Agenda

1. Cookies
2. Jwt
3. authentication

Cookies

1. Think of a cookie like a small note that the web browser and the website pass back and forth. The first time you visit a website, it might give your browser a note (cookie) that says something like “This is visitor number 12345.” The next time you visit that website, your browser shows that note, and the website knows, this is visitor number 12345 again!
2. What Can Cookies Do?
    a. Remembering Login Information: So you don’t have to log in every single time you visit a new page on the same site.
    b. Storing User Preferences: Like the language you prefer, layout settings, or even what you had in your shopping cart.
    c. Tracking User Behavior: Websites can understand how you navigate the site, which helps in improving the user experience.
3. Originally, the web was stateless, meaning each request from a user's browser to a web server was treated as a new interaction, completely independent of any previous requests.
4. This statelessness posed significant limitations, particularly in terms of maintaining a user's state or session.

5. For instance, if you logged into a website and then navigated to another page, the server wouldn't recognize you as the same user who just logged in. There was no easy way for the server to remember your previous interactions.
6. **A cookie is a small piece of data sent from a website and stored on the user's computer by the user's web browser while the user is browsing.**

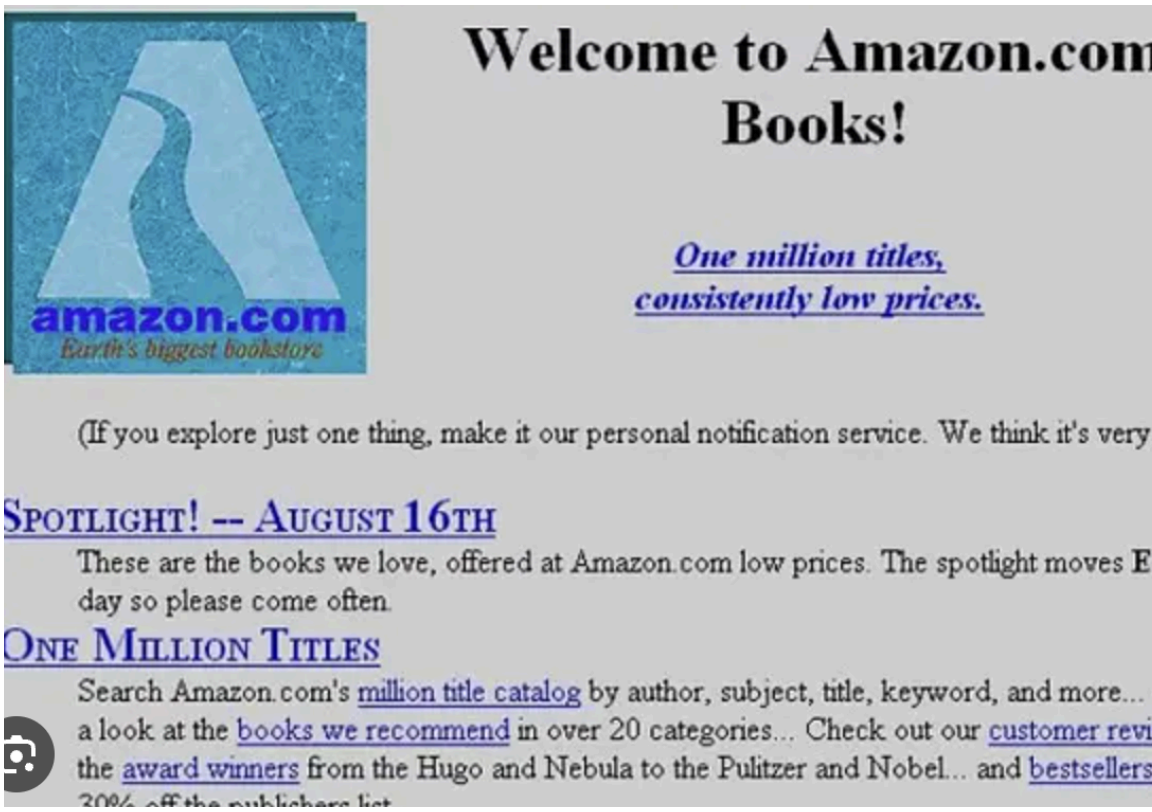
Cookies - Inception

1. The concept of cookies was created in 1994 by Lou Montulli, a programmer at Netscape Communications, which was then a leading company in browser development. Montulli was working on an e-commerce application for Netscape Navigator.
