Lexis Advance CII 2026

CII : Continuous Improvement and Innovation

RELX Wirearchy –Secure Microservices with OAuth2 and JWT

Business_scenario:
Currently, organizations use microservices for scalability, but securing them across distributed systems is complex. Developers spend extra time managing tokens and custom security, slowing delivery and increasing risk.

Presenter


Vivek Kumar & Soumyajit Dhar

Hosted by


G. Tamilselvan
Saravanan.P

Date and Time

27/02/2026

3:00 PM – 4:00 PM

Solution: Securing microservices can be efficiently implemented using OAuth2 authorization flows combined with JWT-based token propagation. OAuth2 manages client authentication and token issuance, while JWTs encapsulate user claims in a cryptographically signed structure that each microservice can independently validate. This eliminates centralized session management, reduces latency, and enables scalable, stateless authentication across the microservice ecosystem.

Benefits:

- Ensures strong, token-based authentication and authorization across all microservices.
- Stateless JWT tokens improve performance and scalability without relying on shared sessions.
- OAuth2 flows protect user credentials while enabling secure access delegation.

Target Audience : RELX associates

LexisNexis®

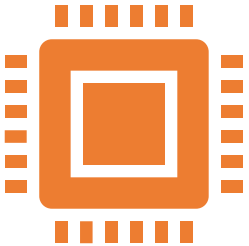
cognizant

Secure Microservices with OAuth2 OAuth2 and JWT: Smarter Security, Trust at Scale

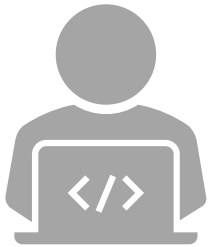
Microservices bring flexibility and scalability, but securing them is a challenge. OAuth2 enables delegated authorization, while JWT provides lightweight, stateless authentication. Together, they deliver smarter security, fine-grained access control, and seamless communication — helping developers build faster while keeping systems resilient and trustworthy.

To know more about how OAuth2 and JWT can be used to secure microservices, please join the session
presented by: **Vivek Kumar & Soumyajit Dhar**

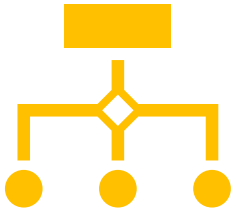
Securing Microservices
with OAuth2 and JWT



Modern enterprise applications rely heavily on microservices architectures, creating complex distributed systems that demand robust security strategies.



Through years of implementing authentication solutions across healthcare platforms, financial services, and e-commerce systems, I've learned that securing Java microservices requires more than adding authentication tokens—it demands a comprehensive approach that balances security, performance, and operational efficiency.



This guide outlines effective OAuth2 and JWT implementation patterns that I've used successfully in production, for systems handling millions of transactions daily and platforms with thousands of users concurrently. We'll explore practical strategies that address real-world challenges while maintaining the scalability and flexibility that make microservices architectures valuable.

Microservices architecture is a software development approach where an application is structured as a collection of loosely coupled, independently deployable services. Each service focuses on a specific business capability, enabling teams to develop, deploy. Scale them independently. This contrasts with monolithic architectures, where the entire application is built as a single unit.

- **Benefits of Microservices:**
    - **Independent Scalability:** Individual services can be scaled based on demand.
    - **Technology Diversity:** Teams can choose the best technology stack for each service.
    - **Faster Development Cycles:** Smaller codebases and independent deployments allow for quicker releases.
    - **Improved Fault Isolation:** Failures in one service are less likely to impact other services.
- **Challenges of Microservices:**
    - **Complexity:** Managing a distributed system is more complex than managing a monolith.
    - **Inter-service Communication:** Ensuring reliable communication between services can be challenging.
    - **Security:** Securing inter-service communication and external access points requires careful planning.
    - **Observability:** Monitoring and troubleshooting a distributed system requires robust logging, tracing. Monitoring tools.

The Microservices Landscape: Why Security is is Paramount

Microservices multiply network boundaries, credentials, and failure modes. Each service-to-service call is an attack surface: lateral surface: lateral movement, privilege escalation, data leakage and compromised tokens. Secure design must combine strong identity, strong identity, least privilege, and robust token controls to prevent breaches while preserving scalability and developer velocity. developer velocity. Observability, automated secrets rotation and layered defence (network policies, mTLS, authorisation) are authorisation) are essential.

