Version: v4

Callbacks

Callbacks are **asynchronous** functions you can use to control what happens when an action is performed.

Callbacks are extremely powerful, especially in scenarios involving JSON Web Tokens as they allow you to implement access controls without a database and to integrate with external databases or APIs.

TIP

If you want to pass data such as an Access Token or User ID to the browser when using JSON Web Tokens, you can persist the data in the token when the `jwt` callback is called, then pass the data through to the browser in the `session` callback.

You can specify a handler for any of the callbacks below.

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

```
...
  callbacks: {
    async signIn({ user, account, profile, email, credentials }) {
      return true
    },
    async redirect({ url, baseUrl }) {
      return baseUrl
    },
    async session({ session, user, token }) {
      return session
    },
    async jwt({ token, user, account, profile, isNewUser }) {
      return token
    }
...
}
```

The documentation below shows how to implement each callback, their default behaviour and an example of what the response for each callback should be. Note that configuration options and authentication providers you are using can impact the values passed to the callbacks.

Sign in callback

Use the `signIn()` callback to control if a user is allowed to sign in.

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

```
...
callbacks: {
  async signIn({ user, account, profile, email, credentials }) {
    const isAllowedToSignIn = true
    if (isAllowedToSignIn) {
      return true
    } else {
      // Return false to display a default error message
      return false
      // Or you can return a URL to redirect to:
      // return '/unauthorized'
    }
  }
}
...
```

- When using the **Email Provider** the `signIn()` callback is triggered both when the user makes a **Verification Request** (before they are sent an email with a link that will allow them to sign in) and again *after* they activate the link in the sign-in email.

  Email accounts do not have profiles in the same way OAuth accounts do. On the first call during email sign in the `email` object will include a property `verificationRequest: true` to indicate it is being triggered in the verification request flow. When the callback is invoked *after* a user has clicked on a sign-in link, this property will not be present.

You can check for the `verificationRequest` property to avoid sending emails to addresses or domains on a blocklist (or to only explicitly generate them for email address in an allow list).

- When using the **Credentials Provider** the `user` object is the response returned from the `authorize` callback and the `profile` object is the raw body of the `HTTP POST` submission.

> **NOTE**
>
> When using NextAuth.js with a database, the User object will be either a user object from the database (including the User ID) if the user has signed in before or a simpler prototype user object (i.e. name, email, image) for users who have not signed in before.
>
> When using NextAuth.js without a database, the user object will always be a prototype user object, with information extracted from the profile.

> **NOTE**
>
> Redirects returned by this callback cancel the authentication flow. Only redirect to error pages that, for example, tell the user why they're not allowed to sign in.
>
> To redirect to a page after a successful sign in, please use the `callbackUrl` option or the redirect callback.

Redirect callback

The redirect callback is called anytime the user is redirected to a callback URL (e.g. on signin or signout).

By default only URLs on the same URL as the site are allowed, you can use the redirect callback to customise that behaviour.

The default redirect callback looks like this:

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

```
...
callbacks: {
    async redirect({ url, baseUrl }) {
      // Allows relative callback URLs
      if (url.startsWith("/")) return `${baseUrl}${url}`
      // Allows callback URLs on the same origin
      else if (new URL(url).origin === baseUrl) return url
      return baseUrl
    }
  }
...
```

NOTE

The redirect callback may be invoked more than once in the same flow.

JWT callback

This callback is called whenever a JSON Web Token is created (i.e. at sign in) or updated (i.e whenever a session is accessed in the client). The returned value will be encrypted, and it is stored in a cookie.

Requests to `/api/auth/signin`, `/api/auth/session` and calls to `getSession()`, `getServerSession()`, `useSession()` will invoke this function, but only if you are using a JWT session. This method is not invoked when you persist sessions in a database.

- As with database persisted session expiry times, token expiry time is extended whenever a session is active.
- The arguments *user*, *account*, *profile* and *isNewUser* are only passed the first time this callback is called on a new session, after the user signs in. In subsequent calls, only `token` will be available.

The contents *user*, *account*, *profile* and *isNewUser* will vary depending on the provider and if you are using a database. You can persist data such as User ID, OAuth Access Token in this token, see the example below for `access_token` and `user.id`. To expose it on the client side, check out the `session()` callback as well.

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

```
...
callbacks: {
  async jwt({ token, account, profile }) {
    // Persist the OAuth access_token and or the user id to the token right after
signin
    if (account) {
      token.accessToken = account.access_token
      token.id = profile.id
    }
    return token
  }
}
...
```

TIP

Use an if branch to check for the existence of parameters (apart from `token`). If they exist, this means that the callback is being invoked for the first time (i.e. the user is being signed in). This is a good place to persist additional data like an `access_token` in the JWT. Subsequent invocations will only contain the `token` parameter.

Session callback

The session callback is called whenever a session is checked. By default, **only a subset of the token is returned for increased security**. If you want to make something available you added to the token (like `access_token` and `user.id` from above) via the `jwt()` callback, you have to explicitly forward it here to make it available to the client.

e.g. `getSession()`, `useSession()`, `/api/auth/session`

- When using database sessions, the User (`user`) object is passed as an argument.
- When using JSON Web Tokens for sessions, the JWT payload (`token`) is provided instead.

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

```
...
callbacks: {
  async session({ session, token, user }) {
    // Send properties to the client, like an access_token and user id from a
provider.
    session.accessToken = token.accessToken
    session.user.id = token.id

    return session
  }
}
...
```

TIP

When using JSON Web Tokens the `jwt()` callback is invoked before the `session()` callback, so anything you add to the JSON Web Token will be immediately available in the session callback, like for example an `access_token` or `id` from a provider.

DANGER

The session object is not persisted server side, even when using database sessions - only data such as the session token, the user, and the expiry time is stored in the session table.

If you need to persist session data server side, you can use the `accessToken` returned for the session as a key - and connect to the database in the `session()` callback to access it. Session `accessToken` values do not rotate and are valid as long as the session is valid.

If using JSON Web Tokens instead of database sessions, you should use the User ID or a unique key stored in the token (you will need to generate a key for this yourself on sign in, as access tokens for sessions are not generated when using JSON Web Tokens).

Edit this page

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