Version: v4

Next.js

`getServerSession`

TIP

You can create a helper function so you don't need to pass `authOptions` around:

auth.ts

```ts
import type {
  GetServerSidePropsContext,
  NextApiRequest,
  NextApiResponse,
} from "next"
import type { NextAuthOptions } from "next-auth"
import { getServerSession } from "next-auth"

// You'll need to import and pass this
// to `NextAuth` in `app/api/auth/[...nextauth]/route.ts`
export const config = {
  providers: [], // rest of your config
} satisfies NextAuthOptions

// Use it in server contexts
export function auth(
  ...args:
    | [GetServerSidePropsContext["req"], GetServerSidePropsContext["res"]]
    | [NextApiRequest, NextApiResponse]
    | []
) {
  return getServerSession(...args, config)
}
```

When calling from the server-side i.e. in Route Handlers, React Server Components, API routes or in `getServerSideProps`, we recommend using this function instead of `getSession` to retrieve the `session` object. This method is especially useful when you are using NextAuth.js with a database. This method can *drastically* reduce response time when used over `getSession` on server-side, due to avoiding an extra `fetch` to an API Route (this is generally **not recommended in Next.js**). In addition, `getServerSession` will correctly update the cookie expiry time and update the session content if `callbacks.jwt` or `callbacks.session` changed something.

`getServerSession` requires passing the same object you would pass to `NextAuth` when initializing NextAuth.js. To do so, you can export your NextAuth.js options in the following way:

In `[...nextauth].ts`:

```
import NextAuth from "next-auth"
import type { NextAuthOptions } from "next-auth"

export const authOptions: NextAuthOptions = {
  // your configs
}

export default NextAuth(authOptions)
```

In `getServerSideProps`:

```
import { authOptions } from "pages/api/auth/[...nextauth]"
import { getServerSession } from "next-auth/next"

export async function getServerSideProps(context) {
  const session = await getServerSession(context.req, context.res, authOptions)

  if (!session) {
    return {
      redirect: {
        destination: "/",
        permanent: false,
      },
    }
  }
```

```
  return {
    props: {
      session,
    },
  }
}
```

In API Routes:

```
import { authOptions } from "pages/api/auth/[...nextauth]"
import { getServerSession } from "next-auth/next"

export default async function handler(req, res) {
  const session = await getServerSession(req, res, authOptions)

  if (!session) {
    res.status(401).json({ message: "You must be logged in." })
    return
  }

  return res.json({
    message: "Success",
  })
}
```

In App Router:

You can also use `getServerSession` in Next.js' server components:

```
import { getServerSession } from "next-auth/next"
import { authOptions } from "pages/api/auth/[...nextauth]"

export default async function Page() {
  const session = await getServerSession(authOptions)
  return <pre>{JSON.stringify(session, null, 2)}</pre>
}
```

> **INFO**

> In contrast to `useSession`, which will return a `session` object whether or not a user has logged in (whether or not cookies are present), `getServerSession` only returns a `session` object when a user has logged in (only when authenticated cookies are present), otherwise, it returns `null`.

> **DANGER**
>
> Currently, the underlying Next.js `cookies()` method only provides read access to the request cookies. This means that the `expires` value is stripped away from `session` in Server Components. Furthermore, there is a hard expiry on sessions, after which the user will be required to sign in again. (The default expiry is 30 days).

Caching

Note that using this function implies personalized data and that you should not store pages or APIs using this in a public cache. For example a host like Vercel will implicitly prevent you from caching publicly due to the `set-cookie` header set by this function.

`unstable_getServerSession`

This method was renamed to `getServerSession`. See the documentation above.

Middleware

You can use a Next.js Middleware with NextAuth.js to protect your site.

Next.js 12 has introduced Middleware. It is a way to run logic before accessing any page, even when they are static. On platforms like Vercel, Middleware is run at the Edge.

If the following options look familiar, this is because they are a subset of these options. You can extract these to a common configuration object to reuse them. In the future, we would like to be able to run everything in Middleware. (See Caveats).