Attack Surface

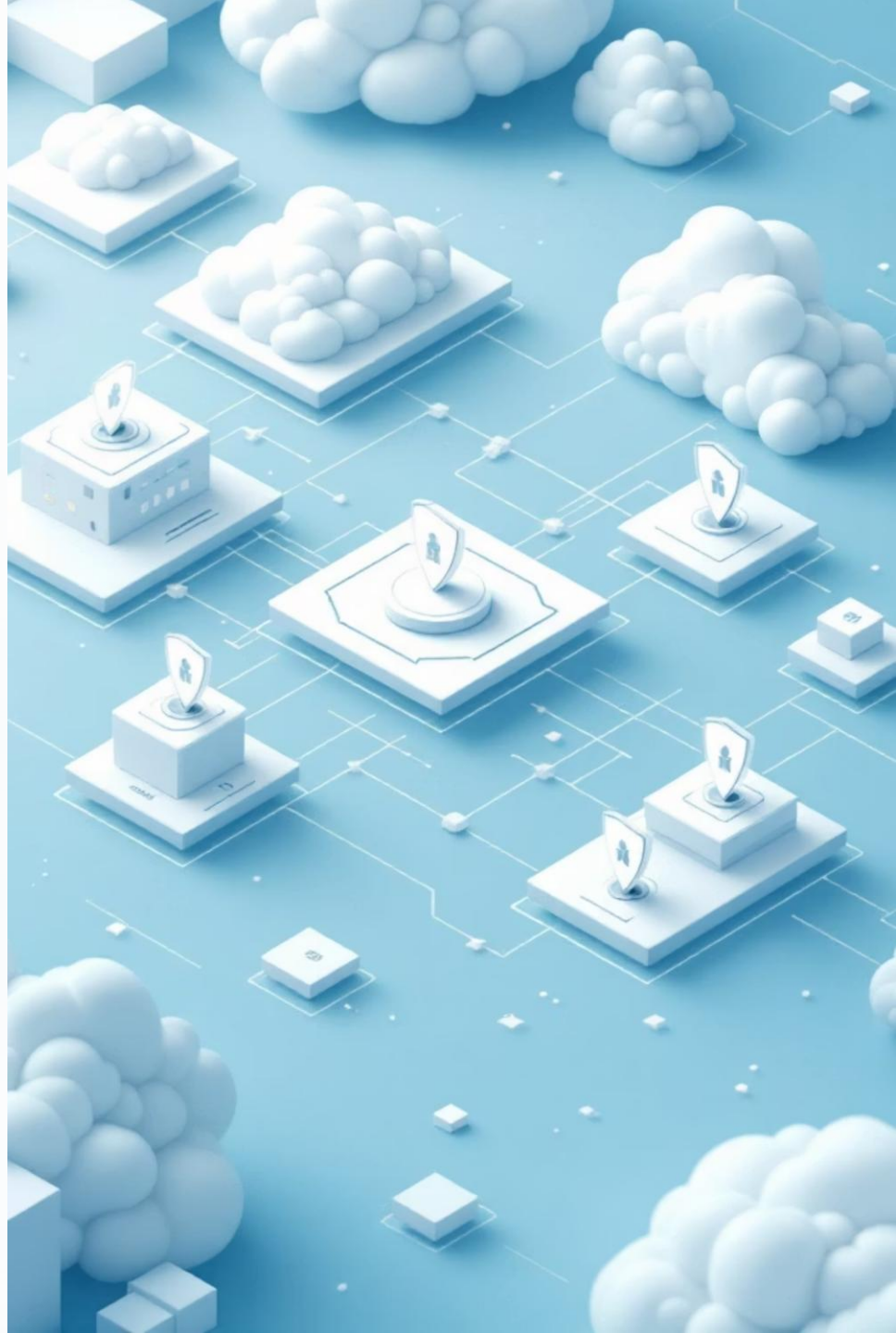
Distributed endpoints increase points of compromise—protect every API boundary.

Identity First

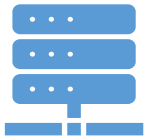
Treat services and users as identities; issue short-lived tokens and verify on every request.

Scale & Safety

Security must scale with traffic—automate policy, rotation and observability.



The Role of API Gateways in Microservices Security



In a microservices architecture, an API gateway acts as a single entry point for all client requests. It sits in front of the microservices and handles tasks such as routing, authentication, authorization, rate limiting. Request transformation.

Key Functions of an API Gateway:

Routing: Directs incoming requests to the appropriate microservice.

Authentication: Verifies the identity of the client.

Authorization: Determines whether the client has permission to access the requested resource.

Rate Limiting: Prevents abuse and ensures fair usage of the API.

Request Transformation: Modifies requests and responses to be compatible with different clients and microservices.

Caching: Improves performance by caching frequently accessed data.

Benefits of Using an API Gateway:

Simplified Client Communication: Clients interact with a single endpoint instead of multiple microservices.

Centralized Security: Security policies can be enforced at the gateway level, reducing the burden on individual microservices.

Improved Performance: Caching and request transformation can optimize performance.

Increased Observability: The gateway provides a central point for monitoring and logging API traffic.

