache: {
      enabled: true, // Enable caching session in cookie (default: `false`)
      maxAge: 300 // 5 minutes
    }
  },
});
````

```

```

* `modelName`: The model name for the session (default: `"session"`)
* `fields`: Map fields to different column names
* `expiresIn`: Expiration time for the session token in seconds (default: `604800` - 7 days)
* `updateAge`: How often the session should be refreshed in seconds (default: `86400` - 1 day)
* `additionalFields`: Additional fields for the session table
* `storeSessionInDatabase`: Store session in database when secondary storage is provided (default: `false`)
* `preserveSessionInDatabase`: Preserve session records in database when deleted from secondary storage (default:
  `false`)
* `cookieCache`: Enable caching session in cookie

```

```

### `account`

```

Account configuration options.

```

````ts
import { betterAuth } from "better-auth";
export const auth = betterAuth({
  account: {
    modelName: "accounts",
    fields: {
      userId: "user_id"
    },
    accountLinking: {
      enabled: true,

```



```

    trustedProviders: ["google", "github", "email-password"],
    allowDifferentEmails: false
  },
},
))
```

```

\* `modelName`: The model name for the account  
 \* `fields`: Map fields to different column names

### ### `updateAccountOnSignIn`

If enabled (true), the user account data (accessToken, idToken, refreshToken, etc.) will be updated on sign in with the latest data from the provider.

### ### `accountLinking`

Configuration for account linking.

\* `enabled`: Enable account linking (default: `false`)  
 \* `trustedProviders`: List of trusted providers  
 \* `allowDifferentEmails`: Allow users to link accounts with different email addresses  
 \* `allowUnlinkingAll`: Allow users to unlink all accounts

### ## `verification`

Verification configuration options.

```

```ts
import { betterAuth } from "better-auth";
export const auth = betterAuth({
  verification: {
    modelName: "verifications",
    fields: {
      userId: "user_id"
    },
    disableCleanup: false
  },
});
```

```

\* `modelName`: The model name for the verification table  
 \* `fields`: Map fields to different column names  
 \* `disableCleanup`: Disable cleaning up expired values when a verification value is fetched

### ## `rateLimit`

Rate limiting configuration.

```

```ts
import { betterAuth } from "better-auth";
export const auth = betterAuth({
  rateLimit: {
    enabled: true,
    window: 10,
    max: 100,
    customRules: {
      "/example/path": {
        window: 10,
        max: 100
      }
    },
    storage: "memory",
    modelName: "rateLimit"
  },
});
```

```

```

  })
  ```

* `enabled`: Enable rate limiting (defaults: `true` in production, `false` in development)
* `window`: Time window to use for rate limiting. The value should be in seconds. (default: `10`)
* `max`: The default maximum number of requests allowed within the window. (default: `100`)
* `customRules`: Custom rate limit rules to apply to specific paths.
* `storage`: Storage configuration. If you passed a secondary storage, rate limiting will be stored in the secondary storage. (options: `"memory", "database", "secondary-storage"`, default: `"memory"`)
* `modelName`: The name of the table to use for rate limiting if database is used as storage. (default: `"rateLimit"`)

```

```

### `advanced`

```

Advanced configuration options.

```

```ts
import { betterAuth } from "better-auth";
export const auth = betterAuth({
  advanced: {
    ipAddress: {
      ipAddressHeaders: ["x-client-ip", "x-forwarded-for"],
      disableIpTracking: false
    },
    useSecureCookies: true,
    disableCSRFCheck: false,
    crossSubDomainCookies: {
      enabled: true,
      additionalCookies: ["custom_cookie"],
      domain: "example.com"
    },
    cookies: {
      session_token: {
        name: "custom_session_token",
        attributes: {
          httpOnly: true,
          secure: true
        }
      }
    },
    defaultCookieAttributes: {
      httpOnly: true,
      secure: true
    },
    cookiePrefix: "myapp",
    database: {
      // If your DB is using auto-incrementing IDs, set this to true.
      useNumberId: false,
      // Use your own custom ID generator, or disable generating IDs as a whole.
      generateId: (((options: {
        model: LiteralUnion<Models, string>;
        size?: number;
      }) => {
        return "my-super-unique-id";
      }) | false,
      defaultFindManyLimit: 100,
    }
  },
});
```

