

# Web Application Security



# Outline

1. Token based Token based Authentication & Authorization (JWT)
2. Authorization for Next.js apps

# Web Security Aspects

- **Authentication (Identity verification):**
  - Verify the identity of the user given the credentials received
  - Making sure the user is who he/she claims to be
- **Authorization (Controlling access):**
  - Determine if the user should be granted access to a particular resource/functionality.
- **Confidentiality:**
  - Encrypt sensitive data (e.g., credit card details) to prevent unauthorized access in transit or in storage
- **Data Integrity:**
  - Sign sensitive data (e.g., authentication token) to prevent the content from being tampered (e.g., changed in transit)

# Token based Authentication & Authorization



# Token based Authentication & Authorization

- After a successful authentication a **JSON Web Token (JWT)** is issued by the server and communicated to the client
- JWT token is a **signed json object** that contains:
  - Claims (i.e., information about *issuer* and the *user*)
  - Signature (encrypted hash for tamper proof & authenticity)
  - An expiration time
- Client must send JWT in an **HTTP authorization header** with subsequent Web API requests
- Web API (i.e., a resource) **validates** the received token and makes authorization decisions (typically based on the user's **role**)

# Web pages Session Management using JWT

- Implements stateless sessions by:
  - Creating utility functions (`createSession`, `verifySession`, `deleteSession`) to manage sessions
  - Using the [jose library](#) to create and verify signed/encrypted JWTs containing user object and expiration
  - Storing the JWT in a secure, `HttpOnly` cookie using `cookies()` from `next/headers`
  - Redirecting users after login/signup or when access is denied using `redirect()` from `next/navigation`

# Advantages of Token based Security

- A primary reason for using token-based authentication is that it is **stateless** and **scalable** authentication mechanism
  - It is suitable for SPA, Web APIs, Web pages and mobile apps
  - The token is stored on the client-side
  - The claims in a JWT are encoded as a **JSON** object that contains information that is useful for making authorization decisions
  - JWT is a simple and widely useful security token format with libraries available in most programming languages
- Can be used for **Single Sign-On**:
  - Sharing the JWT between different applications

# JWT Structure

JWT  
JSON WEB TOKEN



HEADER  
ALGORITHM  
& TOKEN TYPE

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}
```

+

PAYLOAD  
DATA

```
{  
  "sub": "1234567890",  
  "name": "John Doe",  
  "admin": true  
}
```

+

SIGNATURE  
VERIFICATION

