Version: v4

OAuth

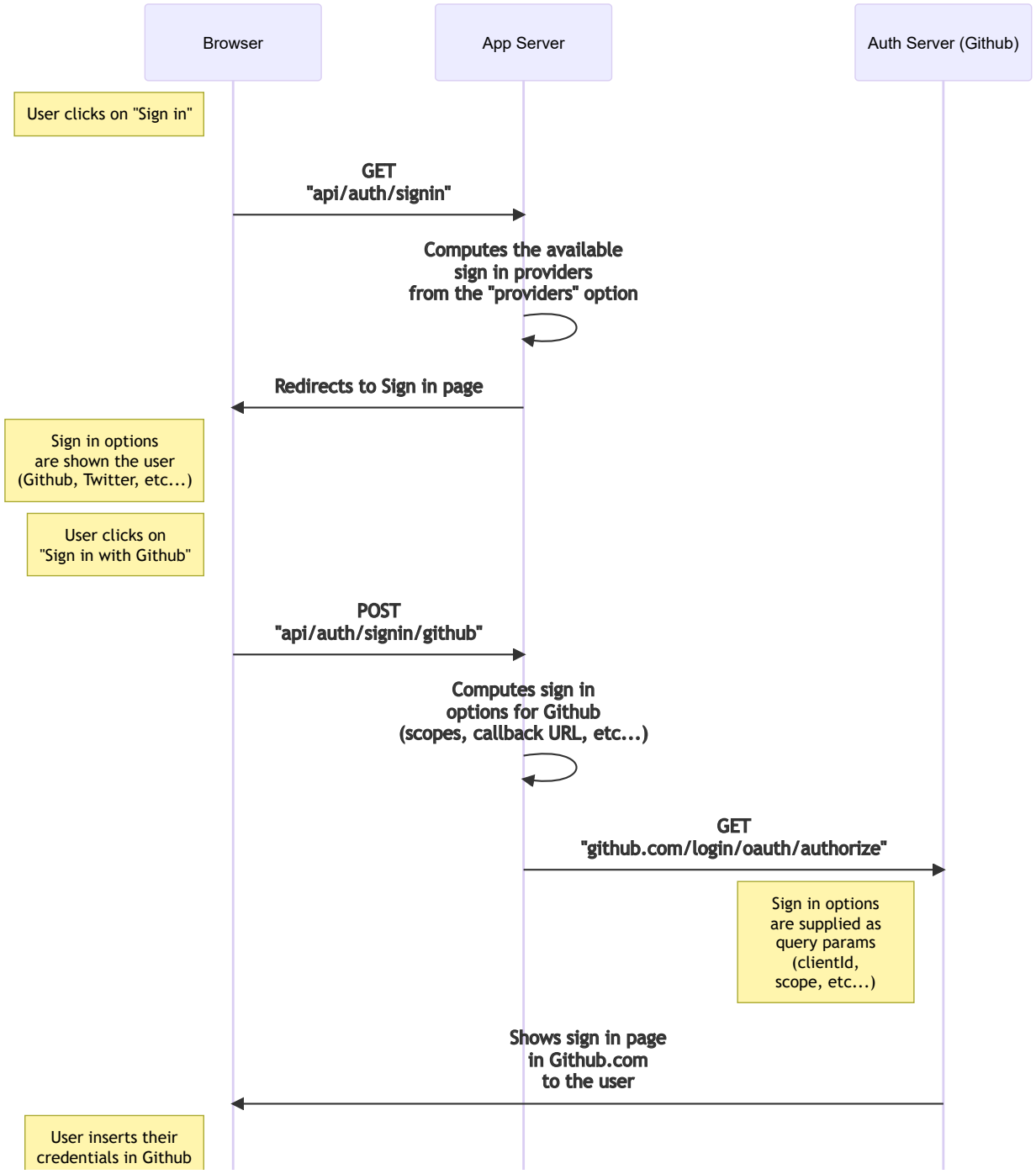
Authentication Providers in **NextAuth.js** are OAuth definitions that allow your users to sign in with their favorite preexisting logins. You can use any of our many predefined providers, or write your own custom OAuth configuration.