2. The initial purpose of cookies was to solve the problem of maintaining a shopping cart for the e-commerce platform. Since the HTTP protocol is stateless (each request is independent), there was no easy way to keep track of user interactions across multiple requests. **Cookies were created to store data about the user's session on their own computer, which could be retrieved by the server in subsequent requests.**

Sumarising

1. Role in Transition from Static to Dynamic Websites:

 Originally, websites were static, serving the same content to every visitor. The introduction of cookies was a key factor in the transition to dynamic websites, allowing for personalized experiences based on user interactions.

1. Earlier the websites were mostly static -serving the same html, css to every user
2. 

Welcome to Amazon.com Books!

One million titles, consistently low prices.

(If you explore just one thing, make it our personal notification service. We think it's very

SPOTLIGHT! -- AUGUST 16TH

These are the books we love, offered at Amazon.com low prices. The spotlight moves E
day so please come often.

ONE MILLION TITLES

Search Amazon.com's million title catalog by author, subject, title, keyword, and more...
a look at the books we recommend in over 20 categories... Check out our customer revi
the award winners from the Hugo and Nebula to the Pulitzer and Nobel... and bestsellers
30% off the publishers list

3. LAter the need for creating more personalized output was needed

        4. Now when you login, you have continue where you left, previous orders, products specific to your purchasing history

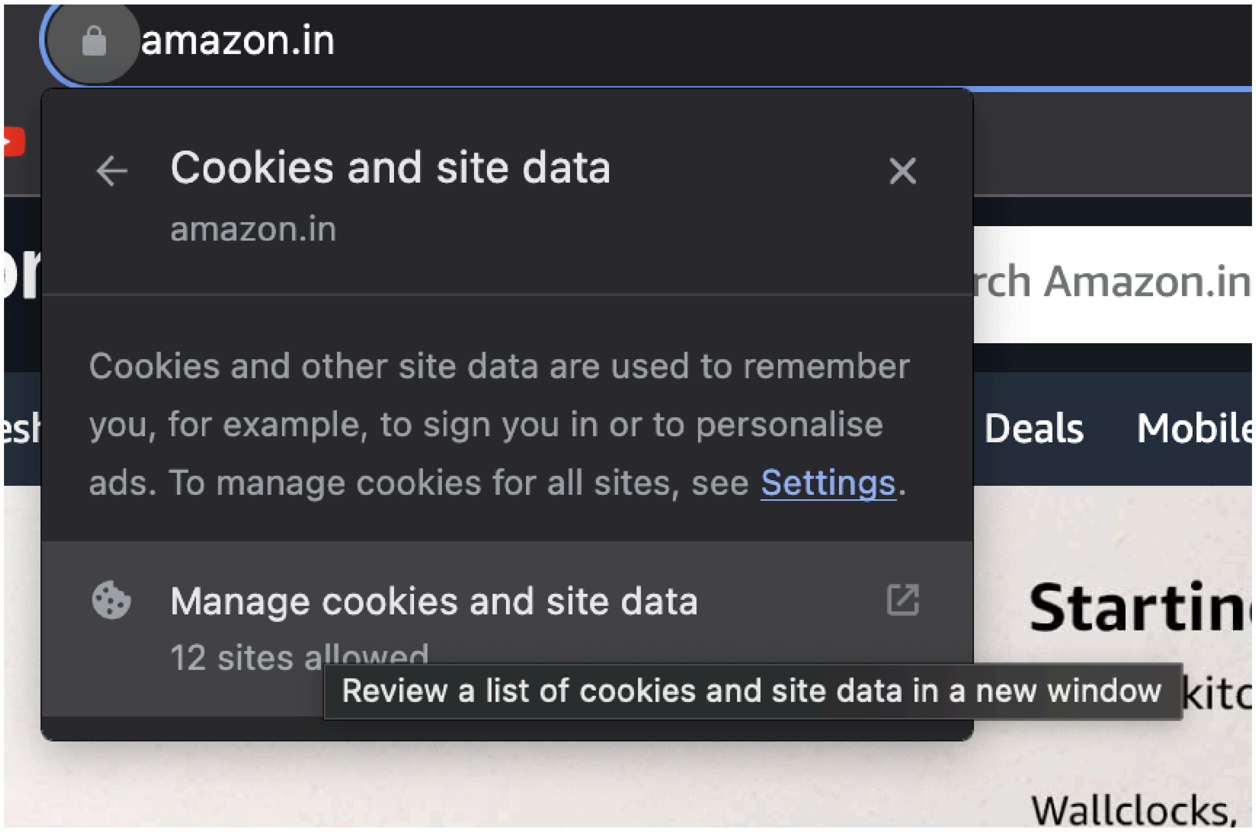
2. Remembering Users
3. Personalization and user experience
4. Privacy and Security Considerations: While cookies have been instrumental in enhancing user experience, they also raise concerns regarding privacy and data security. This has led to the development of regulations and best practices for cookie usage and user consent.

