# Next JS - 4

## Authentication

User authentication is a key feature for almost every application. There are several solutions to implement authentication service. In this class, I will introduce you using "JWT" authentication and authorization. The key concepts are:

- The Next backend provides login API.
- The React frontend submits login data to the login API.
- If the user can log in successfully, a token will be generated and stored in Cookie in Http-Only mode. In this mode, the front end cannot access the cookie data directly, but the browser will send along with the API request automatically.
- The React store user authentication states in User or Authentication context.
- The default login state is false.
- Once user successfully logged in, change it to true.
- Then, when React gets 401 responses, change the login state back to false.

## Next JS Implementation:

1. Install necessary dependencies

```
npm install jsonwebtoken cookie
```

2. Create a directory "login" in "src/app/api/user".
3. Create a file named "route.js" in "src/app/api/user/login" with the code:

```
import corsHeaders from "@/lib/cors";
import { getClientPromise } from "@/lib/mongodb";
import bcrypt from "bcrypt";
import jwt from "jsonwebtoken";
import { NextResponse } from "next/server";

const JWT_SECRET = process.env.JWT_SECRET || "mydefaultjwtsecret"; // Use a
strong secret in production

export async function OPTIONS(req) {
  return new Response(null, {
    status: 200,
    headers: corsHeaders,
  });
}
export async function POST(req) {
  const data = await req.json();
  const { email, password } = data;

  if (!email || !password) {
    return NextResponse.json({
      message: "Missing email or password"
    }, {
      status: 400,
      headers: corsHeaders
    });
  }
  try {
    const client = await getClientPromise();

    const db = client.db("wad-01");
    const user = await db.collection("user").findOne({ email });
    if (!user) {
      return NextResponse.json({
        message: "Invalid email or password"
      }, {
        status: 401,
        headers: corsHeaders
      });
    }
    const passwordMatch = await bcrypt.compare(password, user.password);
    if (!passwordMatch) {
      return NextResponse.json({
        message: "Invalid email or password"
      }, {
        status: 401,
        headers: corsHeaders
      });
    }
    // Generate JWT
    const token = jwt.sign({
      id: user._id,
      email: user.email,
      username: user.username
    }, JWT_SECRET, { expiresIn: "7d" });

    // Set JWT as HTTP-only cookie
    const response = NextResponse.json({
      message: "Login successful"
    }, {
      status: 200,
      headers: corsHeaders
    });
    response.cookies.set("token", token, {
      httpOnly: true,
      sameSite: "lax",
      path: "/",
      maxAge: 60 * 60 * 24 * 7, // 7 days
      secure: process.env.NODE_ENV === "production"
    });
    return response;
  } catch (exception) {
    console.log("exception", exception.toString());
    return NextResponse.json({
      message: "Internal server error"
    }, {
      status: 500,
      headers: corsHeaders
    });
  }
}
```

Cookie settings are the key to this solution.

- Set httpOnly to true is to trigger HTTP-Only mode, for security reasons.
- Set sameSite to "lax" to allow the domain set in Access-Allow-Origin to send Cookie with API request but not for other domains.
- Set path to "/" means the cookie shall send to the domain for every path request.
- The maxAge is the cookie alive period.
- Set secure to true to allow only https and false to allow both http and https. This should be set to be true for production.

The OPTIONS method is to handle browser pre-flight CORS checkup.

4. Create a file named "auth.js" in "lib" with the following code:

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

const JWT_SECRET = process.env.JWT_SECRET || "mydefaulyjwtsecret"; // Use a
strong secret in production

export function verifyJWT(req) {
  try {
    const cookies = req.headers.get("cookie") || "";
    const { token } = cookie.parse(cookies);
    if (!token) {
      return null;
    }
    const decoded = jwt.verify(token, JWT_SECRET);
    return decoded;
  } catch (err) {
    return null;
  }
}

// Example usage in an API route:
// import { verifyJWT } from "@/lib/auth";
// const user = verifyJWT(req);
// if (!user) return NextResponse.json({ message: "Unauthorized" }, { status: 401
});
```

5. Create a directory "logout" in "src/app/api/user".

6. Create a file named route.js in "src/app/api/user/logout"

```
import corsHeaders from "@/lib/cors";
import { NextResponse } from "next/server";

export async function OPTIONS(req) {
  return new Response(null, {
    status: 200,
    headers: corsHeaders,
  });
}

export async function POST() {
  // Clear the JWT cookie by setting it to empty and expired
  const response = NextResponse.json({
    message: "Logout successful"
  }, {
    status: 200,
    headers: corsHeaders
  });
  response.cookies.set("token", "", {
    httpOnly: true,
    sameSite: "lax",
    path: "/",
    maxAge: 0,
    secure: process.env.NODE_ENV === "production"
  });
  return response;
}
```

7. Modify the cors.js:

```
let corsHeaders = {
  "Access-Control-Allow-Credentials":"true",
  // "Access-Control-Allow-Origin": "*",
  "Access-Control-Allow-Origin": "http://localhost:5173",
  "Access-Control-Allow-Methods": "GET, POST, OPTIONS",
  "Access-Control-Allow-Headers": "Content-Type, Authorization",
  "Access-Control-Max-Age": "86400",
};

export default corsHeaders;
```

To utilize the HTTP-Only mode, the Access-Control-Allow-Origin must be fixed to a value. If there are multiple domains to be supported, a dynamic CORS header must be implemented. For example, change the corsHeaders from variable to function and determine Origin dynamically.

8. Now, let's create a API that required authentication for testing, create a directory named profile in "src/app/api/user".
9. Create a file named route.js in "src/app/api/user/profile".

```
import { verifyJWT } from "@/lib/auth";
import corsHeaders from "@/lib/cors";
import { getClientPromise } from "@/lib/mongodb";
import { NextResponse } from "next/server";

export async function OPTIONS(req) {
  return new Response(null, {
    status: 200,
    headers: corsHeaders,
  });
}

export async function GET (req) {
  const user = verifyJWT(req);
  if (!user) {
    return NextResponse.json(
      {
        message: "Unauthorized"
      },
      {
        status: 401,
        headers: corsHeaders
      }
    );
  }

  try {
    const client = await getClientPromise();

    const db = client.db("wad-01");
    const email = user.email;
    const profile = await db.collection("user").findOne({ email });
    console.log("profile: ", profile);
    return NextResponse.json(profile, {
      headers: corsHeaders
    })
  }
  catch(error) {
    console.log("Get Profile Exception: ", error.toString());
    return NextResponse.json(error.toString(), {
      headers: corsHeaders
    })
  }
}
```

In this GET profile method, the API will try to fetch user data using email that bundled in the Token rather than get the email explicitly.

10. Adding JWT_SECRET configuration into ".env.local" file:

```
MONGODB_URI=mongodb+srv://wad_01_admin:BPuarc7SPaW1D1Ws@wad-
01.4rhmvbv.mongodb.net/?appName=wad-01
ADMIN_SETUP_PASS=adminsetuppass
JWT_SECRET=myjwtsecret
```

Note that using "Access-Control-Allow-Origin": "*" together with cookies is not allowed by browsers. That is why we later switch to a fixed origin when using HttpOnly cookies.

## React JS Implementation

In this Authentication mechanism, we need to set up both Frontend and Backend to use the same mechanism which is JWT over HTTP-Only mode cookies.

There can be several components in React based on how you create your React code so far.

In brief, you need to:

- Implement User Context and User Provider (or you may name it Auth Context and Auth Provider)
- Use the local storage to keep the login state to prevent state lose when refreshing the browser or manually type the path.
- Implement login and logout functions to call backend API.
- Implement Login and Logout components.
- Implement Authentication Middleware.

In this example, I assume that you can apply the RequireAuth middleware component as well as UserContext and UserProvider from the React chapter.

1. Add ".env" in to ".gitignore" file.
2. Create a file named ".env" at root directory with this code:

```
VITE_API_URL=http://localhost:3000
```

3. Add UserContext.jsx and UserProvider.jsx from the React chapter into context directory.

4. Modify the UserProvider.jsx to use local storage and call to backend API:

```
//UserProvider.jsx

import { useContext, useState } from "react";
import { UserContext } from "./UserContext";

export function UserProvider ({children}) {

  // const initialUser = {
  //   isLoggedIn: false,
  //   name: '',
  //   email: ''
  // };
  const initialUser = JSON.parse(localStorage.getItem("session")) ?? {
    isLoggedIn: false, name: '', email: ''
  };

  const API_URL = import.meta.env.VITE_API_URL;
  const [user, setUser] = useState(initialUser);

  const login = async (email, password) => {
    try {
      const result = await fetch(`${API_URL}/api/user/login`, {
        method: "POST",
        headers: {
          'Content-Type': 'application/json'
        },
        body: JSON.stringify({
          email: email,
          password: password
        }),
        credentials: "include"
      });
      if (result.status != 200) {
        console.log("Login Exception: ", error);
        return false;
      }
      else {
        console.log("result: ", result);
        const newUser = { isLoggedIn: true, name: '', email: email };
        setUser(newUser);
        localStorage.setItem("session", JSON.stringify(newUser));
      }
    }
    catch (error) {
      console.log("Login Exception: ", error);
      return false;
    }
  }