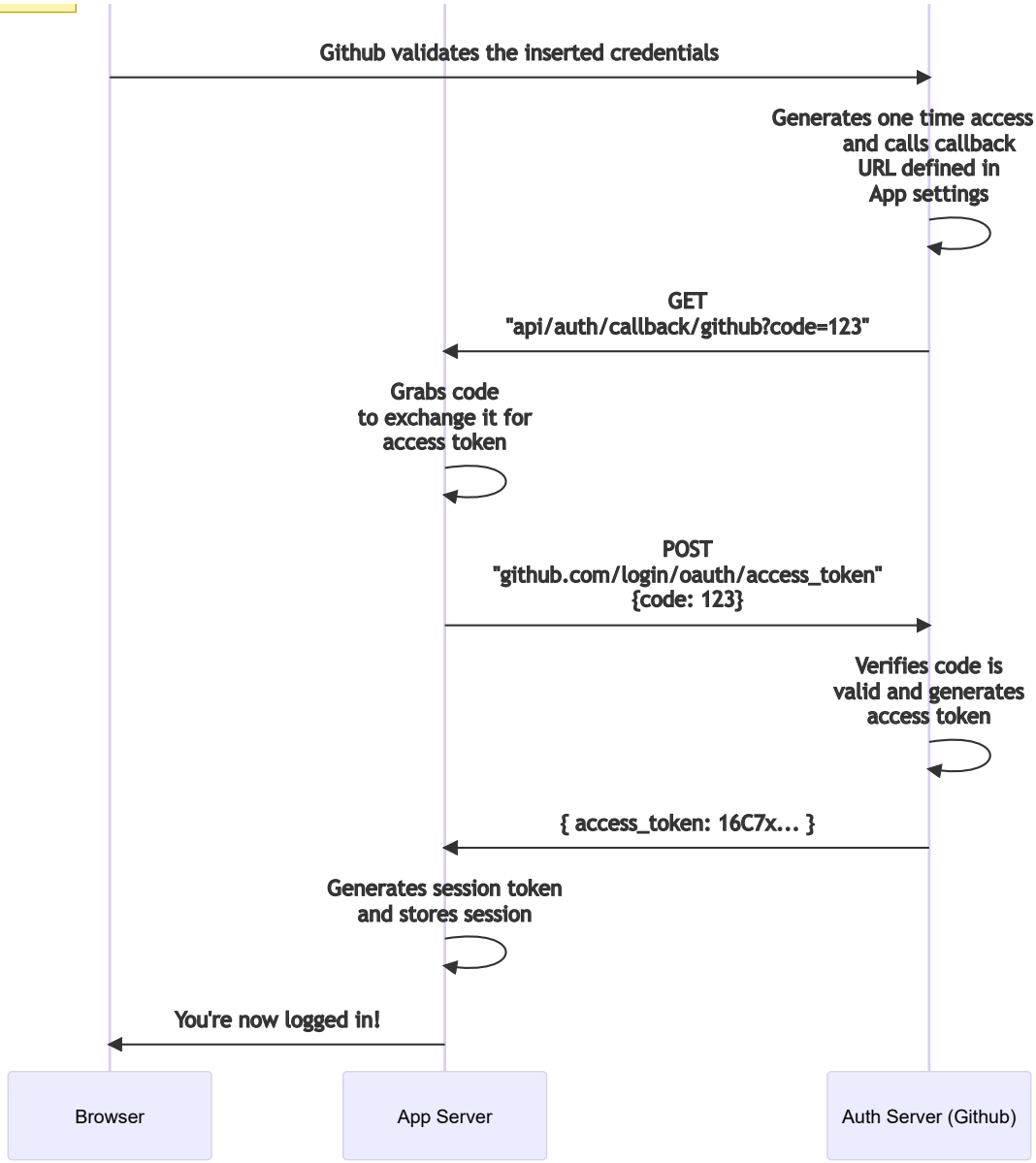
- Using a built-in OAuth Provider (e.g Github, Twitter, Google, etc...)
- Using a custom OAuth Provider

> **NOTE**
>
> NextAuth.js is designed to work with any OAuth service, it supports **OAuth 1.0**, **1.0A**, **2.0** and **OpenID Connect** and has built-in support for most popular sign-in services.