* `ipAddress`: IP address configuration for rate limiting and session tracking
* `useSecureCookies`: Use secure cookies (default: `false`)
* `disableCSRFCheck`: Disable trusted origins check (⚠️ security risk)
* `crossSubDomainCookies`: Configure cookies to be shared across subdomains
* `cookies`: Customize cookie names and attributes

```

- \* `defaultCookieAttributes`: Default attributes for all cookies
- \* `cookiePrefix`: Prefix for cookies
- \* `generateId`: Function to generate a unique ID for a model

### ## `databaseHooks`

Database lifecycle hooks for core operations.

```

```ts
import { betterAuth } from "better-auth";
export const auth = betterAuth({
  databaseHooks: {
    user: {
      create: {
        before: async (user) => {
          // Modify user data before creation
          return { data: { ...user, customField: "value" } };
        },
        after: async (user) => {
          // Perform actions after user creation
        }
      },
      update: {
        before: async (userData) => {
          // Modify user data before update
          return { data: { ...userData, updatedAt: new Date() } };
        },
        after: async (user) => {
          // Perform actions after user update
        }
      }
    },
    session: {
      // Session hooks
    },
    account: {
      // Account hooks
    },
    verification: {
      // Verification hooks
    }
  },
});
```

```

### ## `onAPIError`

API error handling configuration.

```

```ts
import { betterAuth } from "better-auth";
export const auth = betterAuth({
  onAPIError: {
    throw: true,
    onError: (error, ctx) => {
      // Custom error handling
      console.error("Auth error:", error);
    },
    errorURL: "/auth/error"
  },
});
```

```

- \* `throw`: Throw an error on API error (default: `false`)
- \* `onError`: Custom error handler

\* `errorURL`: URL to redirect to on error (default: `/api/auth/error`)

### ## `hooks`

Request lifecycle hooks.

```
```ts
import { betterAuth } from "better-auth";
export const auth = betterAuth({
  hooks: {
    before: async (request, ctx) => {
      // Execute before processing the request
    },
    after: async (request, response, ctx) => {
      // Execute after processing the request
    }
  },
})
```
```

### ## `disabledPaths`

Disable specific auth paths.

```
```ts
import { betterAuth } from "better-auth";
export const auth = betterAuth({
  disabledPaths: ["/sign-up/email", "/sign-in/email"],
})
```
```

### ## `customPaths`

Customize specific auth paths. This allows you to map any existing route to a custom path.

```
```ts
import { betterAuth } from "better-auth";
export const auth = betterAuth({
  customPaths: {
    "/ok": "/okay",
  },
})
```
```

file: ./content/docs/reference/resources.mdx

```
meta: {
  "title": "Resources",
  "description": "A curated collection of resources to help you learn and master Better Auth."
}
```

```
import { Resource } from "@components/resource-section";
```

A curated collection of resources to help you learn and master Better Auth. From blog posts to video tutorials, find everything you need to get started.

### ## Video tutorials

```
<Resource
  resources={
    [
      {
        title: "The State of Authentication",
        description:
          "<strong>Theo(t3.gg)</strong> explores the current landscape of authentication, discussing trends,
```