  const logout = async () => {
    const result = await fetch(`${API_URL}/api/user/logout`, {
      method: "POST",
      credentials: "include"
    });
    const newUser = { isLoggedIn: false, name: '', email: '' };
    setUser(newUser);
    localStorage.setItem("session", JSON.stringify(newUser));
  }

  return (
    <UserContext.Provider value={{user, login, logout}}>
      {children}
    </UserContext.Provider>
  );
}

export function useUser () {
  return useContext(UserContext);
}
```

The credentials property in fetch method is the crucial part for our authentication mechanism; it must be set to “include” and you need to add this property for every API that need the cookies.

5. Insall react-router-dom.
6. Modify main.jsx to use the context and router.

```
import { StrictMode } from 'react'
import { createRoot } from 'react-dom/client'
import './index.css'
import App from './App.jsx'
import { BrowserRouter } from 'react-router-dom'
import { UserProvider } from './contexts/UserProvider.jsx'

createRoot(document.getElementById('root')).render(
  <StrictMode>
    <BrowserRouter>
      <UserProvider>
        <App />
      </UserProvider>
    </BrowserRouter>
  </StrictMode>,
)
```

7. Copy RequireAuth middleware from the React chapter.
8. Modify the App.jsx so to look something like:

```
import { useEffect, useState } from 'react';
import './App.css'
import { Route, Routes } from 'react-router-dom';
import TestApi from './components/TestApi';
import TestMongo from './components/TestMongo';
import RequireAuth from './middleware/RequireAuth';
import Profile from './components/Profile';
import Login from './components/Login';
import Logout from './components/Logout';

function App() {

  return(
    <Routes>
      <Route path='/test_api' element={<TestApi/>}/>
      <Route path='/test_mongo' element={<TestMongo/>}/>
      <Route path='/login' element={<Login/>}/>
      <Route path='/profile' element={
        <RequireAuth>
          <Profile/>
        </RequireAuth>
      }/>
      <Route path='/logout' element={
        <RequireAuth>
          <Logout/>
        </RequireAuth>
      }/>
    </Routes>
  );
}

export default App
```

Note that all components that require authentication must be nested in RequireAuth middleware.

9. Create Login.jsx component:

```
import { useRef, useState } from "react";
import { useUser } from "../contexts/UserProvider";
import { Navigate } from "react-router-dom";

export default function Login() {

  const [controlState, setControlState] = useState({
    isLoggingIn: false,
    isLoginError: false,
    isLoginOk: false
  });

  const emailRef = useRef();
  const passRef = useRef();
  const {user, login} = useUser();

  async function onLogin () {

    setControlState((prev)=>{
      return {
        ...prev,
        isLoggingIn: true
      }
    });
    const email = emailRef.current.value;
    const pass = passRef.current.value;

    const result = await login(email, pass);

    setControlState((prev) => {
      return {
        isLoggingIn: false,
        isLoginError: !result,
        isLoginOk: result
      }
    });
  }

  if (!user.isLoggedIn)
    return (
      <div>
        <table>
          <tbody>
            <tr>
              <th>Email</th>
              <td><input type="text" name="email" id="email" ref={emailRef}/>
</td>
            </tr>
            <tr>
              <th>Password</th>
              <td><input type="password" name="password" id="password"
ref={passRef}/> </td>
            </tr>
          </tbody>
        </table>
        <button onClick={onLogin}
disabled={controlState.isLoggingIn}>Login</button>
        {controlState.isLoginError && <div>Login incorrect</div>}
        {user.isLoggedIn && <div>Login Success</div>}
      </div>
    );
  else
    return (
      <Navigate to="/profile" replace />
    );
}
```

10. Create Logout.jsx component:

```
import { useEffect, useState } from "react";
import { useUser } from "../contexts/UserProvider";
import { Navigate } from "react-router-dom";
import Login from "./Login";

export default function Logout() {

  const [isLoading, setIsLoading] = useState(true);
  const { logout } = useUser();

  async function onLogout() {
    await logout();
    setIsLoading(false);
  }

  useEffect(()=>{
    onLogout();
  },[]);

  if (isLoading) {
    return (<><h3>Loging out...</h3></>);
  }
  else {
    return (<Navigate to={<Login/>} replace/>)
  }
}
```

11. Create Profile.jsx component:

```
import { useUser } from "../contexts/UserProvider";
import { useEffect, useState } from "react";

export default function Profile () {

  const { user, logout } = useUser();
  const [isLoading, setIsLoading] = useState(true);
  const [data, setData] = useState({});

  const API_URL = import.meta.env.VITE_API_URL;

  async function fetchProfile () {
    const result = await fetch(`${API_URL}/api/user/profile`, {
      credentials: "include"
    });
    if (result.status == 401) {
      logout();
    }
    else {
      const data = await result.json();
      console.log("data: ", data);
      setIsLoading(false);
      setData(data);
    }
  }

  useEffect(()=>{
    fetchProfile();
  },[]);

  return (
    <div>
      <h3>Profile...</h3>
      {
        isLoading ?
          <div>Loading...</div> :
          <div>
            ID: {data._id} <br/>
            Email: {data.email} <br/>
            First Name: {data.firstname} <br/>
            Last Name: {data.lastname} <br/>
          </div>
      }
    </div>
  )
}
```

How to test

- Brows to "/profile" it may fetch user data or redirect you to the login page.
- If login successfully, you shall be redirected to profile page.
- If you log in successfully and the code is correct, you shall see user information.
- Observe with Web Development Tools.

Note:

The .env file helps you to centralize the global parameter, so when you need to change it, you can change just one place, this also reduces bugs.

Don't forget to replace anywhere that you call API from static value and apply using API_URL instead. This will impact a lot when you deploy

# Next JS - 5

## File Upload Handling

This section will guide you how to implement Frontend and Backend codes to handle file upload, for example, the user profile image.

The normal solutions to handle users' files are:

- The uploaded files will be stored in specific directories. Some systems create a sub directory to match the user and that directory belongs just to the certain user. Some systems use a shared directory.
- For large systems they commonly do not store files in the same storage of the Backend server, but apply cloud storage like AWS S3, or any object storage service instead.
- The file names are usually hashed or encoded to a unique, using UUID or similar mechanism.
- The database then stores only the file source and its meta data but not the file content.

However, in this example, the files will be stored on the Backend server direct storage.

Profile Image example:

## Backend Code:

The simplest solution to get files from HTTP is using HTML multipart form-data format. We commonly don't apply JSON to upload files.

1. Install a uuid library with:

```
npm install uuid
```

2. Create a directory named "image" under "src\app\api\user\profile" directory.
3. Create a new file route.js with the following code:

```
import { verifyJWT } from "@/lib/auth";
import corsHeaders from "@/lib/cors";
import { getClientPromise } from "@/lib/mongodb";
import { NextResponse } from "next/server";
import { v4 as uuidv4 } from "uuid";
import path from "path";
import fs from "fs/promises";

export async function OPTIONS(req) {
  return new Response(null, {
    status: 200,
    headers: corsHeaders,
  });
}

// Helper to parse multipart form data (works in Node.js API routes)
async function parseMultipartFormData(req) {
  const contentType = req.headers.get("content-type") || "";
  if (!contentType.startsWith("multipart/form-data")) {
    throw new Error("Invalid content-type");
  }
  // Use undici's FormData parser (Node 18+)
  const formData = await req.formData();
  return formData;
}

export async function POST(req) {
  const user = verifyJWT(req);
  if (!user) {
    return NextResponse.json(
      { message: "Unauthorized" },
      { status: 401, headers: corsHeaders }
    );
  }

  let formData;

  try {
    formData = await parseMultipartFormData(req);
  } catch (err) {
    return NextResponse.json(
      { message: "Invalid form data" },
      { status: 400, headers: corsHeaders
    });
  }

  const file = formData.get("file");
  if (!file || typeof file === "string") {
    return NextResponse.json(
      { message: "No file uploaded" },
      { status: 400, headers: corsHeaders
    });
  }

  // Check if file is an image
  const allowedTypes = ["image/jpeg", "image/png", "image/gif", "image/webp"];
  if (!allowedTypes.includes(file.type)) {
    return NextResponse.json(
      { message: "Only image files allowed" },
      { status: 400, headers: corsHeaders
    });
  }

  // Generate unique filename
  const ext = file.name.split(".").pop();
  const filename = uuidv4() + "." + ext;
  const savePath = path.join(process.cwd(), "public", "profile-images",
filename);

  // Save file to disk
  const arrayBuffer = await file.arrayBuffer();
  await fs.writeFile(savePath, Buffer.from(arrayBuffer));