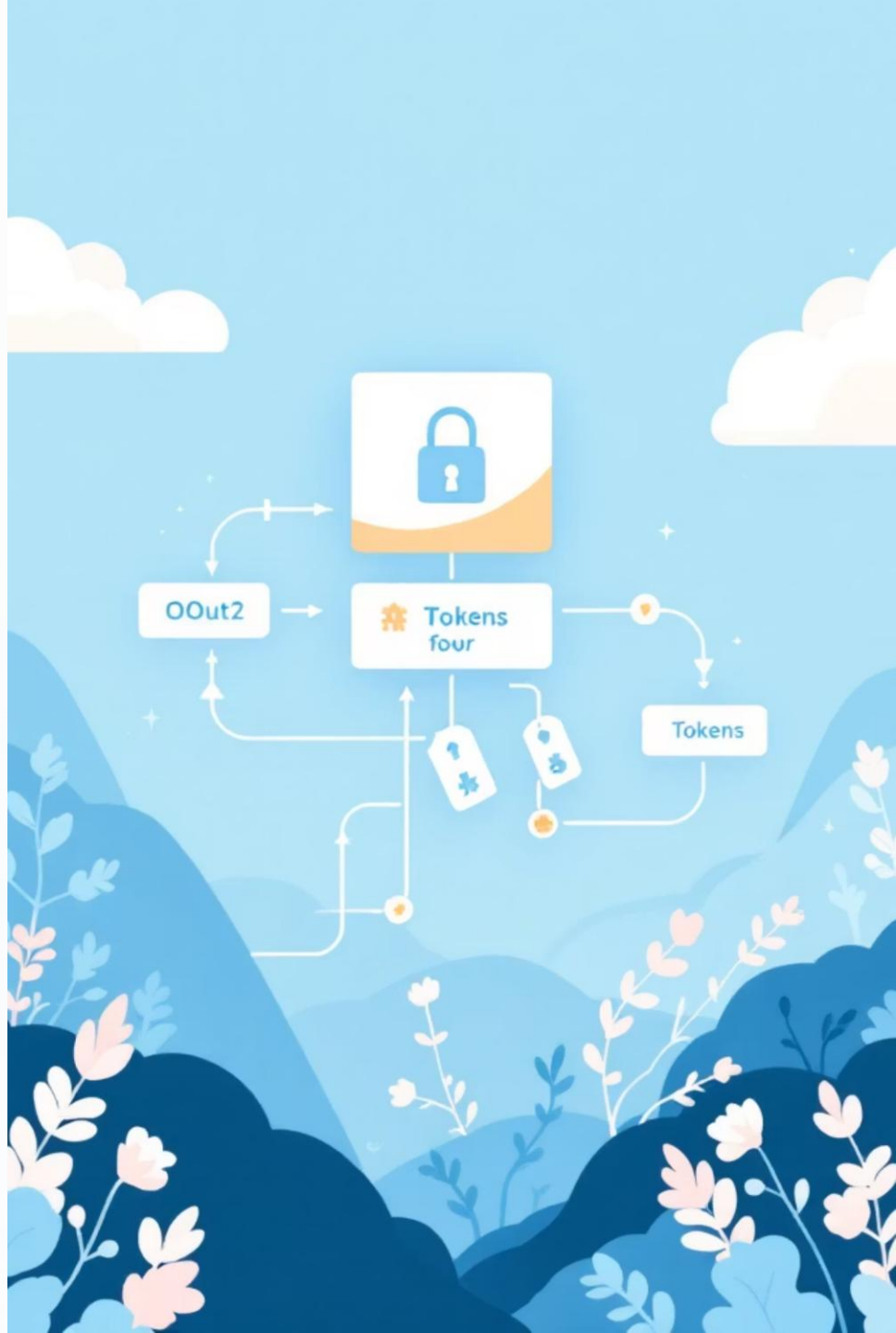
- **Introduction to OAuth 2. 0 and JWT**
- OAuth 2. 0 is an authorization framework that enables third-party applications to obtain limited access to an HTTP service, either on behalf of a resource owner or by allowing the third-party application to obtain access on its own behalf. JSON Web Token (JWT) is a standard for creating access tokens that assert claims about a user or application.
- **OAuth 2. 0 Flows:** OAuth 2. 0 defines several authorization flows, each suited to different scenarios. Common flows include:
    - **Authorization Code Grant:** Used for web applications and mobile apps where the client can securely store a client secret.
    - **Implicit Grant:** Used for browser-based applications (e. G. , single-page applications) where the client secret cannot be securely stored. (Less secure and generally discouraged).
    - **Resource Owner Password Credentials Grant:** Used for trusted applications where the client directly requests access using the resource owner's credentials. (Should be used with caution).
    - **Client Credentials Grant:** Used for application-to-application authentication where the client is acting on its own behalf, not on behalf of a user.
- **JWT Structure:** A JWT consists of three parts:
    - **Header:** Specifies the algorithm used to sign the token (e. G. , HS256, RS256) and the token type.
    - **Payload:** Contains claims about the user or application, such as the user ID, roles. Expiration time.
    - **Signature:** A cryptographic signature that verifies the integrity of the token.
- **Benefits of Using OAuth 2. 0 and JWT:**
    - **Delegated Authorization:** Allows users to grant limited access to their resources without sharing their credentials.
    - **Stateless Authentication:** JWTs contain all the necessary details to authenticate a request, eliminating the need for server-side sessions.
    - **Interoperability:** OAuth 2. 0 and JWT are widely supported standards, making it easy to integrate with different systems.
    - **Scalability:** JWTs can be easily validated by any service that has access to the signing key, enabling horizontal scalability.

Introduction to OAuth2: The Authorisation Framework

OAuth2 is a delegation protocol: it authorises clients to act on behalf of resource owners by owners by issuing access tokens. It separates authentication (who you are) from authorisation authorisation (what you can do). Core components: Resource Owner, Client, Authorization Authorization Server and Resource Server. OAuth2 focuses on secure token issuance and issuance and lifecycle rather than authentication specifics—often combined with OpenID OpenID Connect for identity.

- **Key Concepts**
  - Access Token: short-lived credential for APIs
  - Refresh Token: renews access tokens securely
  - Scopes: granular permissions in tokens

- **Roles**
  - Authorization Server: issues tokens
  - Resource Server: validates tokens, enforces scopes



OAuth2: The Delegation Protocol Protocol — What and Why

OAuth2 is an authorization framework that lets users grant limited access to their resources on resources on one site (resource server) to another site (client) without sharing credentials. It credentials. It separates authentication from authorization, enabling third-party apps to act on apps to act on behalf of a user. Why we need it: improved security (no password sharing), fine-sharing), fine-grained scopes, revocable access, and better UX through consent screens. screens.

Security

Limit exposure by issuing short-lived tokens instead of passwords.

Delegation

Grant specific permissions (scopes) to clients for a defined time.

Control

Revoke and rotate tokens centrally without changing user credentials.