How are cookies provided

1. When client makes a request then the server along with the response, provides a cookie to the client
2. **For each domain, there is a key value pair (both string) like data stored on browser as cookies**
3. Then subsequent requests will carry this cookie data`

Third Party Cookies

1. Open amazon.in and go to the lock icon on the address bar

2. 
amazon.in

Cookies and site data

amazon.in

Cookies and other site data are used to remember you, for example, to sign you in or to personalise ads. To manage cookies for all sites, see Settings.

Manage cookies and site data

12 sites allowed

Review a list of cookies and site data in a new window

3. Role in advertisements. These thord party cookies track user activity. And that is why you see ads on no matter which website you go
4. Social Media buttons
    a. There are websites that have like and share button
    b. When we click on that button (and assuming we are logged on already), a request is sent to the social media server along with user cookies so that we dont have to login again
    c. When a user visits a website with social media buttons, the buttons are often loaded via scripts provided by the social media platforms.
    d. These scripts check for the presence of the platform's cookies in the user’s browser.

e. If the user is logged in (as indicated by the cookies), the scripts adjust the button's functionality accordingly – for instance, allowing the user to like or share content directly.
f. If there are no cookies (the user isn't logged in), the buttons might prompt the user to log in to the social media platform when clicked.
g.

How are cookies sent

1. Cookies are sent as part of http header
2. Http packets have header and body
3. When you first visit a website that uses cookies, the server sends an HTTP response with a Set-Cookie header.
4. The next time your browser makes an HTTP request to the same domain, it includes the cookies relevant to that domain in the Cookie header of the request.

Code

1. Create a new file called cookies.js to test out the cookie in express

```
const express = require('express');

const app = express();

// home
// products
// clearCookie
// will add a cookie and send as part of response
app.get('/', (req, res) => {
   res.json({message: 'Welcome to the home page'})
})
// will check if the user has already visited or visiting for
the first time
app.get('/products', (req, res) => {})
// clear the cookie
app.get('/clearCookie', (req, res) => {})

app.listen(3000, () => { console.log('Server is listening on
port 3000')});
```

2. Run the new file (node cookie.js) and see the output in postman
3. Verify that the cookie tab has not entry
4. Now we add one cookie with expiry date as one week

```
app.get('/', (req, res) => {
   res.cookie('pageVisited','home',{maxAge: 1000 * 60 * 60 * 24 * 7})

   res.json({message: 'Welcome to the home page'})
})
```

5. The points to note are the first two parameters are string that represents the key value pair.
6. The third parameter is the options parameters where we define the max age, security and some other configuration parameters
7. See the cookie now in postman
8. Understand that in the browser, we have the document object which has access to the cookies.
9. Inspect and see

