MERN & Others

1. HOW JWT WORKS ?

JWT: it is a way for securely transmitting information between parties as a JSON object. It is used for authentication and information exchange in web applications. The internal architecture of a JWT consists of three main parts, separated by dots (.):

- HEADER: This part has two parts: the type of token (JWT) and the signing algorithm (like HMAC SHA256 or RSA).
- PAYLOAD: It contains the data which is to be passed.
- SIGNATURE: It is used to verify that the message wasn't altered along the way.

==**How JWT Works in Authentication:** ==

- User Login : When a user logs in, the server verifies their credentials (e.g., username and password).
- Token Generation : If the credentials are valid, the server generates a JWT containing the user's information (e.g., user ID, roles) in the payload and signs it with a secret key.
- Token Sending : The server sends the JWT back to the client, typically in the response body or as an HTTP-only cookie.
- Client Storage : The client stores the JWT, usually in local storage, session storage, or cookies.
- Authentication : For subsequent requests, the client sends the JWT in the Authorization header, typically as a Bearer token:
- Token Verification : The server verifies the JWT by checking its signature using the secret key. If the signature is valid, the server trusts the information in the payload and grants access to the requested resource.

2. COOKIES VS JWT

- Cookies are **small pieces of data stored in the user's browser**, automatically sent with every request to the same origin (domain). It is used for storing session identifiers (like `session_id`) and keeping the users logged in.

- **How it works (Session-Based Auth):**

1. User logs in → server creates a session.
2. Server stores session in memory or database.
3. Server sends back a cookie (`Set-Cookie`) to store session ID in browser.
4. Browser sends cookie with every request → server checks session ID.

- JWTs are self-contained **tokens** that hold **encoded user information**, usually stored in **localStorage** or **Authorization headers**.
- It is used for Stateless authentication which is common in REST APIs.
- ==Cookies and JWT both are used for Authentication and Authorization.

3. WHY JAVASCRIPT IS SINGLE THREADED?

- JavaScript was **designed to be single-threaded** to **simplify concurrency and avoid complex bugs** like **race conditions**.
- Since the **DOM is not thread-safe**, having multiple threads modifying it at the same time would be a disaster. So, JS was made **single-threaded** to make sure: Only one thing changes the DOM at a time → no conflicts, no data races.
- Then how does Javascript Handle async?
  - Even though it's single-threaded, JavaScript is **non-blocking** and **asynchronous**, thanks to the Event Loop + Web APIS (in browsers)
  - ex:

```
console.log("Start");
setTimeout(() => {
  console.log("Timeout done!");
}, 1000);

console.log("End");
```

- output:

```
Start
End
Timeout done!
```

4. WHAT IS ACCESS_TOKEN, REQUEST_TOKEN AND BEARER_TOKEN?

- So, a `request_token` is usually the temporary token we get at the start of an authentication process. It's kind of like a one-time pass that proves the user has authorized your app to access their data. Once that's confirmed, it gets exchanged for an `access_token`.
- The `access_token` is the main thing we use to make actual API requests on behalf of the user. It's like our verified ID for a limited time. We attach it in the headers, and it lets the backend know you're allowed to access certain resources.
- (Optional) Some systems also have a `refresh_token` which is a long-lived token used to obtain new access tokens after the current one expires. It helps maintain a seamless user experience without repeated logins.
- About the **Bearer token** — it's a type of `access_token` that's included in the `Authorization` header of HTTP requests. It's just a way of telling the server that the token being passed is a bearer token, which means "whoever holds this token has access." That's why we see the word `Bearer` in the Authorization header.
  ex:
  🎯 **Step-by-step Flow (with tokens)**

1. **User Clicks "Login with Google"**
   - The frontend app redirects the user to Google's login/authorization page.
2. **User Grants Permission**
   - After entering their credentials, the user agrees to let my app access their info (like email, profile, etc.).
3. **Request Token**
   - In some flows (like OAuth 1.0), Google sends back a `request_token`. This token is like: "Hey, the user said yes, but here's a temporary ticket — now prove you're legit and exchange it."
4. **Access Token (Main Key)**
   - The app then sends that `request_token` to Google again, and in return, gets an `access_token`. This `access_token` is what we will use to hit Google's APIs and get user data like email, profile picture, etc.
5. **Accessing Protected Resources**

- Now my app can say, “Yo Google, give me Ankit’s profile info,” and Google will check if the `access_token` is valid. If it is, you get the data.

6. **Refresh Token (if available)**
   - `access_tokens` usually expire in a few minutes or hours. So, if the API supports it, you’ll also get a `refresh_token`. This lets us silently get a new `access_token` without making the user log in again.

5. WHY DO WE HAVE PACKAGE.JSON & PACKAGE-LOCK.JSON?

So `package.json` is basically the **manifest** of a project — it lists out:

- The name, version, scripts
- And most importantly, the **dependencies** your app needs to run (like `express`, `mongoose`, etc.)
  But here’s the thing — `package.json` usually has **loose versions** like:
  `"express": "^4.18.2"`
  That `^` means — install the latest minor version above `4.18.2` (could be `4.19.x` etc.)
  That’s where `package-lock.json` comes in,
- It locks down the **exact version** of every dependency **and sub-dependency** installed.
- So if my teammate runs `npm install`, they get the **exact same versions** I had — ensuring no “it works on my machine” issues.