```
challenges, and where the industry is heading.",
  href: "https://www.youtube.com/watch?v=IxsInp-ZEMw",
  tags: ["trends", "showcase", "review"],
},
{
  title: "Last Authentication You Will Ever Need",
  description:
    "A comprehensive tutorial demonstrating why Better Auth could be the final authentication solution you'll
need for your projects.",
  href: "https://www.youtube.com/watch?v=hFtufpaMcLM",
  tags: ["implementation", "showcase"],
},
{
  title: "This Might Be My New Favourite Auth Library",
  description:
    "<strong>developedbyed</strong> explores the features and capabilities of Better Auth, explaining why it
stands out among authentication libraries.",
  href: "https://www.youtube.com/watch?v=Hjs3zM7o7NE",
  tags: ["review", "showcase"],
},
{
  title: "Best authentication framework for next.js",
  description:
    "A detailed comparison of authentication frameworks for Next.js, highlighting why Better Auth might be
your best choice.",
  href: "https://www.youtube.com/watch?v=V--T0q9FrEw",
  tags: ["nextjs", "comparison"],
},
{
  title: "Better-Auth: A First Look",
  description:
    "An introductory overview and demonstration of Better Auth's core features and capabilities.",
  href: "https://www.youtube.com/watch?v=2cQTV6NYxis",
  tags: ["implementation", "showcase"],
},
{
  title: "Stripe was never so easy (with better auth)",
  description: "A tutorial on how to integrate Stripe with Better Auth.",
  href: "https://www.youtube.com/watch?v=g-RIrzBEX6M",
  tags: ["implementation"],
},
{
  title: "Nextjs 15 Authentication Made EASY with Better Auth",
  description:
    "A practical guide showing how to seamlessly integrate Better Auth with Next.js 15 for robust
authentication.",
  href: "https://www.youtube.com/watch?v=IxsInp-ZEMw",
  tags: ["nextjs", "implementation", "tutorial"],
},
{
  title: "Better Auth: Headless Authentication for Your TanStack Start App",
  description: "<strong>Jack</strong> demonstrates how to implement headless authentication in your TanStack
Start application using Better Auth, providing a modern approach to auth.",
  href: "https://www.youtube.com/watch?v=Atev8Nxpw7c",
  tags: ["tanstack", "implementation"],
},
{
  title: "Goodbye Clerk, Hello Better Auth - Full Migration Guide!",
  description: "A comprehensive guide showing how to migrate your authentication from Clerk to Better Auth, with
step-by-step instructions and best practices.",
  href: "https://www.youtube.com/watch?v=Za_QihbDSuk",
  tags: ["migration", "clerk", "tutorial"],
},
]
```

```
}
/>
```

## ## Blog posts

```
<Resource
  resources={
    [
      {
        title: "Better Auth with Hono, Bun, TypeScript, React and Vite",
        description:
          "You'll learn how to implement authentication with Better Auth in a client - server architecture, where the
frontend is separate from the backend.",
        href: "https://catalins.tech/better-auth-with-hono-bun-typescript-react-vite",
        tags: ["typescript", "react", "bun", "vite"],
      },
      {
        title: "Polar.sh + BetterAuth for Organizations",
        description:
          "Polar.sh is a platform for building payment integrations. This article will show you how to use Better Auth to
authenticate your users.",
        href: "https://dev.to/phumudzosly/polarsh-betterauth-for-organizations-1j1b",
        tags: ["organizations", "integration", "payments"],
      },
      {
        title: "Authenticating users in Astro with Better Auth",
        description:
          "Step by step guide on how to authenticate users in Astro with Better Auth.",
        href: "https://www.launchfa.st/blog/astro-better-auth",
        tags: ["astro", "integration", "tutorial"],
      },
      {
        title: "Building Multi-Tenant Apps With Better-Auth and ZenStack",
        description:
          "Learn how to build multi-tenant apps with Better-Auth and ZenStack.",
        href: "https://zenstack.dev/blog/better-auth",
        tags: ["multi-tenant", "zenstack", "architecture"],
      },
    ]
  }
/>
```