OAuth2 Grant Types: Choosing the Right Flow for Your Microservices

Selecting the correct grant is critical: Client Credentials for service-to-service, Authorization Code (with PKCE) for user-facing apps, Device Code for constrained devices. Avoid deprecated flows (Implicit, Resource Owner Password). Consider trust boundaries, token lifetimes, and client capabilities. For microservices, prefer Client Credentials + mutual TLS or mTLS-backed token issuance for high-assurance service identity.

Client Credentials

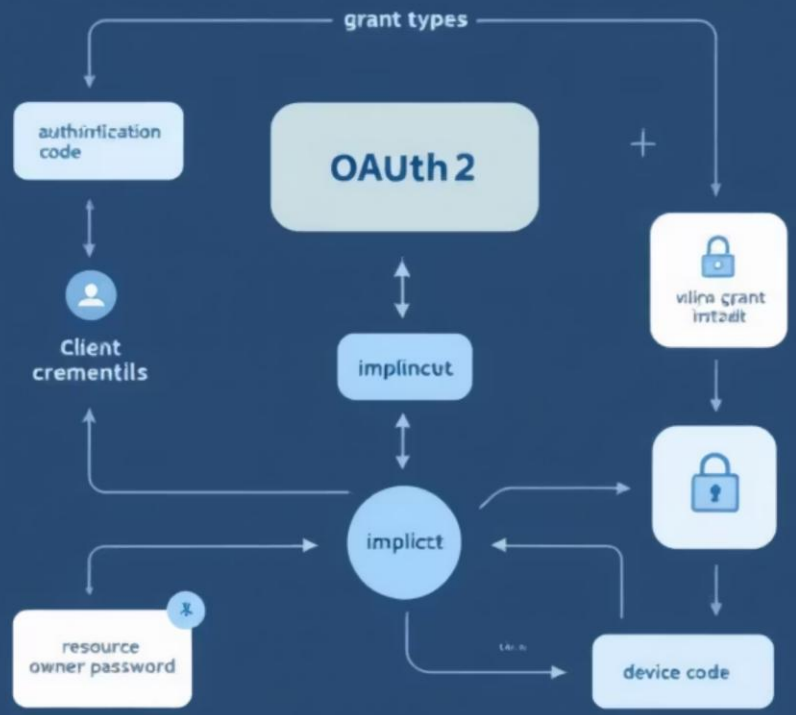
Best for backend services: machine identity, short-lived tokens, no user context.

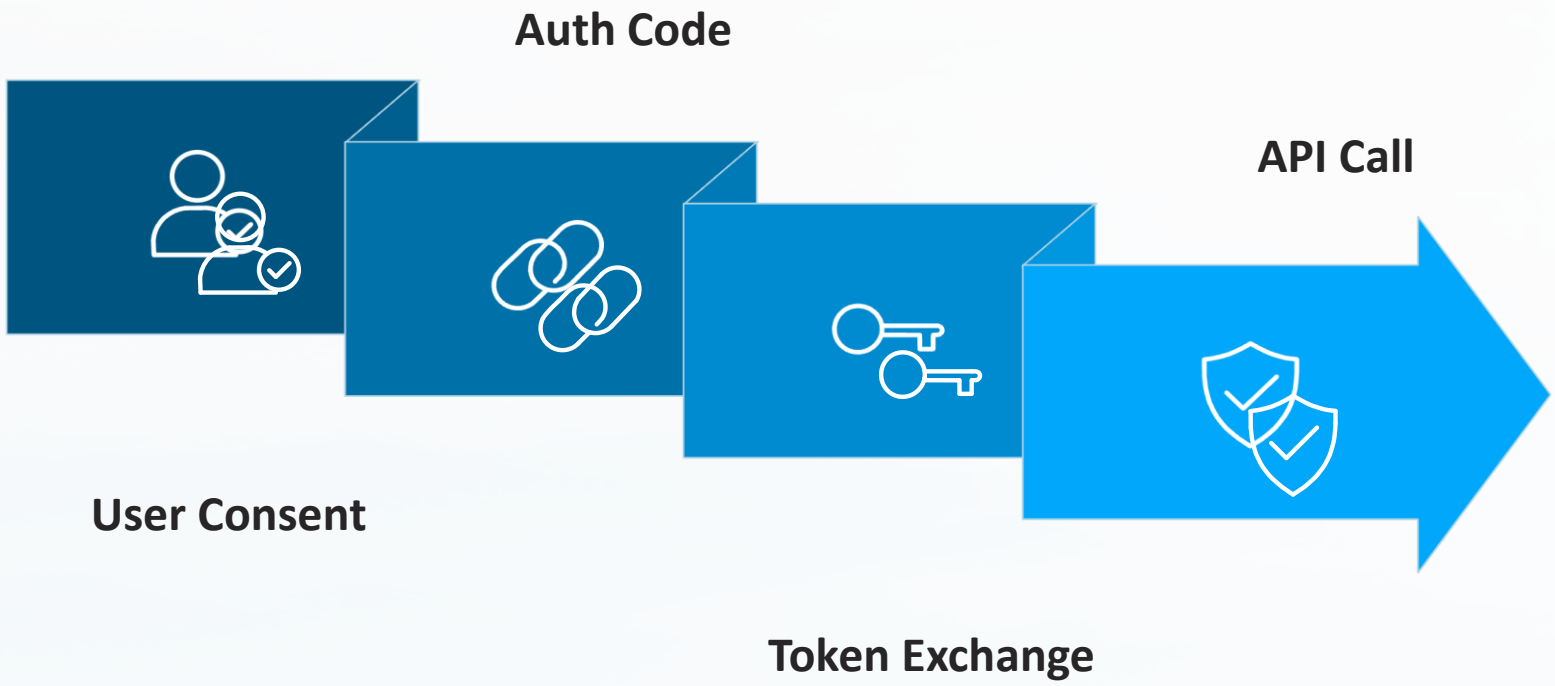
Authorization Code + PKCE

User authentication via auth server—secure for SPAs and mobile apps with PKCE.