Without going into too much detail, the OAuth flow generally has 6 parts:

1. The application requests authorization to access service resources from the user
2. If the user authorized the request, the application receives an authorization grant
3. The application requests an access token from the authorization server (API) by presenting authentication of its own identity, and the authorization grant
4. If the application identity is authenticated and the authorization grant is valid, the authorization server (API) issues an access token to the application. Authorization is complete.
5. The application requests the resource from the resource server (API) and presents the access token for authentication
6. If the access token is valid, the resource server (API) serves the resource to the application


Browser
App Server
Auth Server (Github)
User clicks on "Sign in"
GET
"api/auth/signin"
Computes the available
sign in providers
from the "providers" option
Redirects to Sign in page
Sign in options
are shown the user
(Github, Twitter, etc...)
User clicks on
"Sign in with Github"
POST
"api/auth/signin/github"
Computes sign in
options for Github
(scopes, callback URL, etc...)
GET
"github.com/login/oauth/authorize"
Sign in options
are supplied as
query params
(clientId,
scope, etc...)
Shows sign in page
in Github.com
to the user
User inserts their
credentials in Github



For more details, check out Aaron Parecki's blog post OAuth2 Simplified or Postman's blog post OAuth 2.0: Implicit Flow is Dead, Try PKCE Instead.

How to

1. Register your application at the developer portal of your provider. There are usually links to the portals included in the aforementioned documentation pages for each supported provider with details on how to register your application.

2. The redirect URI (sometimes called Callback URL) should follow this format:

```
[origin]/api/auth/callback/[provider]
```

`[provider]` refers to the `id` of your provider (see **options**). For example, Twitter on `localhost` this would be:

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

Using Google in our example application would look like this:

```
https://next-auth-example.vercel.app/api/auth/callback/google
```

3. Create a `.env` file at the root of your project and add the client ID and client secret. For Twitter this would be:

```
TWITTER_ID=YOUR_TWITTER_CLIENT_ID
TWITTER_SECRET=YOUR_TWITTER_CLIENT_SECRET
```

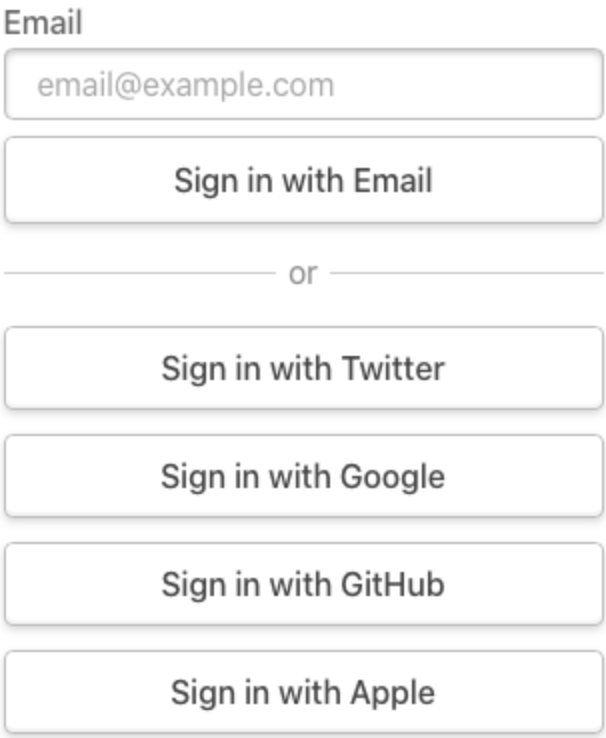
4. Now you can add the provider settings to the NextAuth.js options object. You can add as many OAuth providers as you like, as you can see `providers` is an array.

pages/api/auth/[...nextauth].js

```
import TwitterProvider from "next-auth/providers/twitter"
...
providers: [
  TwitterProvider({
    clientId: process.env.TWITTER_ID,
    clientSecret: process.env.TWITTER_SECRET
  })
],
...
```

5. Once a provider has been setup, you can sign in at the following URL: `[origin]/api/auth/signin`. This is an unbranded auto-generated page with all the

configured providers.