  // Update user in MongoDB
  try {
    const client = await getClientPromise();
    const db = client.db("wad-01");
    await db.collection("user").updateOne(
      { email: user.email },
      { $set: { profileImage: `/profile-images/${filename}` } }
    );
  } catch (err) {
    return NextResponse.json(
      { message: "Failed to update user" },
      { status: 500, headers: corsHeaders
    });
  }
  return NextResponse.json(
    { imageUrl: `/profile-images/${filename}` },
    { status: 200, headers: corsHeaders
  });
}

export async function DELETE (req) {
  const user = verifyJWT(req);
  if (!user) {
    return NextResponse.json(
      { message: "Unauthorized" },
      { status: 401, headers: corsHeaders }
    );
  }
  try {
    const client = await getClientPromise();
    const db = client.db("wad-01");
    const email = user.email;
    const profile = await db.collection("user").findOne({ email });
    if (profile && profile.profileImage) {
      const filePath = path.join(process.cwd(), "public", profile.profileImage);
      try {
        await fs.rm(filePath);
      } catch (err) {
        // File might not exist, ignore
      }
      await db.collection("user").updateOne({ email }, { $set: { profileImage:
null } });
    }
    return NextResponse.json(
      { message: "OK" },
      { status: 200, headers: corsHeaders
    });
  } catch (error) {
    return NextResponse.json(
      { message: error.toString() },
      { status: 500, headers: corsHeaders
    });
  }
}
```

4. Create a directory named "profile-images" under "public" directory.

In the code, you can notice that there are two separate steps, first save file step and then update user profile step.

This example also demonstrates how to detect if file data is not provided or it is string, then it shall return an error.

It also demonstrates how to filter the file type. This is the simplest solution to prevent tampering data, even if it is not the best one.

Then the file will be renamed using uuid method to get a unique name. This to ensure that many files can exist in the directory without name conflict.

Then the user data in Mongo is updated with new property, profileImage.

Note that this API requires HTTP-Only Cookie to deliver Token, so it cannot be tested separately.

## Frontend Code:

1. Replace the Profile.jsx with the following code:

```
import { useUser } from "../contexts/UserProvider";
import { useEffect, useState, useRef } from "react";

export default function Profile () {

  const { user, logout } = useUser();
  const [isLoading, setIsLoading] = useState(true);
  const [data, setData] = useState({});
  const [hasImage, setHasImage] = useState(false);
  const fileInputRef = useRef(null);

  const API_URL = import.meta.env.VITE_API_URL;
  console.log(`URL => ${API_URL}`);

  async function onUpdateImage () {
    const file = fileInputRef.current?.files[0];
    if (!file) {
      alert("Please select a file.");
      return;
    }
    const formData = new FormData();
    formData.append("file", file);
    try {
      const response = await fetch(`${API_URL}/api/user/profile/image`, {
        method: "POST",
        body: formData,
        credentials: "include"
      });
      if (response.ok) {
        alert("Image updated successfully.");
        fetchProfile();
      } else {
        alert("Failed to update image.");
      }
    } catch (err) {
      alert("Error uploading image.");
    }
  }

  async function fetchProfile () {
    const result = await fetch(`${API_URL}/api/user/profile`, {
      credentials: "include"
    });
    if (result.status == 401) {
      logout();
    }
    else {
      const data = await result.json();
      if (data.profileImage != null) {
        console.log("has image...");
        setHasImage(true);
      }
      console.log("data: ", data);
      setIsLoading(false);
      setData(data);
    }
  }

  useEffect(()=>{
    fetchProfile();
  },[]);

  return (
    <div>
      <h3>Profile...</h3>
      {
        isLoading ?
          <div>Loading...</div> :
          <div>
            ID: {data._id} <br/>
            Email: {data.email} <br/>
            First Name: {data.firstname} <br/>
            Last Name: {data.lastname} <br/>
            Image: {hasImage && <img src={`${API_URL}${data.profileImage}`}
width={150} height={150} />} <input type="file" id="profileImage"
name="profileImage" ref={fileInputRef}/> <button onClick={onUpdateImage}>Update
Image</button> <br/>
          </div>
      }
    </div>
  )
}
```

This code demonstrates how to create a multipart form-data to upload. The key statements are:

```
const formData = new FormData();
formData.append("file", file);
```

The formData is empty multipart form-data object, then we append the form with “file” field.

You can apply iteration in case of multiple uploads or use the array in file field. However, it depends on how the Backend API is provided.