You can get the `withAuth` middleware function from `next-auth/middleware` either as a default or a named import:

Prerequisites

You must set the same secret in the middleware that you use in NextAuth. The easiest way is to set the `NEXTAUTH_SECRET` environment variable. It will be picked up by both the NextAuth config, as well as the middleware config.

Alternatively, you can provide the secret using the `secret` option in the middleware config.

We strongly recommend replacing the `secret` value completely with this `NEXTAUTH_SECRET` environment variable.

Basic usage

The most simple usage is when you want to require authentication for your entire site. You can add a `middleware.js` file with the following:

```
export { default } from "next-auth/middleware"
```

That's it! Your application is now secured. 🎉

If you only want to secure certain pages, export a `config` object with a `matcher`:

```
export { default } from "next-auth/middleware"

export const config = { matcher: ["/dashboard"] }
```

Now you will still be able to visit every page, but only `/dashboard` will require authentication.

If a user is not logged in, the default behavior is to redirect them to the sign-in page.

`callbacks`

- **Required:** No

Description

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

Example (default value)

```
callbacks: {
  authorized({ req , token }) {
    if(token) return true // If there is a token, the user is authenticated
  }
}
```

pages

- **Required**: *No*

Description

Specify URLs to be used if you want to create custom sign-in and error pages. The pages specified will override the corresponding built-in page.

> **INFO**
>
> The `pages` configuration should match the same configuration in `[...nextauth].ts`. This is so that the `next-auth` Middleware is aware of your custom pages, so it won't end up redirecting to itself when an unauthenticated condition is met.

Example (default value)

```
import { withAuth } from "next-auth/middleware"

export default withAuth({
  // Matches the pages config in `[...nextauth]`
  pages: {
    signIn: "/login",
    error: "/error",
```

```
  },
})
```

For more information, see the documentation for the pages option.

`secret`

- **Required**: *No*

Description

The same `secret` is used in the NextAuth.js config.

Example (default value)

```
secret: process.env.NEXTAUTH_SECRET
```

Advanced usage

NextAuth.js Middleware is very flexible, there are multiple ways to use it.

> **NOTE**
>
> If you do not define the options, NextAuth.js will use the default values for the omitted options.

wrap middleware

middleware.ts

```
import { withAuth } from "next-auth/middleware"

export default withAuth(
  // `withAuth` augments your `Request` with the user's token.
  function middleware(req) {
```

```
    console.log(req.nextauth.token)
  },
  {
    callbacks: {
      authorized: ({ token }) => token?.role === "admin",
    },
  },
)

export const config = { matcher: ["/admin"] }
```

The `middleware` function will only be invoked if the `authorized` callback returns `true`.

Custom JWT decode method

If you have a custom jwt decode method set in `[...nextauth].ts`, you must also pass the same `decode` method to `withAuth` in order to read the custom-signed JWT correctly. You may want to extract the encode/decode logic to a separate function for consistency.

/api/auth/[...nextauth].ts

```
import type { NextAuthOptions } from "next-auth"
import NextAuth from "next-auth"
import jwt from "jsonwebtoken"

export const authOptions: NextAuthOptions = {
  providers: [...],
  jwt: {
    async encode({ secret, token }) {
      return jwt.sign(token, secret)
    },
    async decode({ secret, token }) {
      return jwt.verify(token, secret)
    },
  },
}

export default NextAuth(authOptions)
```

And:

middleware.ts

```
import withAuth from "next-auth/middleware"
import { authOptions } from "pages/api/auth/[...nextauth]"

export default withAuth({
  jwt: { decode: authOptions.jwt?.decode },
  callbacks: {
    authorized: ({ token }) => !!token,
  },
})
```

Caveats

- Currently only supports session verification, as parts of the sign-in code need to run in a Node.js environment. In the future, we would like to make sure that NextAuth.js can fully run at the Edge
- Only supports the `"jwt"` session strategy. We need to wait until databases at the Edge become mature enough to ensure a fast experience. (If you know of an Edge-compatible database, we would like if you proposed a new Adapter)

Edit this page

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