**Notes on Setting Up Auth.js (NextAuth) for Next.js Authentication**

* **NextAuth.js** has been renamed to **Auth.js**, making it compatible with multiple libraries beyond Next.js.
* **Auth.js** simplifies authentication in **Next.js** applications but is not a standalone authentication service like **Supabase Auth**.
* Supports multiple **authentication providers** such as **Google, GitHub, Facebook**, and **custom credential providers**.
* Alternative **all-in-one** authentication services include **Clerk** and **Lucia**, which offer pre-built components but may require payment.

**Setting Up Auth.js**

1. **Create an auth.js file** in the project.
2. **Set up environment variables**:
   * NEXTAUTH\_URL → Application's running URL (e.g., http://localhost:3000).
   * NEXTAUTH\_SECRET → A secure secret key (can be generated using **Vercel's Generate Secret** tool).
3. **Configure Google Authentication**:
   * Visit **Google Developer Console**.
   * Create a **new project**.
   * Navigate to **APIs & Services → Credentials**.
   * Set up an **OAuth Consent Screen** (external users need configuration).
   * Create **OAuth Client ID** credentials.
   * Add required **redirect URIs** (<http://localhost:3000/api/auth/callback/google>).

Courtesy: [Auth.js | Google](https://authjs.dev/getting-started/providers/google?framework=next-js)

* + Copy and store **Client ID** and **Client Secret** in environment variables:
    - AUTH\_GOOGLE\_ID=<your\_client\_id>
    - AUTH\_GOOGLE\_SECRET=<your\_client\_secret>

**Installing and Using Auth.js**

* Install **Auth.js (NextAuth v5)** using:

sh

CopyEdit

npm install next-auth@beta

* Import required dependencies:

javascript

CopyEdit

import NextAuth from "next-auth";

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

* Use **Google Provider** in the configuration.

**Future Considerations**

* When deploying to production, update **NEXTAUTH\_URL** and OAuth **redirect URIs**.
* Future versions of **Auth.js** may no longer require the @beta flag.

[End of Notes]

Code:

import NextAuth from "next-auth";

import Google from "next-auth/providers/google";

const authConfig = {

  providers: [

    Google({

      clientId: process.env.AUTH\_GOOGLE\_ID,

      clientSecret:

        process.env.AUTH\_GOOGLE\_SECRET,

    }),

  ],

};

//what we export here in auth .js will be exported in the route hanlder

//so that signin and signup pages work

export const {

  auth,

  handlers: { GET, POST },

} = NextAuth(authConfig);