```
HMACSHA256(  
  base64UrlEncode(header) + "." +  
  base64UrlEncode(payload), secretKey)
```

eyJhbGciOiJIub251In0.eyJpc3MiOiJqb2UiLA0KICJleHAiOiJlZzMD.4MTkzODAsDQogImh0dHA6Ly9leGFT

Header

Payload

Signature

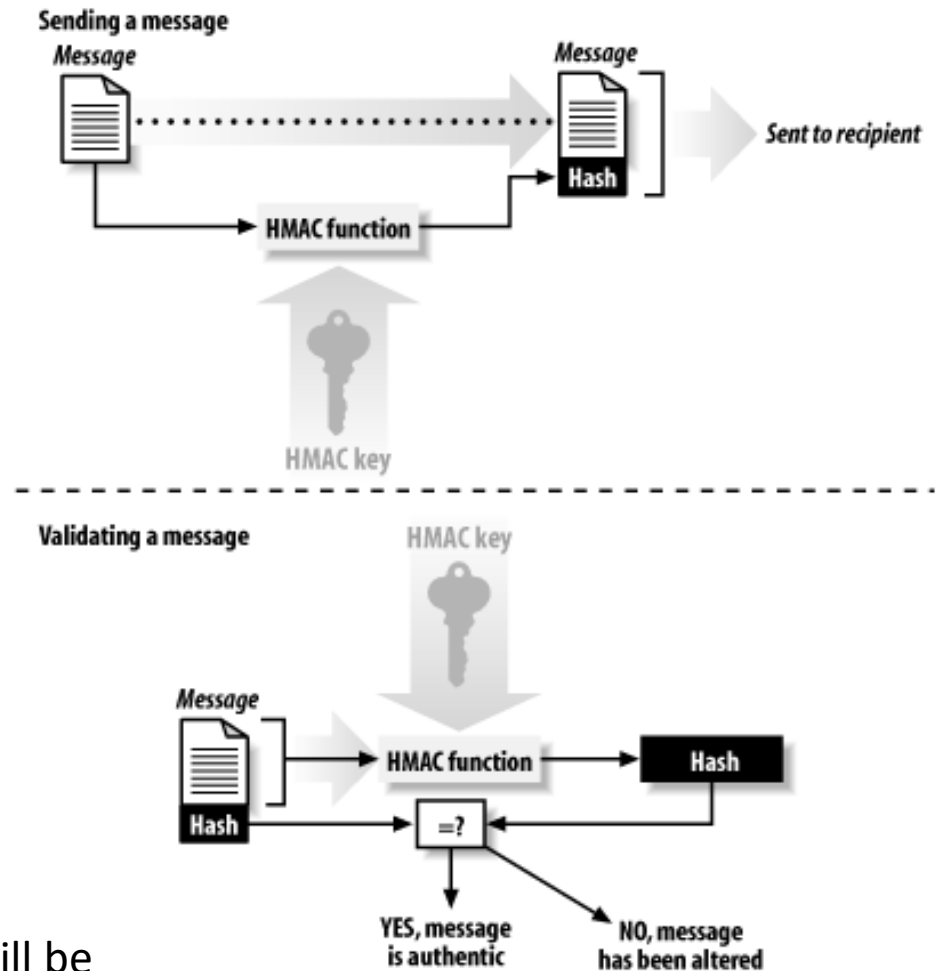


# Hash-based Message Authentication Code (HMAC)

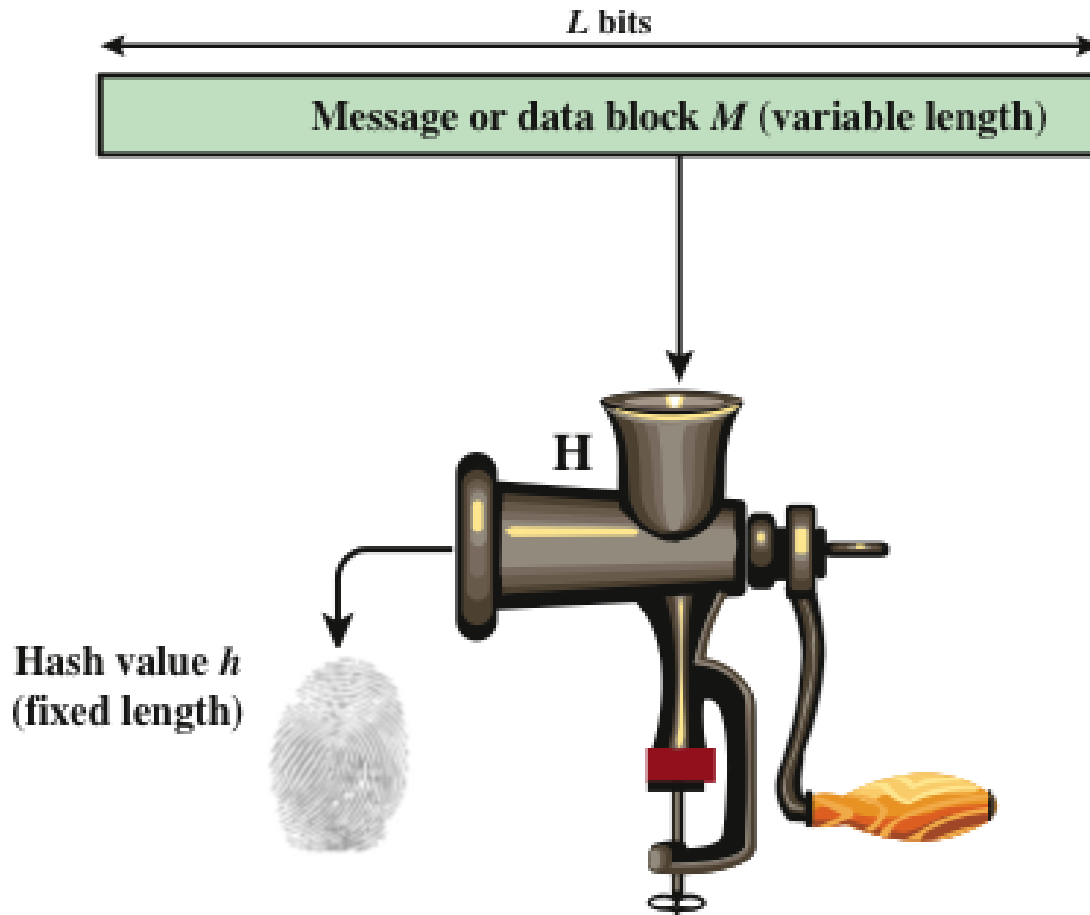
- **HMAC-SHA256** is often used for **signing JWT** to ensure its integrity
- HMAC-SHA256 is a *cryptographic hash function* that uses SHA256 hashing and **a secret key** to generate a MAC (i.e., JWT signature)
- The MAC is appended to the message sent
