Web Sessions: Important Interview Questions

1️⃣ Understanding Statelessness & State Management

Q1: Why is HTTP considered a stateless protocol?

Answer: HTTP is considered stateless because each request from a client to a server is treated as an independent transaction. The server does not retain any memory of previous interactions. This simplifies server design but requires additional mechanisms like cookies, sessions, or tokens to maintain state.

Q2: How do web applications maintain state despite HTTP being stateless?

Answer: Web applications maintain state using:

- **Cookies:** Small pieces of data stored in the browser, often used to store session IDs.
- **Session Storage:** The server stores session data and links it to a session ID stored in the client's cookie.
- **JWT (JSON Web Token):** A self-contained token that carries session data and can be verified without needing server-side storage.
- **LocalStorage & SessionStorage:** Browser-based storage mechanisms that persist data on the client-side.

2️⃣ Session Management Techniques

Q3: What are the differences between server-side sessions and client-side tokens?

Answer:

Feature	Server-Side Sessions	Client-Side Tokens (JWT)
Storage	Stored on the server	Stored on the client
Scalability	Harder to scale (requires session replication)	Easier to scale (stateless authentication)
Security	More secure (session data never leaves the server)	Less secure (can be stolen if not properly protected)
Performance	Extra overhead for session lookup	Faster as it doesn't require DB lookup
Use Case	Best for traditional web apps	Best for microservices and APIs

Q4: When would you use JWT-based authentication instead of session-based authentication?

Answer: JWT-based authentication is preferred when:

- You need a **stateless** authentication mechanism for **scalability** (e.g., microservices, APIs).
- You want to **avoid server-side session storage** to reduce complexity.
- You need a token that can be **used across multiple services** (e.g., OAuth authentication).

However, JWTs should **not** be used when session invalidation or revocation is required, as they cannot be easily deleted from the client once issued.

Q5: How do cookies and sessions work together in session-based authentication?

Answer:

- When a user logs in, the server creates a **session** and stores session data (e.g., user ID) in memory or a database.
- The server generates a **session ID** and sends it to the client via a **cookie**.
- On subsequent requests, the client sends this **cookie** back, allowing the server to retrieve the session data.
- If the session is expired or deleted, authentication fails, requiring a new login.

3️⃣ Security & Best Practices

Q6: What is session hijacking, and how can it be prevented?

Answer: Session hijacking occurs when an attacker steals a user's **session ID** and uses it to impersonate them. It can be prevented by:

- Using **Secure, HttpOnly, and SameSite** cookie attributes.
- Implementing **SSL/TLS** to encrypt session data.
- Rotating session IDs after login or privilege escalation.
- Implementing **short-lived session expiration** and requiring re-authentication.

Q7: How does CSRF exploit session management, and what are the mitigation strategies?

Answer: CSRF (Cross-Site Request Forgery) occurs when an attacker tricks a user into submitting unintended requests on a trusted site where they are authenticated.

Mitigation strategies:

- Implement **CSRF tokens** that validate requests.
- Use **SameSite=Lax or Strict** cookies to restrict cross-site requests.
- Require **re-authentication** for sensitive operations.
- Enforce **CORS policies** to prevent unauthorized origins from making requests.

Q8: Why should cookies be set with Secure, HttpOnly, and SameSite attributes?

Answer: These attributes help protect session cookies:

- **Secure:** Ensures cookies are only sent over HTTPS.
- **HttpOnly:** Prevents JavaScript from accessing the cookie, reducing **XSS (Cross-Site Scripting) risks**.
- **SameSite:** Restricts sending cookies to the same site only, preventing **CSRF attacks**.

Q9: How can session management be scaled in distributed systems?

Answer:

- **Sticky Sessions:** Route users to the same server using load balancers.
- **Distributed Session Stores:** Store sessions in **Redis, Memcached, or databases**.
- **JWTs & Stateless Authentication:** Remove the need for server-side session storage by using tokens.

4️⃣ Real-World Scenarios & System Design

Q10: How does a load-balanced system handle session storage?

Answer:

- **Problem:** If a session is stored on one server, another server in the load balancer pool may not have access to it.
- **Solutions:**
    - Use **sticky sessions** to ensure requests go to the same server.
    - Store sessions in a **distributed session store** (e.g., Redis, DynamoDB).
    - Use **JWTs for stateless authentication**, removing session dependencies.

Q11: In a microservices architecture, how do you manage authentication across multiple services?

Answer:

- Use a **centralized authentication service** (e.g., OAuth, OpenID Connect).
- Issue **JWTs** that are valid across services.
- Services validate the token **without calling the authentication server** for each request.
- Use **refresh tokens** to allow long-term authentication without re-logging in frequently.

Q12: How do large-scale applications like Facebook, Amazon, or Google handle user sessions efficiently?

Answer:

- Use **token-based authentication** (JWTs, OAuth) for scalability.
- Store session data in **distributed systems** (e.g., Redis, Cassandra).

- Implement **single sign-on (SSO)** for seamless authentication across services.
- Use **short-lived tokens with refresh tokens** to enhance security.