Options

Whenever you configure a custom or a built-in OAuth provider, you have the following options available:

```
interface OAuthConfig {
  /**
   * OpenID Connect (OIDC) compliant providers can configure
   * this instead of `authorize`/`token`/`userinfo` options
   * without further configuration needed in most cases.
   * You can still use the `authorize`/`token`/`userinfo`
   * options for advanced control.
   *
   * [Authorization Server Metadata]
(https://datatracker.ietf.org/doc/html/rfc8414#section-3)
   */
  wellKnown?: string
  /**
   * The login process will be initiated by sending the user to this URL.
```

```
   *
   * [Authorization endpoint]
(https://datatracker.ietf.org/doc/html/rfc6749#section-3.1)
   */
  authorization: EndpointHandler<AuthorizationParameters>
  /**
   * Endpoint that returns OAuth 2/OIDC tokens and information about them.
   * This includes `access_token`, `id_token`, `refresh_token`, etc.
   *
   * [Token endpoint](https://datatracker.ietf.org/doc/html/rfc6749#section-3.2)
   */
  token: EndpointHandler<
    UrlParams,
    {
      /**
       * Parameters extracted from the request to the
`/api/auth/callback/:providerId` endpoint.
       * Contains params like `state`.
       */
      params: CallbackParamsType
      /**
       * When using this custom flow, make sure to do all the necessary security
checks.
       * This object contains parameters you have to match against the request to
make sure it is valid.
       */
      checks: OAuthChecks
    },
    { tokens: TokenSet }
  >
  /**
   * When using an OAuth 2 provider, the user information must be requested
   * through an additional request from the userinfo endpoint.
   *
   * [Userinfo endpoint](https://www.oauth.com/oauth2-servers/signing-in-with-
google/verifying-the-user-info)
   */
  userinfo?: EndpointHandler<UrlParams, { tokens: TokenSet }, Profile>
  type: "oauth"
  /**
   * Used in URLs to refer to a certain provider.
   * @example /api/auth/callback/twitter // where the `id` is "twitter"
   */
  id: string
```

```
  version: string
  profile(profile: P, tokens: TokenSet): Awaitable<User>
  checks?: ChecksType | ChecksType[]
  clientId: string
  clientSecret: string
  /**
   * If set to `true`, the user information will be extracted
   * from the `id_token` claims, instead of
   * making a request to the `userinfo` endpoint.
   *
   * `id_token` is usually present in OpenID Connect (OIDC) compliant providers.
   *
   * [`id_token` explanation](https://www.oauth.com/oauth2-servers/openid-
connect/id-tokens)
   */
  idToken?: boolean
  region?: string
  issuer?: string
  client?: Partial<ClientMetadata>
  allowDangerousEmailAccountLinking?: boolean
  /**
   * Object containing the settings for the styling of the providers sign-in
buttons
   */
  style: ProviderStyleType
}
```

`authorization` option

Configure how to construct the request to the *Authorization endpoint*.

There are two ways to use this option:

1. You can either set `authorization` to be a full URL, like `"https://example.com/oauth/authorization?scope=email"`.
2. Use an object with `url` and `params` like so

```
authorization: {
  url: "https://example.com/oauth/authorization",
  params: { scope: "email" }
}
```

> **TIP**
>
> If your Provider is OpenID Connect (OIDC) compliant, we recommend using the `wellKnown` option instead.