10. We do not generally want our cookies to be accesses on the client side
11. The cookie can only be sent to the server in HTTP requests.
12. With HttpOnly, even if a malicious script runs on the webpage, it won't be able to access the cookie's data. This helps protect sensitive information in the cookie, like authentication tokens, from being compromised.

```
app.get('/', (req, res) => {
   res.cookie('pageVisited','home',{maxAge: 1000 * 60 * 60 * 24
* 7, httpOnly: true})

   res.json({message: 'Welcome to the home page'})
})
```

13. Notice the http only column in the postman

Accessing cookies

1. The cookie-parser is a middleware for the Node.js web framework Express. Its main function is to parse Cookie header and populate req.cookies
2. Essentially, it simplifies the process of accessing the cookie headers that a client sends with their HTTP request.
3. Install the cookie-parser package
4. Include in the file

```
const cookieParser = require('cookie-parser');
app.use(cookieParser());
app.get('/products', (req, res) => {
    console.log(req.cookies);
    res.json({message: 'Welcome to the products page'})

})
```

5. Maybe we want to write some logic based on what is in the cookies

```
app.get('/products', (req, res) => {
    console.log(req.cookies);
    const {pageVisited} = req.cookies;
    if(pageVisited) {
        res.json({message: 'Welcome to the products page'})
    } else {
        res.json({message: 'You are visiting for the first
time'})
    }

})
```

6. Update the cookie to expire in 10s and see the run

```
app.get('/', (req, res) => {
    // res.cookie('pageVisited','home',{maxAge: 1000 * 60 * 60 *
24 * 7, httpOnly: true})
    res.cookie('pageVisited','home',{maxAge: 10000, httpOnly:
true})

    res.json({message: 'Welcome to the home page'})
})
```

Update the '/' route for some more meaningful msg

```
app.get('/products', (req, res) => {
    console.log(req.cookies);
    const {pageVisited} = req.cookies;
    if(pageVisited) {
        res.json({message: 'Welcome to the products page'})
    } else {
        res.json({message: 'You are visiting for the first time.
please sign up / sign in'})
    }

})
```

7. Now run the postma. Go to product page. Wait for 10s and then go to /products route

Clearining Cookies

1. With the max age, the cookies will expire on its own

```
// clear the cookie
app.get('/clearCookie', (req, res) => {
    res.clearCookie('pageVisited',{path: '/'});
    res.json({message: 'cookie cleared'})
})
```

2. Different browsers may handle the absence of the path attribute differently when clearing cookies. Some browsers may assume the root path if none is provided, while others may use the current path of the request.
3. There could be multiple cookies with the same name but different paths. Specifying the path ensures you are clearing the correct cookie.
4. For example if we set a cookie for products path

```
res.cookie('product','bestseller',{maxAge: 1000 * 60 * 60
* 24 * 7, httpOnly: true, path: '/products'})
```

5. And if we print the cookies on the main route, we will not see the product path cookies
    a. Also in the postman, we will see only one cookie being passed

Authentication, authorization and identification

1. Identification: Telling Who You Are
    a. Web Application: This is like entering your username or email address into a login form. You're telling the system who you claim to be, but you haven't proven it yet.
2. Authentication: Proving Your Identity
    a. Web Application: This is akin to entering your password on the login page. The system checks the password to ensure it matches the one associated with the username or email you provided..
3. Authorization: Getting Access Based on Your Identity
    a. Once you're logged in, the system checks what permissions are associated with your account.

Different ways to authenticate

1. Login , otp, biometric

2. Web token
    a. This is again like When you first enter to a music festival, you show your ticket and receive a wristband. This wristband allows you to move in and out of different areas of the festival without having to show your ticket again. It's a proof that you've been authenticated at the entrance.
    b. In the digital world, a token serves a similar purpose as the festival wristband.
    c. When you log into a website or an application, instead of continuously entering your username and password for every action or transaction, the system gives you a token.
    d. This token is like a digital wristband that proves you are who you say you are and that you've already gone through the login process.
    e. A token in the context of web authentication is a string of characters that serves as a credential. Once a user is authenticated, the server generates this token and sends it to the user's device.
    f. This is often done using JSON Web Tokens (JWTs), which are encoded with a secret key.
    g. Tokens can be encrypted and signed, ensuring that the user data they carry is secure.
    h. The server doesn't need to remember the user's state between requests. The token itself contains all the necessary information, making the system more scalable and efficient.

