1. What is Stateless Authentication?

It is a form of authentication where the server **does not keep any user session information**. All the required authentication data is carried in each request, typically via a **token** (like JWT). The process is:

- The user logs in.
- The server generates a **token** and sends it to the client.
- The client stores it (e.g., in **localStorage** or a **cookie**).
- With every request, the client sends this token (usually in the **Authorization header**).
- The server validates the token and processes the request — no memory of previous requests.

Advantages: Scalable, fast, easy to distribute.
Disadvantages: Harder to revoke, security risk if not stored safely, token size.

---

2. What and Why JWT?

JWT (JSON Web Token) is a compact, URL-safe token format used mainly for **stateless authentication**.

It has three parts:

- **Header**: Token type (JWT) and algorithm (e.g., HS256).
- **Payload**: Claims — info like user ID, roles, expiry.
- **Signature**: Validates integrity (i.e., it hasn't been tampered with).

Used because:

- No server session storage needed.
- Easy to pass across services or platforms.
- Self-contained and verifiable.
- Works across mobile/web/cloud.

---

3. Cookie-based vs Token-based Authentication

➔Cookie-based Authentication

- Server stores session (stateful).

- A **cookie** with a session ID is sent to the client.
- Browser automatically sends it with each request.
- Server finds session info using the cookie ID.

Pros:

- Built-in browser support.
- Works well for server-rendered apps (e.g., Razor pages, MVC).
- CSRF protection available.

Cons:

- Not ideal for APIs.
- Not cross-origin friendly.
- Less scalable due to server-side session.

➔ **Token-based Authentication (e.g., JWT)**

- **Stateless**: Server doesn't store session.
- The token itself has user info (claims).
- Client stores the token (e.g., localStorage).
- Sent manually with requests (e.g., Authorization: Bearer <token>).

Pros:

- Good for APIs, mobile, and SPAs.
- Easy to scale and decouple.
- Built-in role/claim info.

Cons:

- Needs HTTPS to prevent interception.
- Harder to revoke tokens.
- Manual CSRF protection in web apps.

---

4. Use Identity or JWT or Both?

Depends on your architecture and needs.

Use ASP.NET Identity if:

- You're building a **web app with login/session**.
- You want ready-to-use user management (register, login, roles, etc.).
- You prefer **cookie-based** authentication.

Use JWT if:

- You're building an **API** (e.g., for mobile or SPA).
- You don't want server-side session tracking.
- You issue **tokens manually** or integrate with external identity providers.

Use Both if:

- You're building **modern full-stack apps** (e.g., Identity for registration/login, JWT for token-based access).
- Common in SPAs or mobile apps where **ASP.NET Identity** issues a **JWT** after login.

---

5. Mention 3 Cases to Use OAuth

1. **External API Access**
   App accesses user's Google calendar using Google APIs with permission.
2. **Third-party App Integration**
   A fitness app reads data from Apple Health on behalf of the user.
3. **Social Login**
   User logs in using GitHub or Google — no need to manage passwords.

---

6. Interview: Do You Prefer JWT or Identity? Is This Logical?

JWT is a token format. ASP.NET Identity is a framework for managing users. I would choose based on the app type: Identity with cookies for traditional web apps, or Identity + JWT for SPAs/APIs. So, it's not an either-or — they can work together.