How to add JWT Authentication to NextJS Apps

Prepared by Muhammad Shafiq

create a project in nextJS

npx create-next-app@latests

install dependencies
npm install jsonwebtoken
npm i @types/jsonwebtoken --save-dev
npm i cookie
npm i @types/cookie -D (this for typescript)

creating a new Next.js project and installing specific dependencies.

- `npx create-next-app@latest`: This command creates a new Next.js application with the latest version. `npx` is a package runner tool that comes with `npm`. When you run `npx create-next-app@latest`, it downloads (if not already downloaded) and executes the `create-next-app` command. `create-next-app` is a tool provided by the Next.js team to bootstrap a new Next.js application.
- `npm install jsonwebtoken`: This command is used to install the `jsonwebtoken` package from the npm registry. `jsonwebtoken` is a package that helps in creating, decoding, verifying, and signing JSON Web Tokens (JWT). JSON Web Tokens are used in

authentication systems to represent claims to be transferred between two parties.

`npm i @types/jsonwebtoken --save-dev`: This command is used to install TypeScript definitions for the `jsonwebtoken` package. This is necessary because TypeScript is a statically typed superset of JavaScript, and it requires type definitions for JavaScript libraries. The `--save-dev` flag is used to save the package for development purposes. That is, this package is not used in the production build of your app, but is only used during development.

`npm i cookie`: This command is used to install the `cookie` package. The `cookie` library allows you to parse and serialize HTTP cookies, which are used for maintaining user state or other relevant information across multiple requests.

`npm i @types/cookie -D`: Just like the previous `@types` installation, this command installs TypeScript definitions for the `cookie` library. Again, the `-D` flag is equivalent to `--save-dev`, indicating that this package is needed for development, not for the final, production build of your app.

Create page.tsx in app directory

This is a basic example of a login form in a React.js component using the Next.js framework. The code performs the following tasks:

Importing Required Libraries: The necessary libraries and components are imported. This includes the 'Image' component from 'next/image' which isn't used in the code. Then the 'React' library is imported.

Defining the Component: The `Home` function is declared which is a functional component in React.

Form Submission Handler: Inside the `Home` component, a function `handleSubmit` is defined which is an asynchronous function. This function prevents the default form submission behavior (which would

cause a page reload) and creates a payload object that contains the username and password entered by the user. However, the payload is not used in this code which implies that there should be an API request or some form of submission which isn't shown in the code you provided.

Return JSX: JSX (JavaScript XML) is returned by the component which describes what the UI should look like.

- A `form` is created with a class to apply some CSS styling and an `onSubmit` event that triggers the `handleSubmit` function when the form is submitted.
- Inside the form, two `div` elements are created that each contain a `label` and an `input`. The first one is for the username and the second one is for the password. Both inputs have their type set to 'text', id, name, and CSS classes for styling, and they're both required fields.
- Another `div` contains a `button` of type 'submit' which triggers the form submission when clicked.

Please note, the password input field's type should be 'password' instead of 'text' for security reasons. This masks the input, providing a basic level of security and privacy for the user.

```
'use client'
import Image from 'next/image'
import React from 'react';

export default function Home() {
  const handleSubmit = async (event: React.FormEvent<HTMLElement>) => {
    event?.preventDefault();

    const payload = {
      username : event.currentTarget.username.value,
      password : event.currentTarget.password.value,
```

```
    };

    // POST request using fetch()
    fetch("/api/auth",
      {
        method: "POST",
        body: JSON.stringify(payload),
        headers: {
          "Content-type": "application/json; charset=UTF-8"
        }

      })
      .then(response => response.json())
      .then(json => console.log(json))
      .catch(error => console.error(error));

    };

  return (
    <>
    <main>
      <h1>Nextjs authentication JWT verify http cookie only</h1>

       <form onSubmit={handleSubmit} className='flex flex-col gap-4 w-full
md:w-full'>
        <div className='flex justify-evenly'>
          <label htmlFor='username'>Username</label>
          <input
          type='text'
          id='username'
          name='username' required
          className='border rounded border-green-900'
          />
        </div>

        <div className='flex justify-evenly'>
          <label htmlFor='password'>password</label>
          <input
          type='text'
          id='password'
```

```
          name='password' required
          className='border rounded border-green-900'
          />
        </div>
        <div className='flex justify-evenly'>
          <button
          type='submit'
              className="button border rounded border-green-900 py-2 px-7
bg-green-900 text-white ml-8">Submit</button>
        </div>


      </form>
    </main>
    </>

    )
}
```