Device Code

Suitable for headless or input-limited devices; user authorises on separate device.





This Authorization Code diagram emphasises the separation between the public client browser and the confidential server. Key protections Key protections include PKCE for public clients, using HTTPS for all endpoints, and keeping client secrets on the server. The recommended recommended pattern: browser obtains an authorization code, server redeems code for access and refresh tokens, server stores refresh token stores refresh token securely and uses access token for API calls.

Understanding JSON Web Tokens (JWT): Structure and and Purpose

JWTs are compact, URL-safe tokens encoding claims in JSON: header, payload (claims) and signature. They carry authentication and authorisation data so resource authorisation data so resource servers can make local decisions without round-trips. Use short lifetimes and minimal claims to reduce risk. Typical claims: iss Typical claims: iss (issuer), sub (subject), aud (audience), exp (expiry), iat (issued at), scopes or roles for authorisation.

Header

Specifies algorithm (e.g., RS256) RS256) and token type.

Payload (Claims)

Contains identity, audience, scopes and expiry. Keep minimal minimal and purposeful.

Signature

Verifies integrity and authenticity—signed with private key (asymmetric) or secret (symmetric).

JSON Web Tokens (JWT): Compact, URL-safe Claims

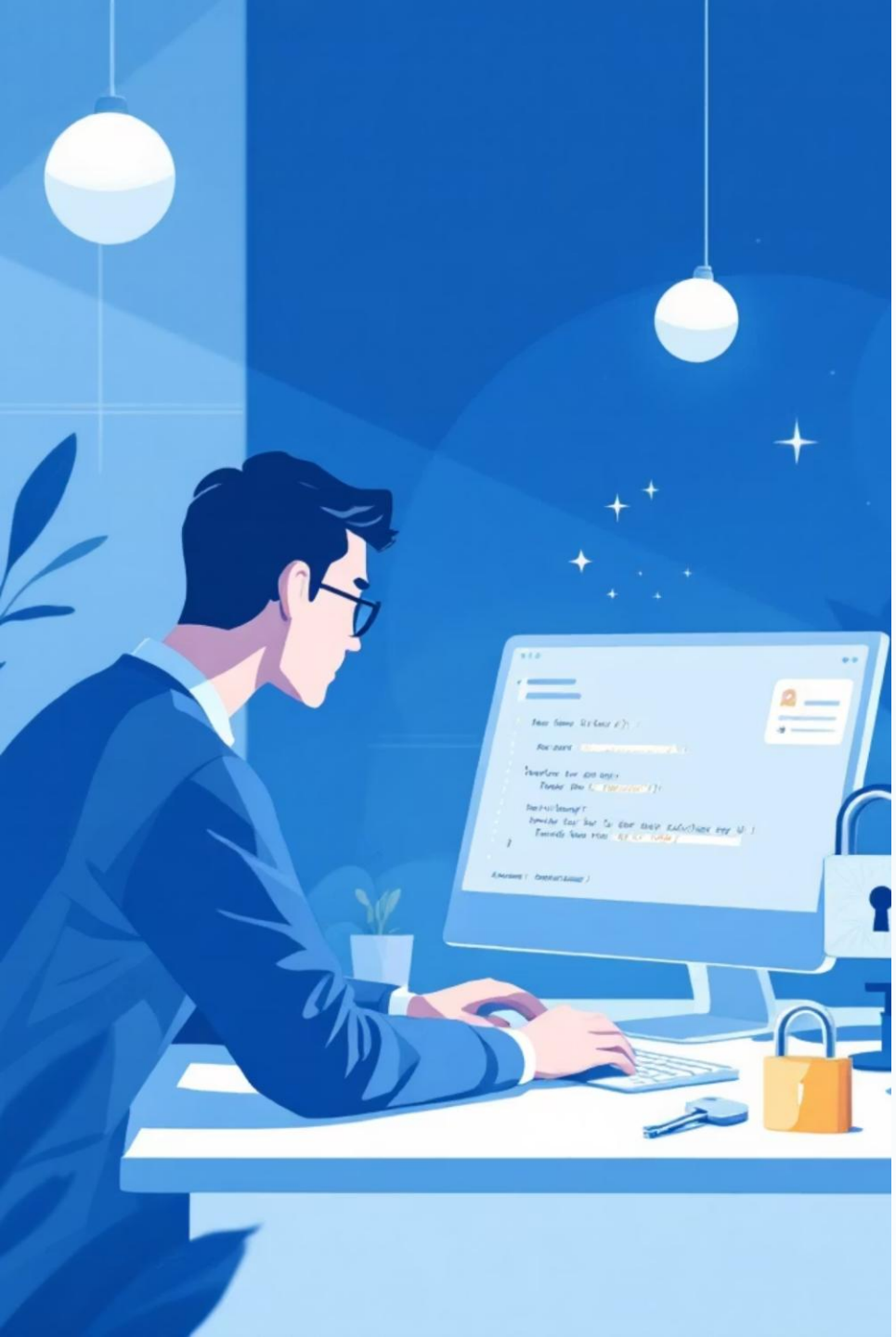
JWTs are compact, URL-safe tokens that represent a set of claims (assertions) about an entity. They are used as access tokens or ID tokens. Advantages: self-contained (carry user/permission data), easy to validate without server-side session storage, interoperable across platforms. Use cases: API access tokens, single sign-on (SSO) ID tokens, and short-lived proof-of-possession patterns.