- MAC provides **message integrity**: Any manipulations of the message during transit will be detected by the receiver



An attacker who alters the message will be **unable** to alter the associated MAC value without knowledge of the secret key

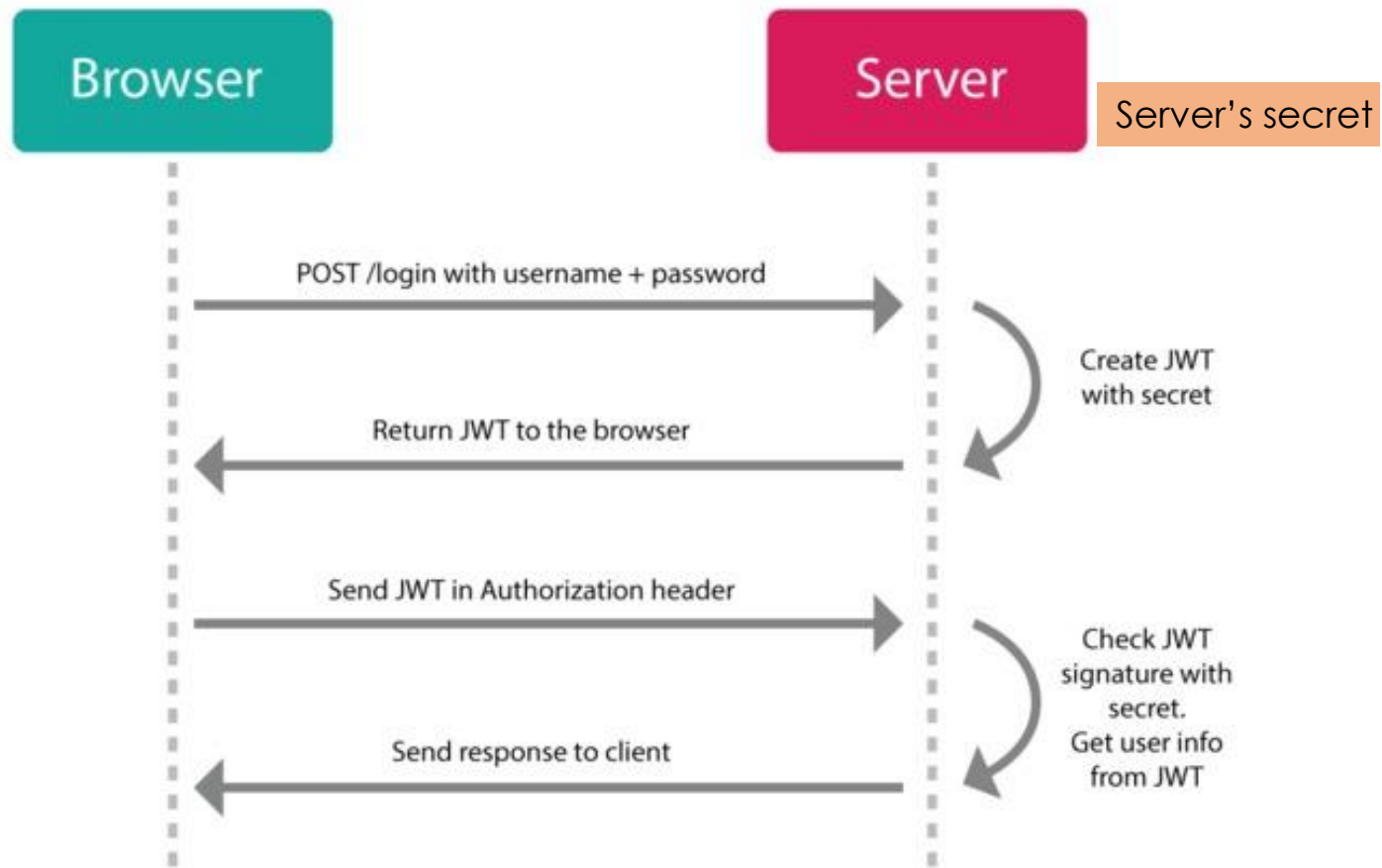


# Hashing



Hash functions are used to compute a digest of a message. It takes a variable size input, produce fixed size pseudorandom output

# JWT for Web API

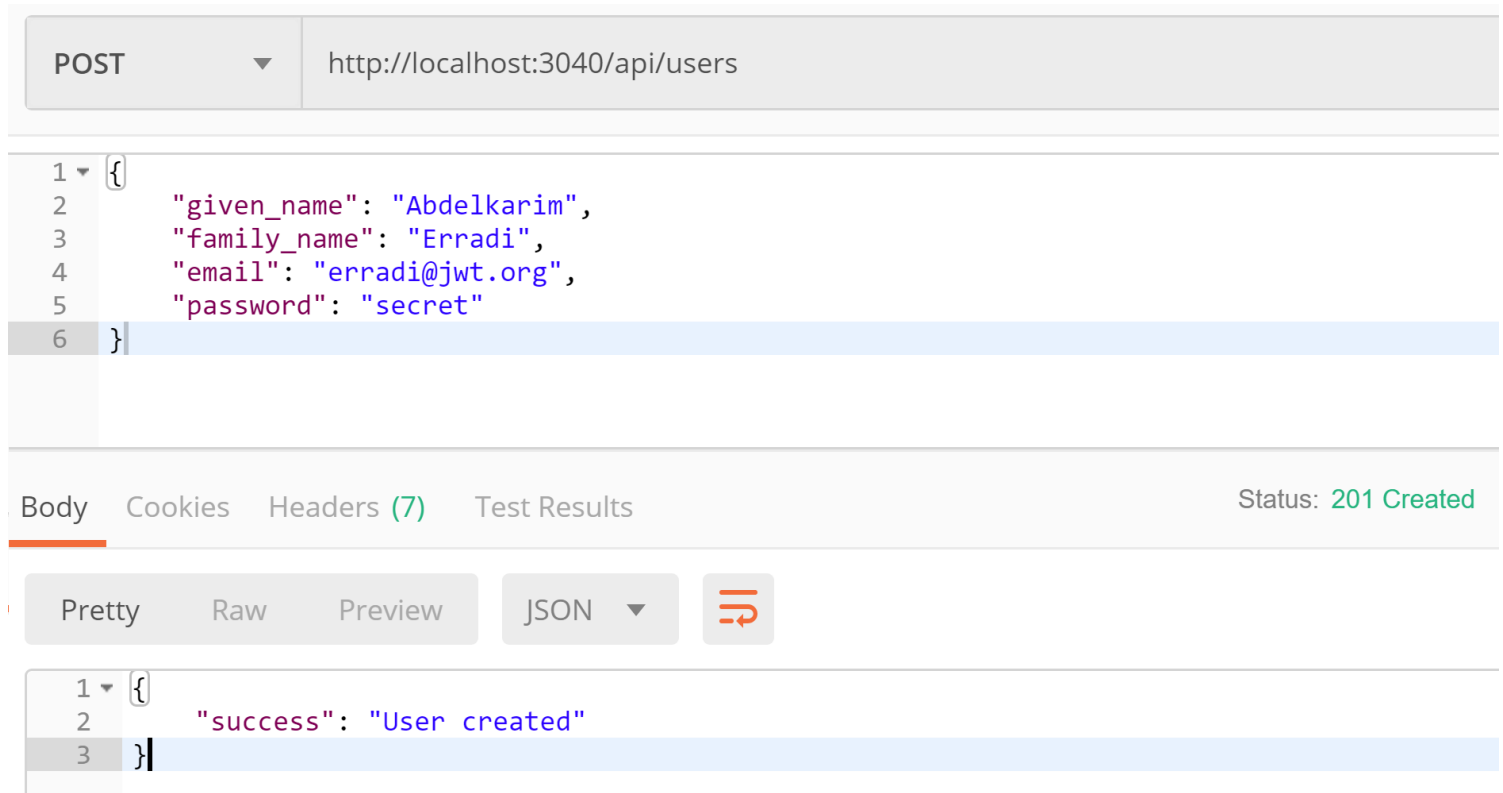


- Every request to a Web API must include a **JWT** in the Authorization header
- Web API checks that the JWT token is valid
- Web API uses info in the token (e.g., **role**) to make authorization decisions

# Sign-Up Example

- Sign up @ <http://localhost:3040/api/users>

Try it with  
Postman



# Successful Login to get JWT

- Sign in @ <http://localhost:3040/api/users/login>

The screenshot shows a REST client interface with a POST request to `http://localhost:3040/api/users/login`. The request body is a JSON object with `email: "erradi@jwt.org"` and `password: "secret"`. The response status is `200 OK`. The response body is a JSON object containing an `id_token` (highlighted in yellow), which is a long JWT string.

