perplexity

Model Answer: Leveraging OAuth2 in a Microservices Application

"I leveraged OAuth2 to decouple authentication from my application logic and implement a secure, stateless Single Sign-On (SSO) system for our microservices architecture."

Here is how I implemented it specifically:

1. The Architecture (Context)

We had a React frontend and multiple Spring Boot microservices. We needed a secure way to manage user identities without each service handling passwords or sessions directly.

2. The Implementation (Action)

- **Identity Provider (IdP):** I integrated an Identity Provider (like **Keycloak**, **Auth0**, or **Okta**) to act as the Authorization Server.
- **Frontend (Public Client):**
  - For the React app, I used the **Authorization Code Flow with PKCE** (Proof Key for Code Exchange).
  - When a user logs in, they are redirected to the IdP. Upon success, the frontend receives an **Access Token (JWT)** and a Refresh Token.
  - This flow is crucial because it avoids exposing client secrets on the frontend.
- **Backend (Resource Server):**
  - The microservices act as Resource Servers. They don't store session data.
  - I configured an **API Gateway** (or Spring Security filter chain) to intercept every request.
  - It validates the **JWT signature** against the IdP's public key (JWK Set) to ensure the token hasn't been tampered with.
- **Service-to-Service Communication:**
  - For internal background jobs (e.g., Service A calling Service B), I implemented the **Client Credentials Flow**. Service A authenticates itself with a Client ID and Secret to get its own token, ensuring even internal traffic is authorized.

3. Handling Authorization (RBAC)

- I used the **Claims** inside the JWT (e.g., `roles: ["admin", "editor"]`) to implement Role-Based Access Control (RBAC).
- Instead of looking up permissions in a database for every request, the service simply reads the scope/roles directly from the token, significantly reducing database load.

4. The Result

- **Security:** We removed all password handling from our code; credentials never touch our backend services.
- **Scalability:** Since JWTs are self-contained (stateless), we could horizontally scale our services without worrying about sticky sessions.
- **User Experience:** Users got a seamless SSO experience; logging into one app gave them access to the entire suite without re-authenticating.

Key Concepts to Mention if Probed:

- **Why PKCE?** "To prevent authorization code interception attacks on public clients (SPAs/Mobile)."
- **Why JWT?** "To allow stateless authentication; the token contains all necessary user data."
- **Refresh Tokens:** "To acquire new access tokens silently without forcing the user to log in again when the short-lived access token expires."

⁂

1. https://ppl-ai-file-upload.s3.amazonaws.com/web/direct-files/attachments/images/81815274/429036f4-3bc3-4a7a-8c6f-90246e73b9b9/image.jpg
2. https://ppl-ai-file-upload.s3.amazonaws.com/web/direct-files/attachments/images/81815274/24f18c0a-b56e-4862-a619-e2959036a5c2/image.jpg
3. https://ppl-ai-file-upload.s3.amazonaws.com/web/direct-files/attachments/images/81815274/97e9ed86-ed11-4035-a18b-1eb05c1bab4e/image.jpg
4. https://ppl-ai-file-upload.s3.amazonaws.com/web/direct-files/attachments/images/81815274/87f62423-96f2-4071-9802-8f6699e0ecd8/image.jpg