When to use JWT

Stateless APIs where scalability and decentralised verification matter.

When not to use JWT

If you need frequent revocation or store large session data — consider opaque tokens with central introspection.



JWT Signatures and Verification: Ensuring Data Integrity

Sign JWTs using strong algorithms: prefer asymmetric (RS256 or ES256) for distributed systems—authorization server signs with private key, resource servers verify with public key. Rotate keys using key identifiers (kid) in header and expose JWKS endpoint. Validate exp, nbf, aud, iss, and critical claims. Reject tokens with missing or mismatched claims; log verification failures for incident detection.

Asymmetric Signing

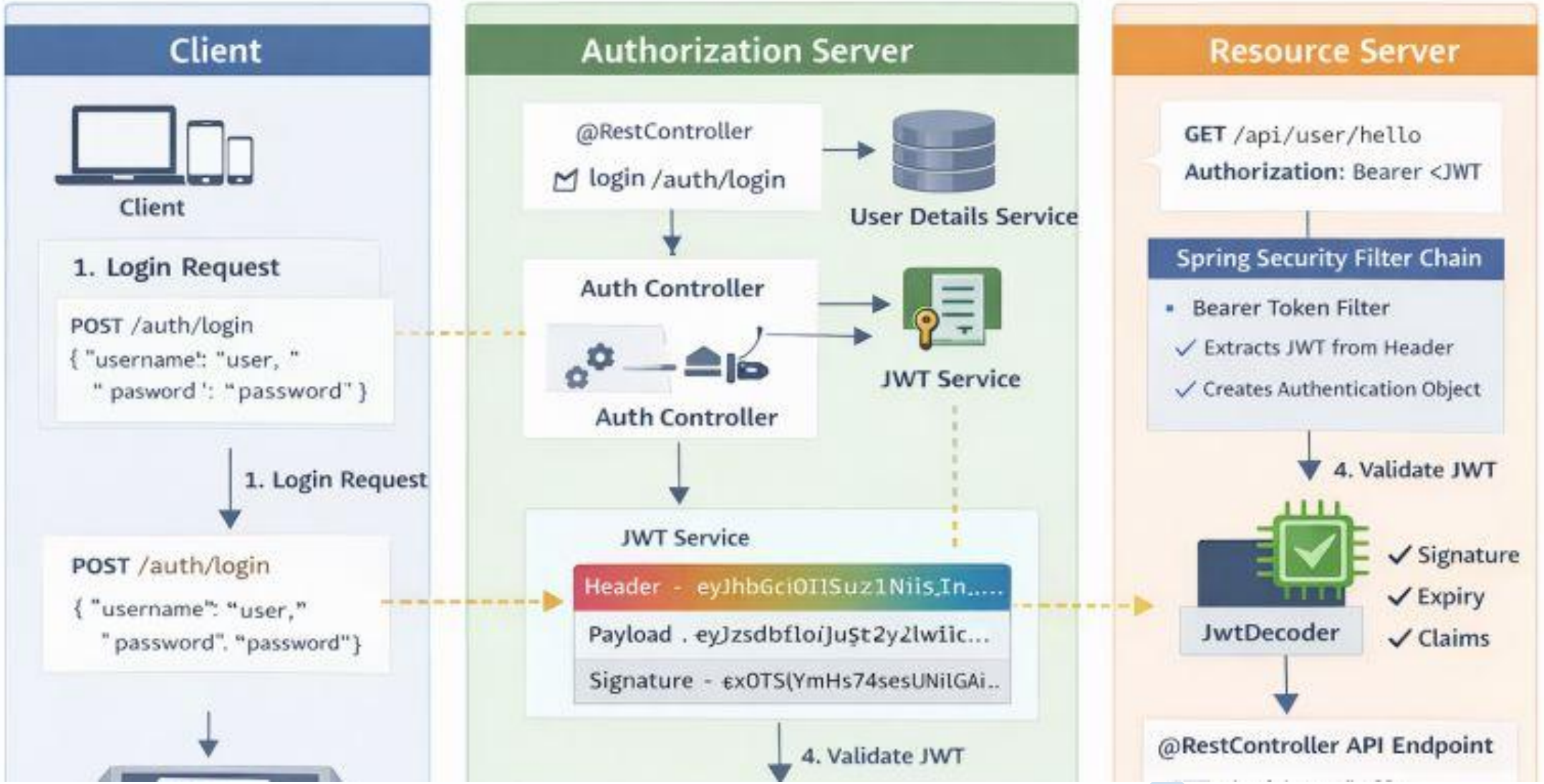
Use RS/ES algorithms and JWKS for safe distribution of verification keys.

Validation Checklist

Validate signature, exp/nbf, aud, iss, and custom security claims.

Key Rotation

Support multiple keys (kid); rotate and deprecate old keys gracefully.


