

# Next.js에서의 인증과 보안

## 1. Next-Auth/Auth.js 기본설정

= > 다양한 프레임워크에서 인증을 간편하게 구현할 수 있는 범용 인증 라이브러리

### 1. 패키지 설치

```
npm install next-auth
```

### 2. 환경 변수 설정

```
npx auth secret
```

### 3. API 라우트 파일 만들기

- App Router를 사용한다면 (app 디렉토리): app/api/auth/[...nextauth]/route.ts (또는 .js) 파일 생성
- Pages Router를 사용한다면 (pages 디렉토리): pages/api/auth/[...nextauth].ts (또는 .js) 파일 생성

### 4. Auth.js 기본 설정 작성

- App Router 방식

```
import NextAuth from "next-auth";
// 사용할 provider를 import 설정

const handler = NextAuth({
  providers: [
    // Provider 설정들을 추가
  ],
  // 필요한 다른 설정들 추가
});

export { handler as GET, handler as POST };
```

- Pages Router 방식

```
import NextAuth from "next-auth";
// 사용할 provider를 import 설정

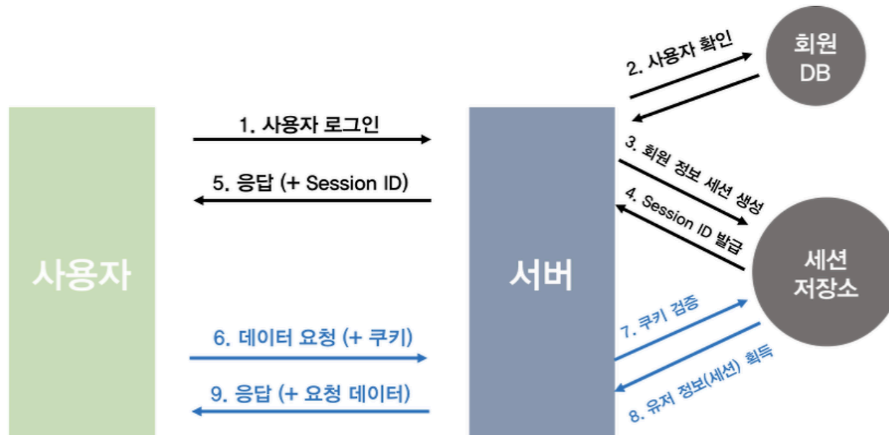
const handler = NextAuth({
  providers: [
    // Provider 설정들을 추가
  ],
  // 필요한 다른 설정들 추가
});

export default handler;
```

## 2. Session 관리

= > 세션 관리는 사용자의 로그인 상태를 유지하기 위한 방식

### 1. 동작 순서



### 2. 세션 관리 방식

- 상태 비저장형 (Stateful) : JWT - 세션 데이터(또는 토큰)를 브라우저 쿠키에 저장

- 1) 세션에 서명하는 데 사용될 비밀 키를 생성하고 이를 환경 변수로 저장
- 2) 세션 관리 라이브러리를 사용하여 세션 데이터를 암호화/복호화하는 로직을 작성
- 3) Next.js cookiesAPI를 사용하여 쿠키를 관리

```
openssl rand -base64 32
```

```
SESSION_SECRET=your_secret_key
```

```
import 'server-only'
import { SignJWT, jwtVerify } from 'jose'
import { SessionPayload } from '@app/lib/definitions'

const secretKey = process.env.SESSION_SECRET
const encodedKey = new TextEncoder().encode(secretKey)

export async function encrypt(payload: SessionPayload) {
  return new SignJWT(payload)
    .setProtectedHeader({ alg: 'HS256' })
    .setIssuedAt()
    .setExpirationTime('7d')
    .sign(encodedKey)
}

export async function decrypt(session: string | undefined = '') {
  try {
    const { payload } = await jwtVerify(session, encodedKey, {
      algorithms: ['HS256'],
    })
    return payload
  } catch (error) {
    console.log('Failed to verify session')
  }
}
```