i. Tokens should have an expiration time to prevent long-term use in case they are compromised.
j. What we are going to do is keep the token in our cookie so that subsequent requests carry the token

3. Process step by step
    a. User logs in using their credentials.
    b. Upon successful authentication, the server generates a JWT that contains claims (payload data) about the user, such as user ID, roles, or other relevant data.
    c. The server signs this JWT using a secret key (or a private key in the case of asymmetric signing).
    d. The server sends the signed JWT back to the client.
    e. he client typically stores this JWT in a secure way, such as in an HTTP-only cookie, sessionStorage, or localStorage
    f. For subsequent requests that require authentication or authorization, the client sends the JWT back to the server, usually in the Authorization header of the HTTP request.
    g. Upon receiving a request with a JWT, the server does NOT look up the token in a database to validate it. Instead, it verifies the token's signature using the same secret key (or the corresponding public key in the case of asymmetric signing) that was used to sign the JWT initially.
    h. If the signature is valid, the server trusts the content (claims) of the JWT.

   i. The server can then parse the JWT's payload to get the user's information and handle the request based on the claims present in the JWT.
   j. This signing of token can be understood by Imagine you are living in a time when messages are sent via letters, and each noble house has a unique wax seal.

JWT

1. Structure of JWT
   a. A JWT typically looks like this: xxxxx.yyyyy.zzzzz where:
   b. xxxxx is the Base64Url encoded header.
   c. yyyyy is the Base64Url encoded payload.
   d. zzzzz is the signature.
2. Header
   a. Content: The header typically consists of two parts: the type of the token, which is JWT, and the signing algorithm being used, such as HMAC SHA256 or RSA.
   b. Example: {"alg": "HS256", "typ": "JWT"}
   c. Purpose: It specifies the type of token and the algorithm used for signing the token.
3. Payload
   a. Content: The second part of the token is the payload, which contains the claims.

    b. Example: {"sub": "1234567890", "name": "John Doe", "admin": true}

4. Signature
    a. Creation: **To create the signature part you have to take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that**.
    b. Purpose: The signature is used to verify that the sender of the JWT is who it says it is and to ensure that the message wasn't changed along the way.
5. Extra note
    a. Base64 URL encoding is a variation of the standard Base64 encoding, designed to make Base64-encoded data safe to use in URLs and filenames. The main difference lies in the set of characters used and how they are handled in URL contexts.
    b. Standard Base64 Encoding:
    c. Base64 Encoding: It's a method of converting binary data into a string of ASCII characters. This is done to ensure safe transmission of data over protocols that might not handle binary data correctly.
    d. Character Set: Standard Base64 uses + and / in its character set along with alphanumeric characters (A-Z, a-z, 0-9).
    e. Issues with Standard Base64 in URLs:

f. The + and / characters have special meanings in URLs. For instance, + is treated as a space in URLs, and / is used as a path delimiter.
g. This can lead to issues when standard Base64-encoded strings are used in URLs or filenames.
h. Base64 URL Encoding:
i. To make Base64-encoded strings safe for use in URLs, the Base64 URL encoding scheme replaces + with - (minus) and / with _ (underscore).
j. Additionally, padding characters (usually = at the end of the encoded string) are often omitted in Base64 URL encoding, as they're not necessary for decoding and can cause issues in URLs.
k. Example:
l. Standard Base64: QmFzZTY0IEVuY29kaW5nIEV4YW1wbGU=
m. Base64 URL-encoded: QmFzZTY0IEVuY29kaW5nIEV4YW1wbGU
n. Use in JWT:
o. In JSON Web Tokens (JWTs), Base64 URL encoding is used for both the header and the payload. This ensures that the JWT can be safely used in URLs (e.g., in query parameters) or passed in HTTP headers without encoding issues.