Client
Client
1. Login Request
POST /auth/login
{ "username": "user, "
" pasword ": "password" }
1. Login Request
POST /auth/login
{ "username": "user,"
" password". "password"}
Authorization Server
@RestController
login /auth/login
User Details Service
Auth Controller
JWT Service
Auth Controller
JWT Service
Header - eyJhbGciOIISuz1NiisIn.....
Payload . eyJzsdbfloiJuşt2y2lwiic...
Signature - ex0TS(YmHs74sesUNilGAi..
4. Validate JWT
Resource Server
GET /api/user/hello
Authorization: Bearer <JWT
Spring Security Filter Chain
Bearer Token Filter
Extracts JWT from Header
Creates Authentication Object
4. Validate JWT
Signature
Expiry
Claims
JwtDecoder
@RestController API Endpoint

Integrating OAuth2 and JWT for Microservice Authentication

Combine OAuth2 for token issuance with JWTs as the access token format. Authorization server performs authentication and encodes necessary claims into JWT. Resource servers validate JWT locally to minimise latency. Use audience claim to bind tokens to specific APIs. For extra assurance, validate token at introspection endpoint for opaque tokens or when immediate revocation state is needed.

1. Authenticate & Issue

Client authenticates to Authorization Server; receives JWT access token and optional refresh token.

2. Local Verification

Resource server validates the JWT signature, expiry and audience before granting access.

3. Optional Introspection

Use introspection for opaque tokens or when server must check revocation or active state.

Implementing OAuth2 in Java Microservices

Authorization Server Setup

Setting up a robust authorization server requires careful configuration and security considerations. Spring Authorization Server provides excellent production capabilities for Java-based systems.
Core configuration includes integrating a database for persistent storage, managing clients with secure credential storage, and customizing tokens according to user roles and permissions. Production deployments require high availability configurations, SSL termination with proper certificate management, and comprehensive monitoring for security events.

Spring Security provides comprehensive OAuth2 support that integrates seamlessly with microservices architectures:

```
@Configuration
@EnableWebSecurity
public class ResourceServerConfig {

  @Bean
  public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http.oauth2ResourceServer(oauth2 ->
      oauth2.jwt(jwt ->
        jwt.decoder(jwtDecoder())
          .jwtAuthenticationConverter(jwtConverter())
      )
    );
    return http.build();
  }
}
```

Key implementation points include JWT decoder configuration for proper key resolution, authentication converters mapping JWT claims to Spring Security authorities, and method security using

@PreAuthorize

annotations for fine-grained access control.

- **Implementing API Gateways with OAuth 2. 0 and JWT: A Step-by-Step Guide**
- Securing microservices with an API gateway using OAuth 2. 0 and JWT involves several steps. Here's a general outline:
    - **Choose an API Gateway:** Select an API gateway technology based on your requirements. Popular options include Kong, Tyk, Apigee, AWS API Gateway. Azure API Management.
    - **Set up an Authorization Server:** Implement or integrate with an OAuth 2. 0 authorization server (e. G. , Keycloak, Auth0, Okta). This server will be responsible for issuing access tokens.
    - **Configure the API Gateway:** Configure the API gateway to validate incoming requests against the authorization server. This typically involves configuring a plugin or policy that intercepts requests and verifies the JWT.
    - **Define API Endpoints and Permissions:** Define the API endpoints that your microservices expose and configure the necessary permissions for each endpoint. This might involve mapping roles or scopes from the JWT to specific microservices or resources.
    - **Implement Client Authentication:** Clients need to authenticate with the authorization server to obtain an access token. Implement the appropriate OAuth 2. 0 flow (e. G. , Authorization Code Grant, Client Credentials Grant) based on the client type.
    - **Secure Inter-service Communication:** Consider securing communication between microservices as well, perhaps using mutual TLS (mTLS) or by passing JWTs between services.
    - **Monitor and Log:** Implement robust monitoring and logging to track API usage, identify security threats. Troubleshoot issues.

- Example Kong configuration for JWT validation
- plugins: - name: jwt service: my-service config: key_claim_name: sub Claim containing the user ID secret: your-secret-key Shared secret with the authorization server (HS256) or public key (RS256) algorithms: - HS256

Comparing API Gateway Solutions

- Choosing the right API gateway is crucial for securing your microservices. Here's a comparison of some popular options:
- Consider factors like cost, features, ease of use. Integration with your existing infrastructure when making your decision. If you're heavily invested in AWS, then AWS API Gateway is a natural fit. For more flexibility and open-source options, Kong or Tyk might be better choices.

Feature	Kong	Tyk	Apigee (Google Cloud)	AWS API Gateway	Azure API Management
Open Source	Yes (Core)	Yes (Limited)	No	No	No
Authentication	JWT, OAuth 2. 0, API Key	JWT, OAuth 2. 0, API Key	OAuth 2. 0, API Key, SAML	IAM, Cognito, Custom Authorizers	OAuth 2. 0, API Key, Certificates
Rate Limiting	Yes	Yes	Yes	Yes	Yes
Traffic Management	Yes	Yes	Yes	Yes	Yes
Transformations	Yes	Yes	Yes	Yes	Yes
Monitoring	Via Plugins	Built-in	Built-in	CloudWatch Integration	Azure Monitor Integration
Deployment	On-premise, Cloud	On-premise, Cloud	Cloud	Cloud	Cloud

- **Real-World Use Cases**
- **E-commerce Platform:** An e-commerce platform uses microservices for product catalog, order management. Payment processing. An API gateway secures these services, authenticates users via OAuth 2. 0. Uses JWTs to authorize access to specific resources, like viewing order history or updating profile data.
- **Financial Services:** A bank uses microservices for account management, transaction processing. Fraud detection. The API gateway enforces strict security policies, including multi-factor authentication and rate limiting, to protect sensitive financial data.
- **Healthcare Application:** A healthcare provider uses microservices to manage patient records, appointments. Prescriptions. The API gateway ensures compliance with HIPAA regulations by controlling access to patient data based on user roles and permissions defined in the JWT.