```
POST http://localhost:3040/api/users/login Send
```

```
{
  "email": "erradi@jwt.org",
  "password": "secret"
}
```

Body Cookies Headers (7) Test Results Status: 200 OK

Pretty Raw Preview JSON

```
{
  "id_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJvaWQiOiJerradi@jwt.org",
  "password": "secret"
}
```

# Use JWT to Access Protected Resource

- Get users <http://localhost:3040/api/users>

The screenshot shows a REST client interface with a GET request to `http://localhost:3040/api/users`. The request headers are set to `Content-Type: application/json` and `Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...`. The response body is a JSON array containing one user object. A callout box points to the Authorization header, stating: "Add the JWT token to standard **Authorization** header of HTTP requests to allow the Web API to verify it and allow access to resources".

Method	URL	Send
GET	http://localhost:3040/api/users	Send

Header	Value
Content-Type	application/json
Authorization	Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...

Body

```
1 [
2   {
3     "oidProvider": "local",
4     "role": "Admin",
5     "_id": "5cba142119e7a83ac0739b45",
6     "given_name": "Abdelkarim",
7     "family_name": "Erradi",
8     "email": "erradi@jwt.org",
9     "__v": 0
10  }
11 ]
```

Add the JWT token to standard **Authorization** header of HTTP requests to allow the Web API to verify it and allow access to resources

# Storing JWT in Browser Local Storage

Local Storage allows storing a set of name value pairs directly accessible with **client-side** JavaScript

- **Store**

```
localStorage.id_token = "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXLTJ5In06eyJ1bm90cnVzIjoiYm9keSIsImV4cCI6MTY1MjY0MDAwfQ=="
```

- **Retrieve**