create api route in app/api/auth/route.ts
This code exports an asynchronous function named `POST` that serves as a handler for POST requests. This function can be used to handle user login requests in a Node.js server, particularly when using Next.js. Here's a step-by-step explanation:

> The function begins by parsing the JSON request body with `await request.json()`. It expects the request body to contain a JSON object with `username` and `password` properties.
> The provided `username` and `password` are then checked against predefined values ('admin' for both). If they don't match, a response

with a 401 status code (indicating unauthorized access) and a message "Unauthorized" is returned.
If the credentials are correct, a JSON Web Token (JWT) is generated using the `sign` function from the `jsonwebtoken` library. The payload of the JWT contains the `username` and `id`, and it is signed with a secret key that is retrieved from the environment variables (`process.env.JWT_SECRET`). If the environment variable is not set, it defaults to the string "as00". The token will expire in 24 hours (`24*60*60` seconds).
This token is then set as a cookie. The `serialize` function from the `cookie` library is used to prepare the cookie string. The cookie's name is "OutSiteJWT", and it is HTTP-only and has the same lifespan as the JWT (24 hours). It is also secure if the environment is production (`process.env.NODE_ENV === "production"`), meaning it will only be sent over HTTPS connections. The cookie has a `sameSite` policy of "strict", meaning it will only be sent in requests originating from the same site.
The server responds with a 200 status code (indicating successful request) and a message "Authenticated!". The response also includes the Set-Cookie header with the serialized JWT, which ensures that the cookie will be stored on the client's browser.

Remember that this is a basic example and doesn't involve any real-world user authentication. In a real-world scenario, you'd likely have to use hashed passwords and communicate with a database to verify user credentials.

```
import { sign } from "jsonwebtoken"
import { NextResponse } from "next/server"
import { serialize } from "cookie";

export async function POST(request: Request){
    const body = await request.json();
```

```
    const { username, password} = body;

    if (username!= 'admin' || password!= 'admin') {
        return NextResponse.json(
            {
                message: "Unauthorized",
            },
            {
                status: 401,
            }
        )
    }
    const secret = process.env.JWT_SECRET || "as00";

    const token = sign(
        {
            username: username,
            id: 1,

        },
        secret,
        {
            expiresIn: 24*60*60
        }
    )
    const serialized = serialize("OutSiteJWT", token, {
        httpOnly: true,
        secure: process.env.NODE_ENV === "production",
        sameSite: "strict",
        maxAge: 24*60*60*1000,
        path: "/",
    });

    const response = {
        message: "Authenticated!"
    }
    return new Response(JSON.stringify(response), {
        status: 200,
        headers: {"Set-Cookie": serialized}
```

```
    });
}
```

Create “middleware.ts” in src or root of your project

This script appears to be a middleware function for a Next.js server-side application that performs some kind of authentication based on the presence of a specific cookie, namely 'OutSiteJWT'.

Here is an explanation of the key parts:

> import { NextResponse } from 'next/server'
> import type { NextRequest } from 'next/server'
> These lines are importing the necessary types and classes from the 'next/server' package. `NextResponse` and `NextRequest` are used to type the request and response objects.
> export function middleware(request: NextRequest)
> This line declares a middleware function that will be executed for every incoming HTTP request. It takes a single parameter `request`, which is an object of type `NextRequest`.
> if(request.cookies.has('OutSiteJWT'))
> This line checks if the request has a cookie named 'OutSiteJWT'. If it does, it does nothing in its current form (the intended operations are commented out), but the original intent seems to be to retrieve the