📌 Key takeaways:

- OAuth2 defines who is authorized and how tokens are issued
- JWT is the access token format
- Private key → signs token (Authorization Server only)
- Public key → validates token (Resource Server)
- Fully stateless, no sessions, no DB lookups
- Ideal for microservices and distributed systems

Common Pitfalls and Vulnerabilities to Avoid

Be aware of frequent mistakes that undermine security: accepting tokens without signature signature verification, failing to check exp/aud/iss, storing secrets in client-side code, using code, using Implicit flow for SPAs, and neglecting CSRF protection on redirect endpoints. Also endpoints. Also watch for token leakage via logs, URLs, or referrers. Mitigations include strict include strict validation, secure storage, PKCE, rotating secrets, and implementing revocation revocation and introspection.

Signature & Claim Validation

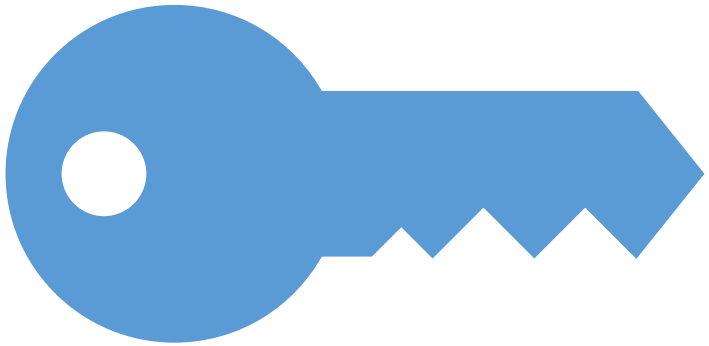
Always verify signature, exp, aud, and iss before trusting a token.

Avoid Token in URLs

Never transmit tokens in query strings; use secure cookies or headers.

Revoke & Rotate

Implement token revocation endpoints and rotate keys/secrets regularly.



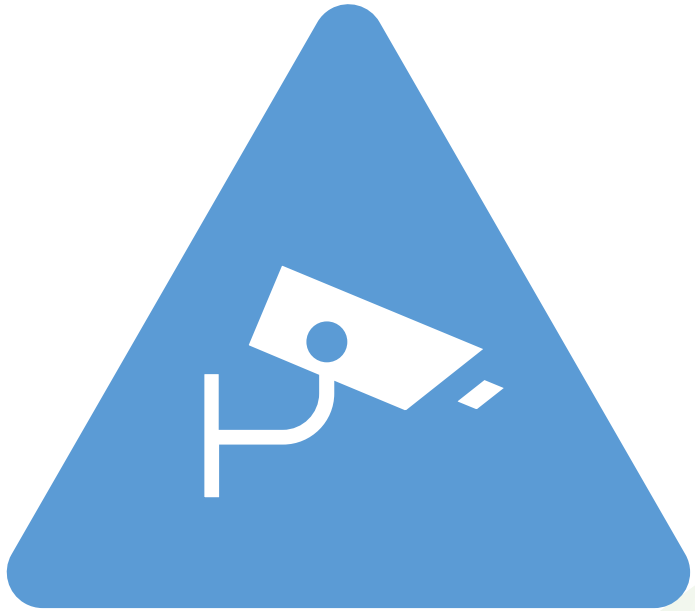
Best Practices for Securing Microservices with API Gateways

- **Use Strong Cryptography:** Employ strong encryption algorithms (e. G. , AES-256) and secure hashing functions (e. G. , SHA-256) for protecting sensitive data.
- **Rotate Keys Regularly:** Regularly rotate your encryption keys and secrets to minimize the impact of a potential key compromise.
- **Implement Least Privilege:** Grant users and applications only the minimum necessary permissions to access the resources they need.
- **Validate Input:** Thoroughly validate all incoming data to prevent injection attacks (e. G. , SQL injection, cross-site scripting).
- **Monitor and Audit:** Continuously monitor your API gateway and microservices for suspicious activity and regularly audit your security configurations.
- **Secure the Authorization Server:** The authorization server is a critical component of your security infrastructure. Harden it against attacks and ensure it is properly configured. Adhering to these best practices will help you build a robust and secure microservices architecture. Remember to regularly update your security measures to stay ahead of emerging threats.
- **Building Authority with Technical Content: A Guide for SEO Success** is essential to keep your organization up to date.

✅ Conclusion:

- Securing microservices with **OAuth2 and JWT in Spring Boot** enables robust, scalable, and stateless authentication. It allows services to independently validate tokens, minimizing coupling and improving scalability.
- 🔑 With OAuth2 and JWT:
- APIs are protected
- Tokens are self-contained and verifiable
- Authorization can be role or scope-based

- We’ve explored the critical role of API Gateways in securing microservices using OAuth 2. 0 and JWT. Looking ahead, the rise of distributed tracing and advanced threat detection systems will further enhance API security. My prediction is that we will see a move towards more context-aware authorization, where access decisions are based not only on user identity but also on real-time risk assessments.



LexisNexis®

Cognizant

THANK YOU

LexisNexis®

Cognizant