```
console.log(localStorage.id_token)
```

- **Remove**

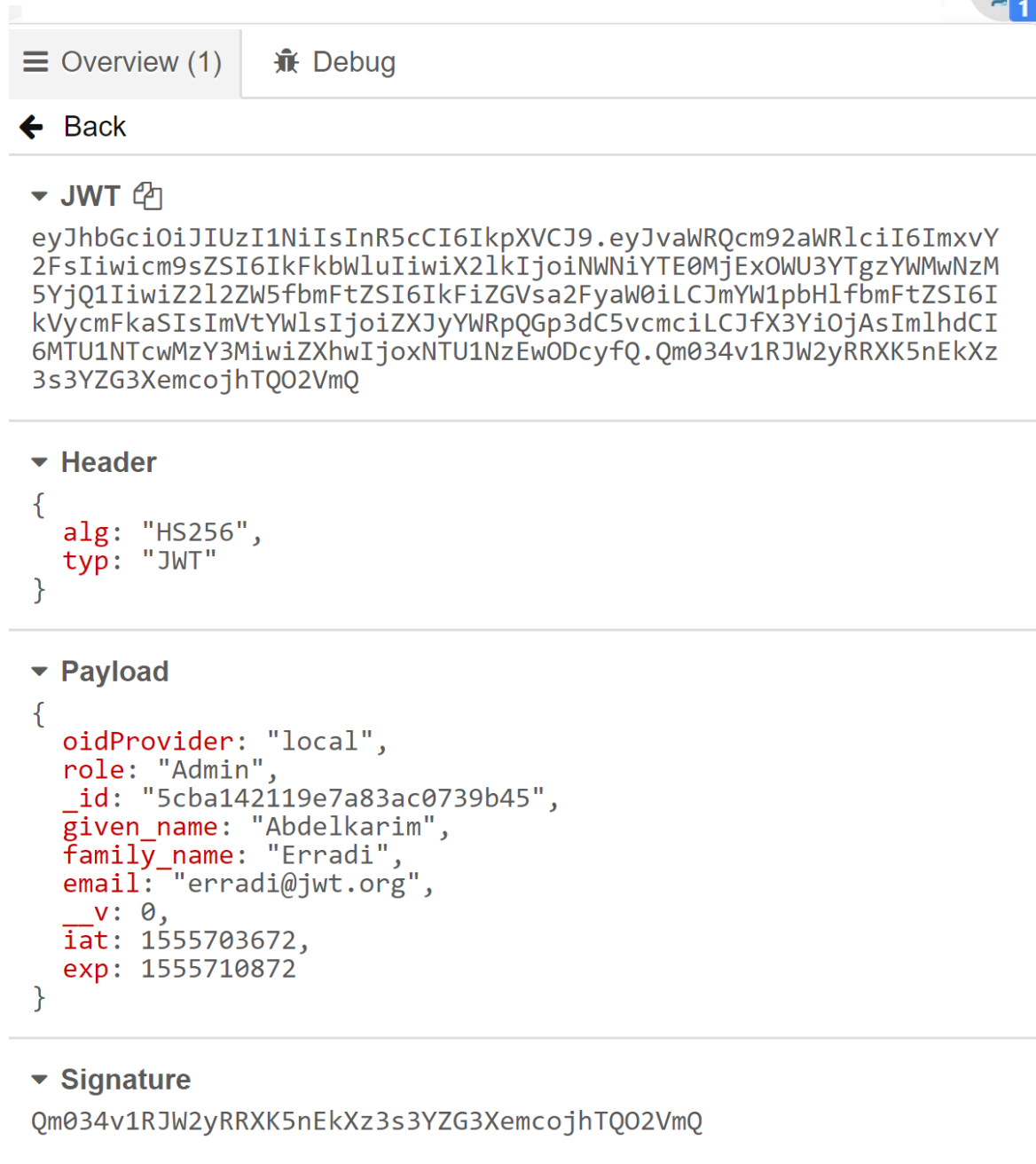
```
delete localStorage.id_token
```

- **Remove all saved data**  