value of the 'OutSiteJWT' cookie and add it to the `request` object as a `token` property.
return new NextResponse(JSON.stringify({ success: false, message: 'Unauthorized' }), { status: 401, headers: { 'content-type': 'application/json' } })
If the request does not have the 'OutSiteJWT' cookie, the middleware sends a JSON response with a 401 status code (Unauthorized) and a message indicating that authorization has failed.
*export const config = { matcher: '/admin/:path' }**
The `config` object exported from this module contains configuration settings for the middleware. The `matcher` property specifies a URL pattern that the middleware will apply to. In this case, it will match any URL that starts with '/admin/'.

Please note that the code inside the `if` statement is currently commented out, so it doesn't do anything. You would need to uncomment this code for it to actually read the token from the cookie and attach it to the request.

```
import { NextResponse } from 'next/server'
import type { NextRequest } from 'next/server'

export default function middleware(request: NextRequest) {



    if(request.cookies.has('OutSiteJWT')){
    //      let cookie = request.cookies.get('OutSiteJWT')
    //      // console.log(cookie) // => { name: 'OutSiteJWT', value: 'ebmneheheiuiy678eiu', Path: '/' }
    //      const {value}:any = cookie;
    //       console.log(value)
    //      //   request.cookies.delete('nextjs')
```

```
    //     request.token = value;

    }else{
       return new NextResponse(
        JSON.stringify({ success: false, message: 'Unauthorized' }),
        { status: 401, headers: { 'content-type': 'application/json' } }
      )


    }
}


export const config = {
    matcher: '/admin/:path*',
}
```

Create new file and enter app/admin/me/page.tsx

This is a script for a page in a Next.js application that uses JWT (JSON Web Tokens) for authentication. This page is trying to verify if the user is logged in or not, based on the JWT stored in the cookies of the browser. If the user is logged in (i.e., if the JWT can be verified), it displays some user-specific information. Otherwise, it provides a link to the login page. Here's a step-by-step explanation:

It imports necessary libraries and modules: 'jsonwebtoken' to verify JWT, 'next/server' to handle requests in Next.js, 'next/headers' to manipulate cookies, and 'next/link' to handle internal routing in Next.js.
It defines a functional React component, `Page`, which takes a `NextRequest` object as input. `NextRequest` is an object representing an incoming HTTP request in Next.js.
In the component, it first declares a variable `data` to store the decoded JWT data.
It tries to verify the JWT stored in the 'OutSiteJWT' cookie.
It gets the JWT secret from environment variables or uses a default value 'as00' if it doesn't exist.
It initializes the `cookieStore` using `cookies()`, gets the 'OutSiteJWT' cookie from the `cookieStore`, and then gets the JWT from the cookie.
It verifies the JWT using the secret. If the JWT is valid, `verify()` returns the payload of the JWT, which is an object containing user information, and assigns it to `data`.
If an error occurs during this process, it catches the error and logs it to the console.

In the return statement of the component, it checks if `data` is truthy (i.e., if the JWT was verified successfully). If it is, it displays some user-specific information from `data` (the username and the user ID). Otherwise, it provides a link to the login page.
Finally, it exports the `Page` component as a default export, so it can be imported and used elsewhere in the application.

```
import { verify } from 'jsonwebtoken';
import { NextRequest } from 'next/server';
import { cookies } from 'next/headers'
import React from 'react'
import Link from 'next/link';


function Page(request:NextRequest) {
  let data:any = undefined;
try {
  const secret = process.env.JWT_SECRET || "as00";
  const cookieStore = cookies()
  const cookie = cookieStore.get('OutSiteJWT')
  const token:any = cookie?.value;


  data = verify(token, secret);


  }catch(err){
    console.log("ERROR: ",err)
  }
  return (
    <>{data && <div>
        <div>Me</div>
        <div>User: {data?.username}</div>
        <div>User ID: {data?.id}</div>
      </div>
    }
    {
```

```
        !data &&
        <Link href="/">Login</Link>
      }
     </>
   )
}


export default Page
```

To Be Continued