`token` option

Configure how to construct the request to the *Token endpoint*.

There are three ways to use this option:

1. You can either set `token` to be a full URL, like `"https://example.com/oauth/token?some=param"`.
2. Use an object with `url` and `params` like so

```
token: {
  url: "https://example.com/oauth/token",
  params: { some: "param" }
}
```

3. Completely take control of the request:

```
token: {
  url: "https://example.com/oauth/token",
  async request(context) {
    // context contains useful properties to help you make the request.
    const tokens = await makeTokenRequest(context)
    return { tokens }
  }
}
```

> **DANGER**
>
> Option 3. should not be necessary in most cases, but if your provider does not follow the spec, or you have some very unique constraints it can be useful. Try to avoid it, if possible.

> **TIP**

> If your Provider is OpenID Connect (OIDC) compliant, we recommend using the `wellKnown` option instead.

`userinfo` option

A `userinfo` endpoint returns information about the logged-in user. It is not part of the OAuth specification, but usually available for most providers.

There are three ways to use this option:

1. You can either set `userinfo` to be a full URL, like `"https://example.com/oauth/userinfo?some=param"`.
2. Use an object with `url` and `params` like so

```
userinfo: {
  url: "https://example.com/oauth/userinfo",
  params: { some: "param" }
}
```

3. Completely take control of the request:

```
userinfo: {
  url: "https://example.com/oauth/userinfo",
  // The result of this method will be the input to the `profile` callback.
  async request(context) {
    // context contains useful properties to help you make the request.
    return await makeUserinfoRequest(context)
  }
}
```

> **DANGER**
>
> Option 3. should not be necessary in most cases, but if your provider does not follow the spec, or you have some very unique constraints it can be useful. Try to avoid it, if possible.

> **TIP**

In the rare case you don't care about what this endpoint returns, or your provider does not have one, you could create a noop function:

```
userinfo: {
  request: () => {}
}
```

TIP

If your Provider is OpenID Connect (OIDC) compliant, we recommend using the `wellKnown` option instead. OIDC usually returns an `id_token` from the `token` endpoint. `next-auth` can decode the `id_token` to get the user information, instead of making an additional request to the `userinfo` endpoint. Just set `idToken: true` at the top-level of your provider configuration. If not set, `next-auth` will still try to contact this endpoint.

`client` option

An advanced option, hopefully you won't need it in most cases. `next-auth` uses `openid-client` under the hood, see the docs on this option here.

`allowDangerousEmailAccountLinking` option

Normally, when you sign in with an OAuth provider and another account with the same email address already exists, the accounts are not linked automatically. Automatic account linking on sign in is not secure between arbitrary providers and is disabled by default (see our Security FAQ). However, it may be desirable to allow automatic account linking if you trust that the provider involved has securely verified the email address associated with the account. Just set `allowDangerousEmailAccountLinking: true` in your provider configuration to enable automatic account linking.

If the user is already signed in with any provider, when using `signIn` again via a different provider, the new provider account *will* be linked automatically to the same authenticated user. This happens regardless of the primary emails for each provider accounts. This flow is not affected by the `allowDangerousEmailAccountLinking` option.

Using a custom provider

You can use an OAuth provider that isn't built-in by using a custom object.

As an example of what this looks like, this is the provider object returned for the Google provider:

```
{
  id: "google",
  name: "Google",
  type: "oauth",
  wellKnown: "https://accounts.google.com/.well-known/openid-configuration",
  authorization: { params: { scope: "openid email profile" } },
  idToken: true,
  checks: ["pkce", "state"],
  profile(profile) {
    return {
      id: profile.sub,
      name: profile.name,
      email: profile.email,
      image: profile.picture,
    }
  },
}
```

As you can see, if your provider supports OpenID Connect and the `/.well-known/openid-configuration` endpoint contains support for the `grant_type`: `authorization_code`, you only need to pass the URL to that configuration file and define some basic fields like `name` and `type`.