```
localStorage.clear();
```



JWT Inspector is a chrome extension that lets you **decode** and **inspect** JWT in requests, and local storage





# 401 vs. 403

- ***401 Unauthorized***

- Should be returned in case of failed authentication

- ***403 Forbidden***

- Should be returned in case of failed authorization

- The user is authenticated but not authorized to perform the requested operation on the given resource



**Auth.js**



# NextAuth.js

- [Auth.js](#) is a flexible, easy to use and open-source authentication library for Web applications
  - Support multiple providers such as Facebook, Google, Twitter, Github, ... and the traditional email/password authentication
  - Supports passwordless sign in
- A working Next.js example is available at this [link](#)

# NextAuth.js Programming Steps (1 of 2)

1. Install NextAuth.js `npm install next-auth@beta`
2. Configure the Authentication Providers to be used such as GitHub, Google ([more info](#)):
  - Create `./app/api/auth/[...nextauth]/route.js`
  - Creating a new `auth.js` file at the root of your app and configure the desired auth provider such as Github provider with the **clientId** and the **secret**. Get them from <https://github.com/settings/applications/new>Enter them in the `.env` file in the root of the project

```
import GithubProvider from "next-auth/providers/github"
import NextAuth from "next-auth"

const authOptions = {
  providers: [
    GithubProvider({
      clientId: process.env.GITHUB_ID,
      clientSecret: process.env.GITHUB_SECRET,
    }) ] }
export default NextAuth(authOptions)
```

# Auth Web API

- Well, the magic has happened already. If we navigate to <http://localhost:3000/api/auth/signin> and you should see this

A screenshot of a web page showing a 'Sign in with GitHub' button. The button is a light gray rectangle with rounded corners, centered within a larger, slightly darker gray rounded rectangle. The text 'Sign in with GitHub' is in a dark gray, sans-serif font.

# Create and Configure OAuth Client

- Add/Update GitHub OAuth Client  
<https://github.com/settings/developers>
- Add/Update Google OAuth Client  
<https://console.developers.google.com/apis/credentials>
- Other Auth Providers provide similar UI to add and configure an OAuth Client (more [info](#))

## Register a new OAuth application

Application name \*

WebSec

Something users will recognize and trust.

Homepage URL \*

http://localhost:3000

The full URL to your application homepage.

Application description

Application description is optional

This is displayed to all users of your application.

Authorization callback URL \*

http://localhost:3000/api/auth

Your application's callback URL. Read our [OAuth documentation](#) for more information.

☐ Enable Device Flow

Allow this OAuth App to authorize users via the Device Flow.

Read the [Device Flow documentation](#) for more information.

Register application

Cancel

# Auth.js API

- creating a new auth.js file at the root of your app with the following content:

```
import NextAuth from "next-auth"

export const { handlers, signIn, signOut, auth } = NextAuth({
  providers: [],
})
```

- **signIn** and **signOut** functions can be used to perform the login and logout
- **auth** can be used to check the user login status and return the user's details

# Authorization for Next.js Apps





# Middleware.js to Check Authentication

- Use middleware.js to check if the user is **authenticated** and **authorized** before handling their request

```
const protectedRoutes = ["/", "/assessments", "/comments", "/workload-report"];

export function middleware(req) {
  const token = req.cookies.get("auth_token")?.value;
  const path = req.nextUrl.pathname;

  console.log("Middleware - Request Path:", path);
  // Check if the current path is a protected route
  const isProtectedRoute = protectedRoutes.some(
    (route) => path === route || (route !== "/" && path.startsWith(route))
  );
  // Redirect to login if accessing a protected route without a token
  if (isProtectedRoute && !token) {
    return NextResponse.redirect(new URL("/login", req.url));
  }
  // Continue if authenticated or accessing public route
  return NextResponse.next();
}
```

# Resources

- Good resource to learn about JWT

<https://jwt.io/>

- JWT Handbook

<https://auth0.com/resources/ebooks>

- OAuth 2

<https://www.oauth.com/>