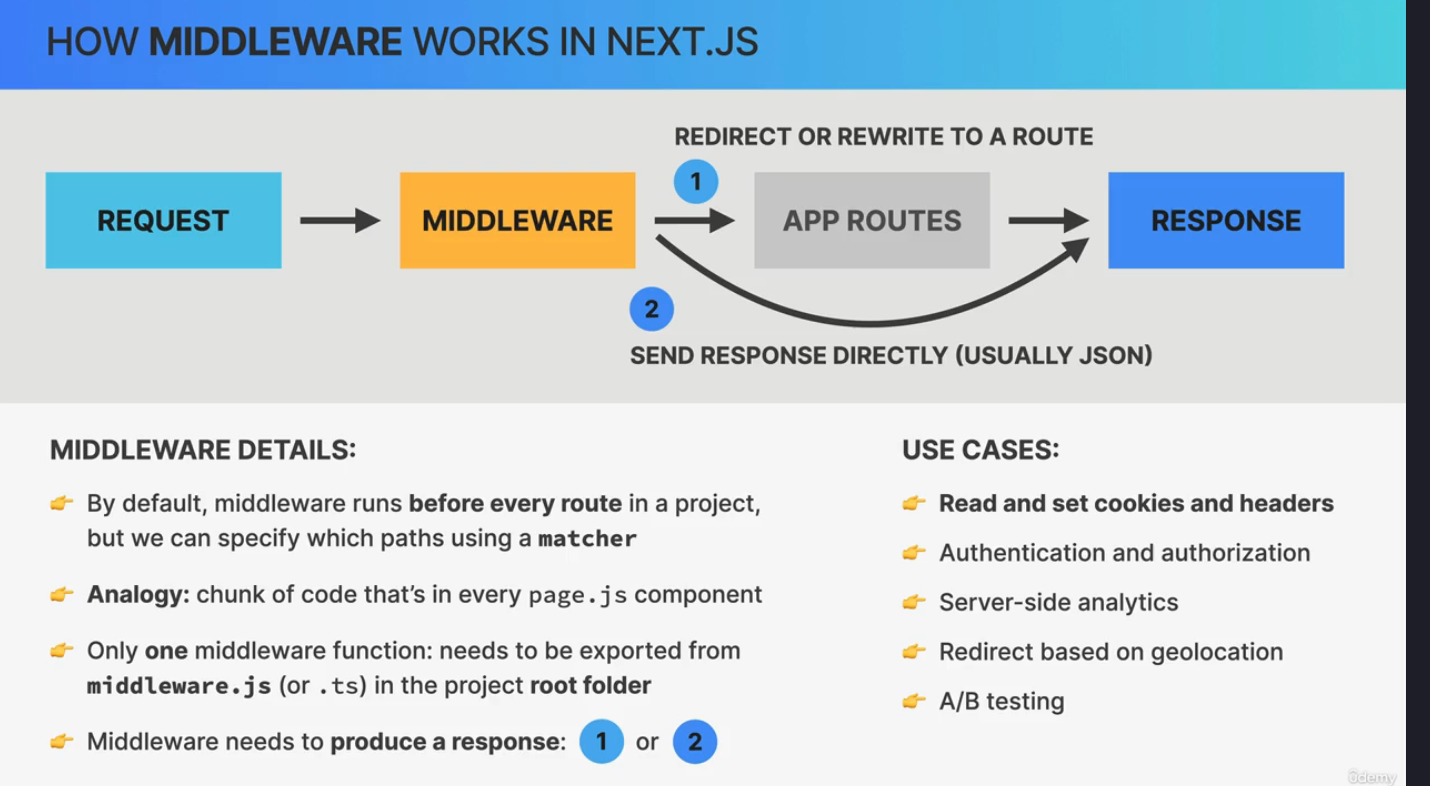
//in route hanlder we import those route handler functions and export in route.js

//@ /app/api/auth/[...nextauth]/route.js

**Notes on Authentication and Authorization in Web Development**

* **Displaying User Avatar**
  + After authentication, display the logged-in user’s avatar near the guest area.
  + Fetch user data using the auth function, which provides session details (name, email, image).
  + Use **conditional rendering** to show the avatar only if a user is logged in.
  + Implement image styling (height 8, rounded-full for circular shape).
  + Next.js recommends using the built-in Image component but can be skipped.
* **Session Handling**
  + The auth function retrieves session data and must be awaited inside an **async function**.
  + Session object contains user property with name, email, and image.
  + If the session is null, the user is logged out.
  + **Optional chaining** (session?.user) prevents errors when session data is unavailable.
* **Dynamic vs Static Routes**
  + Authentication relies on cookies and headers, making routes **dynamic**.
  + Calling auth in the navigation makes the entire website dynamic.
  + To avoid making all routes dynamic, avoid calling auth in global components like navigation.
* **Restricting Form Access**
  + Only logged-in users should see the reservation form.
  + If a user is not logged in, they see a message: **"Please log in to reserve this cabin."**
  + Conditional rendering:
    - If session.user exists → show the reservation form.
    - If session.user is null → show a **LoginMessage** component.
* **Passing Data Between Server and Client**
  + Authentication logic is handled **on the server** for security.
  + User data is passed from **server to client components**.
  + Example: The server fetches session data, then passes session.user to the reservation form.
* **Authentication vs Authorization**
  + **Authentication**: Confirms a user's identity (who they are).
  + **Authorization**: Determines what users are allowed to access.
  + Authorization ensures restricted content is only available to authenticated users.

[End of Notes



 **Next.js Middleware**: A function that runs between an incoming request and the response, allowing execution of code before rendering a route.

 **Default Behavior**: Middleware runs before every route in a Next.js app.

 **Custom Matching**: Use a *matcher* to specify which routes the middleware should apply to.

 **Centralized Logic**: Instead of adding logic to every route, middleware keeps code clean by centralizing common logic.

 **Middleware File**:

* Named middleware.js or middleware.ts (for TypeScript).
* Must be placed in the **root folder** of the project (not inside app).