Otherwise, you can pass a more full set of URLs for each OAuth2.0 flow step, for example:

```
{
  id: "kakao",
  name: "Kakao",
  type: "oauth",
  authorization: "https://kauth.kakao.com/oauth/authorize",
  token: "https://kauth.kakao.com/oauth/token",
  userinfo: "https://kapi.kakao.com/v2/user/me",
  profile(profile) {
    return {
      id: profile.id,
```

```
      name: profile.kakao_account?.profile.nickname,
      email: profile.kakao_account?.email,
      image: profile.kakao_account?.profile.profile_image_url,
    }
  },
}
```

Replace all the options in this JSON object with the ones from your custom provider - be sure to give it a unique ID and specify the required URLs, and finally add it to the providers array when initializing the library:

pages/api/auth/[...nextauth].js

```
import TwitterProvider from "next-auth/providers/twitter"
...
providers: [
  TwitterProvider({
    clientId: process.env.TWITTER_ID,
    clientSecret: process.env.TWITTER_SECRET,
  }),
  {
    id: 'customProvider',
    name: 'CustomProvider',
    type: 'oauth',
    scope: ''  // Make sure to request the users email address
    ...
  }
]
...
```

Built-in providers

NextAuth.js comes with a set of built-in providers. You can find them here. Each built-in provider has its own documentation page:

42 School, Amazon Cognito, Apple, Atlassian, Auth0, Authentik, Azure Active Directory, Azure Active Directory B2C, Battle.net, Box, BoxyHQ SAML, Bungie, Coinbase, Discord, Dropbox, DuendeIdentityServer6, EVE Online, Facebook, FACEIT, Foursquare, Freshbooks, FusionAuth, GitHub, GitLab, Google, HubSpot, IdentityServer4, Instagram, Kakao, Keycloak, LINE, LinkedIn,

Mail.ru, Mailchimp, Medium, Naver, Netlify, Okta, OneLogin, Osso, osu!, Patreon, Pinterest, Pipedrive, Reddit, Salesforce, Slack, Spotify, Strava, Todoist, Trakt, Twitch, Twitter, United Effects, VK, Wikimedia, WordPress.com, WorkOS, Yandex, Zitadel, Zoho, Zoom,

Override default options

For built-in providers, in most cases you will only need to specify the `clientId` and `clientSecret`. If you need to override any of the defaults, add your own options.

Even if you are using a built-in provider, you can override any of these options to tweak the default configuration.

> **NOTE**
>
> The user provided options are deeply merged with the default options. That means you only have to override part of the options that you need to be different. For example if you want different scopes, overriding `authorization.params.scope` is enough, instead of the whole `authorization` option.

/api/auth/[...nextauth].js

```
import Auth0Provider from "next-auth/providers/auth0"

Auth0Provider({
  clientId: process.env.CLIENT_ID,
  clientSecret: process.env.CLIENT_SECRET,
  issuer: process.env.ISSUER,
  authorization: { params: { scope: "openid your_custom_scope" } },
})
```

Another example, the `profile` callback will return `id`, `name`, `email` and `picture` by default, but you might need more information from the provider. After setting the correct scopes, you can then do something like this:

/api/auth/[...nextauth].js

```
import GoogleProvider from "next-auth/providers/google"

GoogleProvider({
  clientId: process.env.GOOGLE_CLIENT_ID,
  clientSecret: process.env.GOOGLE_CLIENT_SECRET,
  profile(profile) {
    return {
      // Return all the profile information you need.
      // The only truly required field is `id`
      // to be able identify the account when added to a database
    }
  },
})
```

An example of how to enable automatic account linking:

/api/auth/[...nextauth].js

```
import GoogleProvider from "next-auth/providers/google"

GoogleProvider({
  clientId: process.env.GOOGLE_CLIENT_ID,
  clientSecret: process.env.GOOGLE_CLIENT_SECRET,
  allowDangerousEmailAccountLinking: true,
})
```

Edit this page

*Last updated on **May 16, 2024** by **Codie Newark***