```
file: ./content/docs/reference/security.mdx
meta: {
  "title": "Security",
  "description": "Better Auth security features."
}
```

This page contains information about security features of Better Auth.

## ## Password Hashing

Better Auth uses the `scrypt` algorithm to hash passwords by default. This algorithm is designed to be memory-hard and CPU-intensive, making it resistant to brute-force attacks. You can customize the password hashing function by setting the `password` option in the configuration. This option should include a `hash` function to hash passwords and a `verify` function to verify them.

## ## Session Management

### ### Session Expiration

Better Auth uses secure session management to protect user data. Sessions are stored in the database or a secondary storage, if configured, to prevent unauthorized access. By default, sessions expire after 7 days, but you can customize this value in the configuration. Additionally, each time a session is used, if it reaches the `updateAge` threshold, the expiration

date is extended, which by default is set to 1 day.

### ### Session Revocation

Better Auth allows you to revoke sessions to enhance security. When a session is revoked, the user is logged out and can no longer access the application. A logged in user can also revoke their own sessions to log out from different devices or browsers.

See the [session management](/docs/concepts/session-management) for more details.

## ## CSRF Protection

Better Auth ensures CSRF protection by validating the Origin header in requests. This check confirms that requests originate from the application or a trusted source. If a request comes from an untrusted origin, it is blocked to prevent potential CSRF attacks. By default, the origin matching the base URL is trusted, but you can set a list of trusted origins in the trustedOrigins configuration option.

## ## OAuth State and PKCE

To secure OAuth flows, Better Auth stores the OAuth state and PKCE (Proof Key for Code Exchange) in the database. The state helps prevent CSRF attacks, while PKCE protects against code injection threats. Once the OAuth process completes, these values are removed from the database.

## ## Cookies

Better Auth assigns secure cookies by default when the base URL uses `https`. These secure cookies are encrypted and only sent over secure connections, adding an extra layer of protection. They are also set with the `sameSite` attribute to `lax` by default to prevent cross-site request forgery attacks. And the `httpOnly` attribute is enabled to prevent client-side JavaScript from accessing the cookie.

For Cross-Subdomain Cookies, you can set the `crossSubDomainCookies` option in the configuration. This option allows cookies to be shared across subdomains, enabling seamless authentication across multiple subdomains.

### ### Customizing Cookies

You can customize cookie names to minimize the risk of fingerprinting attacks and set specific cookie options as needed for additional control. For more information, refer to the [cookie options](/docs/concepts/cookies).

Plugins can also set custom cookie options to align with specific security needs. If you're using Better Auth in non-browser environments, plugins offer ways to manage cookies securely in those contexts as well.

## ## Rate Limiting

Better Auth includes built-in rate limiting to safeguard against brute-force attacks. Rate limits are applied across all routes by default, with specific routes subject to stricter limits based on potential risk.

## ## IP Address Headers

Better Auth uses client IP addresses for rate limiting and security monitoring. By default, it reads the IP address from the standard `X-Forwarded-For` header. However, you can configure a specific trusted header to ensure accurate IP address detection and prevent IP spoofing attacks.

You can configure the IP address header in your Better Auth configuration:

```
```typescript
{
  advanced: {
    ipAddress: {
      ipAddressHeaders: ['cf-connecting-ip'] // or any other custom header
    }
  }
}
```
```

This ensures that Better Auth only accepts IP addresses from your trusted proxy's header, making it more difficult for

attackers to bypass rate limiting or other IP-based security measures by spoofing headers.

➤ **Important**: When setting a custom IP address header, ensure that your proxy or load balancer is properly configured to set this header, and that it cannot be set by end users directly.

## ## Trusted Origins

Trusted origins prevent CSRF attacks and block open redirects. You can set a list of trusted origins in the `trustedOrigins` configuration option. Requests from origins not on this list are automatically blocked.

### ### Basic Usage

The most basic usage is to specify exact origins:

```
```typescript
{
  trustedOrigins: [
    "https://example.com",
    "https://app.example.com",
    "http://localhost:3000"
  ]
}
```

### ### Wildcard Domains

Better Auth supports wildcard patterns in trusted origins, which allows you to trust multiple subdomains with a single entry:

```
```typescript
{
  trustedOrigins: [
    "*.example.com", // Trust all subdomains of example.com (any protocol)
    "https://*.example.com", // Trust only HTTPS subdomains of example.com
    "http://*.dev.example.com" // Trust all HTTP subdomains of dev.example.com
  ]
}
```

### #### Protocol-specific wildcards

When using a wildcard pattern with a protocol prefix (like `https://`):

- \* The protocol must match exactly
- \* The domain can have any subdomain in place of the `\*`
- \* Requests using a different protocol will be rejected, even if the domain matches

### #### Protocol-agnostic wildcards

When using a wildcard pattern without a protocol prefix (like `\*.example.com`):

- \* Any protocol (http, https, etc.) will be accepted
- \* The domain must match the wildcard pattern

## ### Custom Schemes

Trusted origins also support custom schemes for mobile apps and browser extensions:

```
```typescript
{
  trustedOrigins: [
    "myapp://", // Mobile app scheme
    "chrome-extension://YOUR_EXTENSION_ID" // Browser extension
  ]
}
```



## ## Reporting Vulnerabilities

If you discover a security vulnerability in Better Auth, please report it to us at [security@better-auth.com] (mailto:security@better-auth.com). We address all reports promptly, and credits will be given for validated discoveries.