 **Primary Use Cases**:

* Reading and setting **cookies** and **headers**.
* Implementing **authentication** and **authorization**.
* Performing **server-side analytics**.
* Handling **redirects** (e.g., based on **geolocation**).
* Enabling **A/B testing**.

 **Middleware Response Types**:

1. **Redirects/Rewrites**:
   * Middleware modifies the request before rendering the route.
2. **Direct JSON Response**:
   * Middleware sends a JSON response without rendering a route (useful for APIs).

 **Key Exception**: When middleware sends a JSON response, the route is completely bypassed.

**Protecting Routes in Next.js with NextAuth Middleware**

This guide will walk through implementing **middleware** in Next.js to **protect guest area routes** using **NextAuth.js**. We'll start by manually creating middleware and then integrate NextAuth's built-in middleware for authentication.

**Step 1: Creating a Middleware File**

Middleware in Next.js is conventionally placed in the **root folder** (not inside the app folder).

**Create a middleware.js file in the root directory:**

javascript

CopyEdit

// middleware.js

export function middleware(request) {

console.log(request); // Logs the incoming request details

return Response.next(); // Allows the request to continue

}

🔹 **Explanation:**

* The middleware function receives the request object.
* We log the request object to inspect what data it contains (cookies, headers, geolocation, etc.).
* Response.next() ensures that the request continues as usual.

**Step 2: Redirecting Users in Middleware**

Now, let's **redirect** users to a different page.

**Update middleware.js:**

javascript

CopyEdit

import { NextResponse } from "next/server";

export function middleware(request) {

return NextResponse.redirect(new URL("/about", request.url));

}

🔹 **Explanation:**

* We import NextResponse to manage responses.
* NextResponse.redirect() redirects **all requests** to the /about page.

⚠ **Issue:** This middleware runs on **every route**, causing an **infinite redirect loop** (e.g., /account → /about → /about again).

**Step 3: Restricting Middleware to Certain Routes**

To apply middleware **only to specific routes**, we use a **matcher**.

**Update middleware.js with a matcher:**

javascript

CopyEdit

export const config = {

matcher: ["/account", "/cabins"],

};

🔹 **Explanation:**

* The matcher ensures middleware **only runs on** /account and /cabins.
* Now, only these routes redirect to /about, avoiding the infinite loop.

**Step 4: Protecting Routes Using NextAuth Middleware**

Instead of manually handling redirects, we can use **NextAuth middleware** for authentication.

**Install NextAuth.js (if not installed)**

sh

CopyEdit

npm install next-auth

**Update middleware.js to use NextAuth:**

javascript

CopyEdit

import { auth } from "./app/lib/auth"; // Import auth helper

export const middleware = auth; // Use auth as middleware

export const config = {

matcher: ["/account"], // Protect only the /account route

};

🔹 **Explanation:**

* **auth** is a function from NextAuth that handles authentication.
* **We simply export it as middleware**, making authentication **automatic**.
* **Only /account is protected**, so unauthorized users are redirected.

**Step 5: Configuring NextAuth with Callbacks**

Instead of manually checking the session in middleware.js, we define **authentication rules inside authConfig.js**.

**Modify authConfig.js**

javascript

Copy code

import Google from "next-auth/providers/google";

export const authConfig = {

providers: [

Google({

clientId: process.env.AUTH\_GOOGLE\_ID,

clientSecret: process.env.AUTH\_GOOGLE\_SECRET,

}),

],

callbacks: {

authorized({ auth, request }) {

return !!auth?.user; // Only allow access if user is logged in

},

},

};

🔹 **Explanation:**

* **providers** → Configures Google authentication.
* **callbacks.authorized()** → Automatically runs for protected routes:
  + If the user is authenticated (auth?.user exists) → **Allow access** ✅
  + If the user is **not logged in** → **Deny access** ❌

**Step 6: Testing the Middleware**

1. **Try accessing /account without logging in** → You should be redirected to the **sign-in page**.
2. **Log in** → You should now access /account successfully.
3. **Sign out and try again** → You will be redirected to **sign-in** again.