- 상태 저장형 (Stateless) : DB - 세션 데이터를 데이터베이스에 저장
- 1) 세션과 데이터를 저장할 데이터베이스에 테이블 작성(또는 인증 라이브러리로 확인)
- 2) 세션을 삽입, 업데이트, 삭제하는 기능을 구현
- 3) 사용자 브라우저에 저장하기 전에 세션 ID를 암호화하고, 데이터베이스와 쿠키가 동기화된 상태를 유지

```

import cookies from 'next/headers'
import { db } from '@app/lib/db'
import { encrypt } from '@app/lib/session'

export async function createSession(id: number) {
  const expiresAt = new Date(Date.now() + 7 * 24 * 60 * 60 * 1000)

  // 1. Create a session in the database
  const data = await db
    .insert(sessions)
    .values({
      userId: id,
      expiresAt,
    })
  // Return the session ID
  .returning({ id: sessions.id })

  const sessionId = data[0].id

  // 2. Encrypt the session ID
  const session = await encrypt({ sessionId, expiresAt })

  // 3. Store the session in cookies for optimistic auth checks
  const cookieStore = await cookies()
  cookieStore.set('session', session, {
    httpOnly: true,
    secure: true,
    expires: expiresAt,
    sameSite: 'lax',
    path: '/',
  })
}

```

### 3. 세션 관리 방식에 따른 권장 환경 비교

| 항목      | JWT                           | DB                                |
|---------|-------------------------------|-----------------------------------|
| 상태 관리   | 무상태(Stateless)                | 상태 존재(Stateful)                   |
| 보안성     | 토큰 탈취시 위험                     | 세션 서버 만료 가능                       |
| 속도      | 빠름(DB 조회X)                    | 느림(매 요청마다 DB 조회)                  |
| 사용 방식   | 클라이언트 쿠키에 JWT 저장<br>및 서버에서 검증 | 세션 ID를 쿠키에 저장,<br>서버DB에서 인증 정보 조회 |
| 갱신/로그아웃 | JWT 만료 설정 및 수동 무효화 필요         | 세션을 DB에서 삭제                       |
| 연동 방식   | jwt 콜백에서 커스터마이징               | adapter 사용, 세션 자동<br>저장           |

### 3. Middleware를 활용한 보호된 라우트

=> 요청이 완료되기 전에 코드를 실행하여 수신되는 요청에 따라 요청 혹은 헤더를 재작성 후, 리디렉션, 수정을 하는 과정

#### - Middleware 파일 설정후 경로 지정

```
// middleware.ts
import { NextResponse } from 'next/server';
import type { NextRequest } from 'next/server';

export function middleware(request: NextRequest) {
  const token = request.cookies.get('next-auth.session-token');
  const isAuthPage = request.nextUrl.pathname.startsWith('/auth');

  if (!token && !isAuthPage) {
    // 인증되지 않은 사용자는 로그인 페이지로 리디렉션
    return NextResponse.redirect(new URL('/auth/signin', request.url));
  }

  if (token && isAuthPage) {
    // 인증된 사용자가 인증 페이지에 접근하면 대시보드로 리디렉션
    return NextResponse.redirect(new URL('/dashboard', request.url));
  }

  return NextResponse.next();
}

export const config = {
  matcher: ['/dashboard/:path*', '/profile/:path*'], // 보호할 경로 지정
};
```

### 4. OAuth 및 소셜 로그인

=> OAuth는 사용자가 비밀번호를 제공하지 않고도 제3자 애플리케이션이 사용자의 자원에 접근할 수 있도록 허용하는 인증 및 권한 부여 프로토콜

- 1) 패키지 설정
- 2) 환경 변수 설정
- 3) API 라우트 파일 만들기
- 4) 소셜 로그인 Provider 설치
- 5) 설정 파일에 Provider 추가
- 6) 리다이렉트 URI 설정

```
npm install next-auth
```

```
npm install @next-auth/google
```

```
// - app/api/auth/[...nextauth]/route.ts (App Router)
// - pages/api/auth/[...nextauth].ts (Pages Router)
// Router 방식에 따라 파일 만들기

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

const authOptions = {
  providers: [
    GoogleProvider({
      clientId: process.env.GOOGLE_CLIENT_ID!,
      clientSecret: process.env.GOOGLE_CLIENT_SECRET!,
    }),
  ],
  // 소셜 주소 연결

  callbacks: {
    async session({ session, token, user }) {
      session.user.id = token.sub; // 사용자 ID 전달
      return session;
    },
  },
  // 콜백

  session: {
    strategy: "jwt",
  },

  secret: process.env.NEXTAUTH_SECRET,
};
// DB 방식 사용시 : database

const handler = NextAuth(authOptions);
export { handler as GET, handler as POST };
// Pages Router 방식 사용시 export default NextAuth(authOptions);
```