1. Creation of JWT (when login)
2. Validation (accessing protecting routes / data)
3.

4. JWT has three parts - payload (data), algo name for encryption, signature (encryption of payload, algo and the secret key stored in the server)
5. Secret key is only stored on server
6. Server takes the payload, algo and using secret keys builds signature again and compares it with the receiving tokem
7. If resulting signature will not match then validation fails

Code JWT

1. Install package jsonwebtoken
2. Open the npm page and see the sign method

```
const jwt = require('jsonwebtoken');
const SECRET_KEY = "SomeRandomKey@12321";


app.get("/signin", async function (req, res) {
const payload = 1234;

 try {
   jwt.sign(
     { data: payload },
     SECRET_KEY,
     { expiresIn: "1h" }, // options
     function (err, data) { // callback
       if (err) {
         throw new Error(err.message);
       }
       res.cookie("token", data, {
         maxAge: 1000 * 60 * 60 * 24 * 7,
          httpOnly: true,
        });
        res.json({ message: data });
      }
    );
  } catch (err) {
    res.json({ message: err.message });
  }
});
```

3. Note the response as well as the cookie where we are passing the tokens
4. To get back the data or verify the token , we have the verify method on jwt

```
app.get("/verify", async function (req, res) {
   try {
       const { token } = req.cookies;
       const decoded = jwt.verify(token, SECRET_KEY);
       res.json({ message: decoded });
   } catch (err) {
       res.json({ message: err.message });
   }
})
```

5. Run in postman and see
6. Iat is the timestamp

Login, Signin, Protect Routes

1. Let us implement login, signin and a getUser route which will be protected
2. First of all, whether the login and signup be get or post

```
app.post('/signup', async function(req, res) {})
app.post('/login', async function(req, res) {})
app.get('userData',protectRoute, async function(req, res) {})
```

3. Signup handler

```
app.post('/signup', async function(req, res) {
   try{
       const userObject = req.body;
       console.log("userObject", userObject)
       const user = await User.create(userObject);
       res.json({message: 'user created successfully', data:
user})
   } catch(err) {
       res.json({message: err.message})
   }

})
```

Need to import few things

```
const User = require("./models/userModel");
const mongoose = require("mongoose");
require("dotenv").config();
app.use(express.json())
```

Run the mongoose connection

```
mongoose.connect(process.env.DB_URL).then((connection) => {
   console.log("DB connected");
 }).catch((err) => {
   console.log("DB connection failed", err.message);
 })
```

Login handler

1. What all should be done for login

```
app.post('/login', async function(req, res) {
   // validate credentials
   // send token
})
```

2. app.post("/login", async function (req, res) {

```
 // validate credentials
 try {
   const { email, password } = req.body;
   const user = await User.findOne({ email: email });
   console.log("user", user);
   if (!user) {
     res.status(400).json({ message: "user not found" });
   } else {
       console.log("user.password",user.password,"and
password",password)
     if (user.password == password) {
       const token = jwt.sign({ data: user._id }, SECRET_KEY);
       res.cookie("token", token, {
         maxAge: 1000 * 60 * 60 * 24 * 7,
         httpOnly: true,
       });
       res.json({ message: "user logged in successfully", data:
user });
```

```
     } else {
       res.status(400).json({ message: "invalid credentials"
});
     }
   }
 } catch (err) {
   res.json({ message: err.message });
 }
 // send token
});
```

3. See the new cookie on postman

Validating token on subsequent requests

```
const protectRoute = async function(req, res, next){
   try{
       const { token } = req.cookies;
       const decoded = jwt.verify(token, SECRET_KEY);
       const user = await User.findById(decoded.data);
       if(!user){
           res.status(400).json({message:"user not found"})
       }else{
           req.user = user;
           next();
       }
   }catch(err){
       res.status(400).json({message:err.message})
   }
}
app.get("/userData", protectRoute, async function (req, res) {
   res.status(200).json({message:"user data", data:req.user})
});
```

What should you do for the logout

```
app.get('/logout', function(req,res){
   res.clearCookie('token');
   res.json({message:"user logged out successfully"})
})
```