To implement **signIn using a form and server actions** in your Next.js app with **NextAuth**, follow these steps carefully.

**1. Ensure signIn is Exported from auth.js**

Since your auth.js already exports the required functions, you don't need to change it.

✅ **Ensure this structure is correct** in app/auth.js:

js

CopyEdit

import NextAuth from "next-auth";

import Google from "next-auth/providers/google";

const authConfig = {

providers: [

Google({

clientId: process.env.AUTH\_GOOGLE\_ID,

clientSecret: process.env.AUTH\_GOOGLE\_SECRET,

}),

],

callbacks: {

authorized({ auth }) {

return !!auth?.user; // Allow access if the user is logged in

},

},

pages: {

signIn: "/login", // Custom login page

},

};

export const {

auth,

handlers: { GET, POST },

signIn,

signOut,

} = NextAuth(authConfig);

**2. Create handleSignIn Inside lib/actions.js**

✅ **Create lib/actions.js**

js

CopyEdit

// app/lib/actions.js

"use server";

import { signIn } from "@/app/auth";

* export async function signInAction() {
* await signIn("google", {
* redirectTo: "/account",
* });
* // •  Redirects to /account after successful login.
* }
* **Location:** app/lib/actions.js
* **Runs on the Server**: "use server" ensures this function is executed server-side.
* **Calls signIn** with **Google** authentication.
* **Redirects to /account** after successful login.

**3. Create a Sign-In Form Component**

✅ **Create SignInForm.js**

jsx

CopyEdit

// app/components/SignInForm.js

import { signInAction } from "@/app/lib/actions";

export default function SignInForm() {

return (

<form action={signInAction}>

<button type="submit">Sign In with Google</button>

</form>

);

}

* Uses the **server action** inside lib/actions.js.
* Clicking the button **submits the form** and triggers handleSignIn().

**4. Update the /login Page**

✅ **Modify app/login/page.js**

jsx

CopyEdit

// app/login/page.js

import SignInForm from "@/app/components/SignInForm";

export const metadata = {

title: "Login",

};

export default function LoginPage() {

return (

<div>

<h2>Login</h2>

<SignInForm />

</div>

);

}

* **Renders the SignInForm component.**
* **Handles metadata for SEO.**

**5. Display User Info After Login**

✅ **Create app/account/page.js**

jsx

CopyEdit

// app/account/page.js

import { auth } from "@/app/auth";

export default async function AccountPage() {

const session = await auth();

const firstName = session?.user?.name?.split(" ")[0] ?? "User";

return <h2>Welcome, {firstName}!</h2>;

}

* **Fetches user session data.**
* **Displays first name** (or "User" if undefined).

**Notes on Implementing a Custom Sign-Out Button**

* **Create a server action** for signing out, similar to the previous sign-in action.
* Define an **async function** called SignOutAction().
  + Calls the previously exported **signOut function**.
  + Passes an **object with the redirect property** set to the root (/).
  + Ensures the user is redirected to the homepage after signing out.
* **Integrating the Sign-Out Button**:
  + The button is part of the **side navigation**, which is a **client component**.
  + Since client components **cannot directly execute server actions**, we wrap it in a **form**.
  + The **form’s action prop** is set to SignOutAction.
* **Server Actions in Client Components**:
  + **Server actions can be called from client components** but still execute **only on the server**.
  + This allows secure processing while keeping UI interactions smooth.
* **Testing the Sign-Out Functionality**:
  + Clicking the button removes the **user’s session**.
  + Navigating to protected pages **prompts the user to log in again**.
  + Example: Trying to reserve a cabin now asks for authentication.
* **Next Steps**:
  + Connect authenticated users to **guests in the Supabase database**.

[End of Notes]

**Explanation of Callbacks in Your Code with Respect to the Documentation**

Your NextAuth configuration defines several **callbacks**, which are asynchronous functions used to control authentication behavior. Below is how each callback works in your auth.js file and how it aligns with the documentation.

**1. authorized Callback**

**Your Code:**

js

CopyEdit

authorized({ auth, request }) {

return !!auth?.user;

}

**What It Does:**

* This callback **runs automatically** for protected routes.
* It checks whether the user is authenticated (auth?.user exists).
* If auth?.user is **truthy**, access is allowed (✅).
* If auth?.user is **falsy**, access is denied (❌), and the user is redirected to the sign-in page.

**Relation to Docs:**

* Not explicitly mentioned in the documentation but acts as a middleware-like function that enforces authentication for certain routes.

**2. signIn Callback**

**Your Code:**

js

CopyEdit

async signIn({ user, account, profile }) {

try {

const existingGuest = await getGuest(user.email);

if (!existingGuest) {

await createGuest({

email: user.email,

fullName: user.name,

});

return true;

}

} catch {

return false;

}

}

**What It Does:**

* Runs when a user attempts to sign in.
* Calls getGuest(user.email) to check if the user already exists in your system.
* If the user does **not** exist, it creates a new guest entry.
* Returns true to allow sign-in or false to reject it.

**Relation to Docs:**

* Matches the signIn callback in the docs:

js

CopyEdit

async signIn({ user, account, profile, email, credentials }) {

return true;

}

* Instead of just allowing sign-in (return true), your code **adds logic** to check if the user exists in your database.

**3. session Callback**

**Your Code:**

js

CopyEdit

async session({ session, user }) {

const guest = await getGuest(session.user.email);

session.user.guestId = guest.id;

return session;

}

**What It Does:**

* Runs whenever the session is accessed (e.g., useSession(), getSession(), /api/auth/session).
* Looks up the guestId associated with the user's email.
* Attaches guestId to session.user, making it available in the frontend.

**Relation to Docs:**

* The docs show how to modify the session:

js

CopyEdit

async session({ session, token, user }) {

session.accessToken = token.accessToken;

session.user.id = token.id;

return session;

}

* Your implementation does something similar but **fetches the guestId from your database** instead of using a JWT token.

**Step-by-Step Flow**

1. **User Clicks "Sign in with Google"**
   * NextAuth redirects them to Google's authentication page.
2. **User Enters Email & Password on Google's Page**
   * If the password is **incorrect**, Google blocks the login and does **not** redirect back to your app.
   * If the password is **correct**, Google **authenticates** the user and redirects them back to your app.
3. **NextAuth Receives User Data**
   * After successful authentication, Google sends user details (email, name, etc.) back to NextAuth.
4. **signIn Callback Runs**
   * The signIn function in your NextAuth config is triggered.
   * It checks if the user's email exists in your database (getGuest(user.email)).
   * If the user does not exist, it creates a new guest (createGuest(...)).
   * If any error occurs (e.g., database issues), it returns false, preventing login.
5. **User is Logged In**
   * If everything is successful, NextAuth completes the login process and creates a session.

**Step-by-Step Timeline**

1. **User Signs In Successfully**
   * The user logs in using Google.
   * The signIn callback (if defined) runs first.
   * If everything is successful, NextAuth **creates a session**.
2. **session Callback is Triggered Immediately**
   * Right after session creation, NextAuth **calls the session callback**.
   * This happens **before sending the session data to the frontend**.
   * The session object is modified inside this callback.
3. **Modified Session is Stored and Sent to the Client**
   * The updated session (with extra fields like guestId) is saved and sent to the frontend.
   * The frontend can access this session using:

js

CopyEdit

import { useSession } from "next-auth/react";

const { data: session } = useSession();

* + The session object now contains:

json

CopyEdit

{

"user": {

"name": "John Doe",

"email": "john@example.com",

"image": "profile-pic-url",

"guestId": "12345"

},

"expires": "2024-12-31T23:59:59.999Z"

}

**Other Times When the session Callback Runs**

Apart from the initial sign-in, the session callback is also triggered when:

1. **A Page Loads and Calls useSession()**
   * If a page uses useSession(), NextAuth fetches the session and calls the callback.
2. **API Calls Request the Session (/api/auth/session)**
   * If an API route or a client-side request fetches the session, the session callback runs again.
3. **Session is Refreshed or Revalidated**
   * If the user’s session is refreshed (e.g., re-fetching data, token renewal), the callback is executed again.

**Key Takeaways**

✔️ The session callback **runs immediately after the session is created**.  
✔️ It ensures that the session contains extra data before reaching the frontend.  
✔️ It **also runs whenever the session is accessed or refreshed**.

So, yes, **right after session creation, the session callback is executed!**