Microservice

Authentication

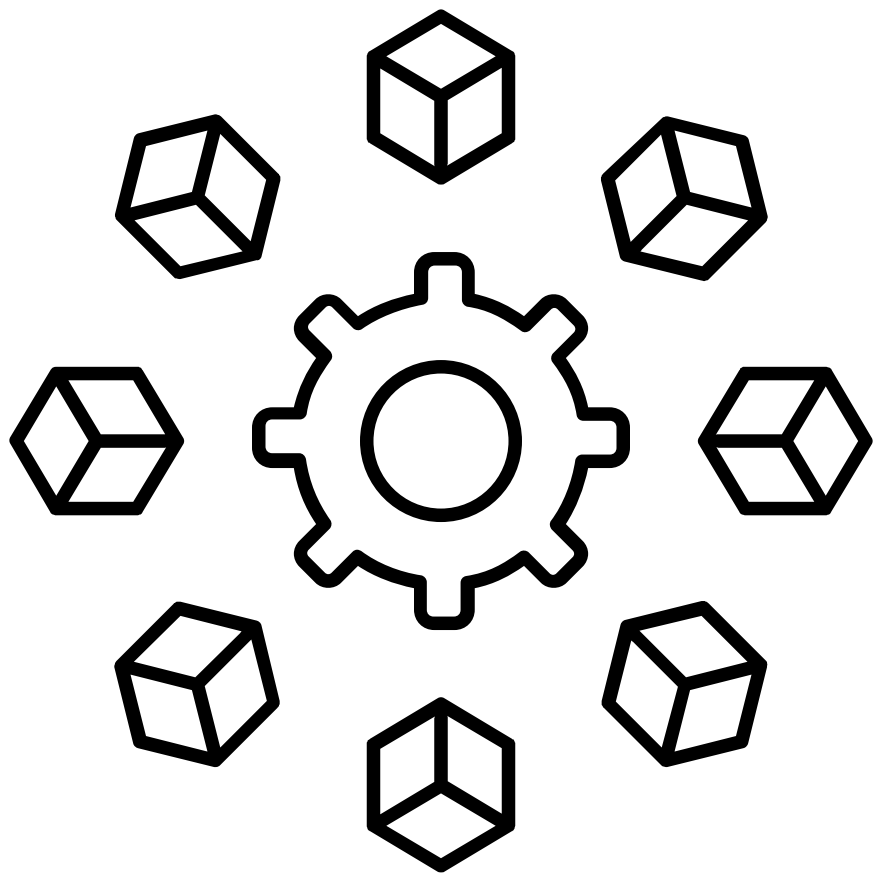
Types

Quick Guide with examples

By SHAILESH SHAKYA @BEGINNERSBLOG.ORG

Why Microservice Authentication Matters?

Imagine each microservice as a room in a house. Authentication is like the locks on those doors. It ensures only authorized "guests" (applications, users, etc.) can access the valuable resources (data, functionalities) each microservice holds.



1. API Keys (Basic Authentication)

How It Works: You assign a unique secret key to each client application. The client includes this key in the Authorization header of every request.

Pros: Simple to implement, easy to understand.

Cons: Offers limited security as the keys are often sent in plain text (use HTTPS to mitigate). Not scalable for large numbers of users.

```
Authorization: Basic <Base64 encoded username:password>
```

2. JSON Web Tokens (JWT)

How It Works: A JWT is a signed token containing claims (user info, permissions) encoded in JSON. The client sends this token with requests.

Pros: Stateless (no server-side session tracking), flexible (can contain custom data), works well in distributed environments.

Cons: Can get large if you include a lot of claims. Requires careful key management to prevent token forgery.

```
Authorization: Bearer <JWT Token>
```

3. OAuth 2.0 and OpenID Connect (OIDC)

How It Works: OAuth focuses on authorization (what you can do), while OIDC extends it to authentication (who you are). Clients get tokens from an authorization server after users log in.

Pros: Industry standard, highly flexible, supports various grant types (authorization code, implicit, client credentials, etc.).

Cons: More complex to set up, but libraries and frameworks can help.

```
Authorization: Bearer <Access Token>
```

4. Mutual TLS (mTLS)

How It Works: Both the client and the server present digital certificates to verify each other's identity.

Pros: Strong security, as the communication is encrypted and the identities are verified. Well-suited for microservice-to-microservice communication.

Cons: Requires certificate management infrastructure.

Practical Examples

API Key for Simple Access:

Let's say you have a microservice that gives weather forecasts. A mobile app developer can use an API key to get access to the service.

See the example code in the next slide

NEXT

Practical Examples

Client-Side (Mobile App - JavaScript Fetch API):

```
Const API_KEY = 'YOUR_API_KEY';
const city = 'New Delhi';

fetch(`https://api.example.com/weather?
city=${city}`, {
  headers: {
    'Authorization': `Basic
${btoa(API_KEY)}` // Base64 encode the
key
  }
})
.then(response => response.json())
.then(data => console.log(data));
```

Practical Examples

JWT for User Sessions:

A JWT can represent a user's logged-in session in an e-commerce app. It allows microservices to validate the user's permissions without making database calls.

See the Example code in the next slide

NEXT

JWT for User Sessions (E-commerce App - Node.js/Express): >> Server-Side

```
Const express = require('express');
const jwt = require('jsonwebtoken');

// ... (other app setup)

app.post('/login', (req, res) => {
  // ... (authenticate user)

  const token = jwt.sign({ userId:
user.id }, 'YOUR_SECRET_KEY');
  res.json({ token });
});

app.get('/products', (req, res) => {
  const token =
req.headers.authorization.split(' ')
[1];

  try {
    jwt.verify(token,
'YOUR_SECRET_KEY');
    // ... (fetch and send products)
  } catch(err) {
    res.sendStatus(401); //
Unauthorized
  }
});
```

Practical Example

OAuth for Third-Party Integrations:

A social media sharing service can use OAuth to allow users to authorize your app to post on their behalf.

See the example code in the next slide

OAuth for Third-Party Integrations (Social Media Sharing - Python/Flask): (Simplified Example - Authorization Code Grant):

```python
from flask import Flask, redirect
from authlib.integrations.flask_client import OAuth

app = Flask(__name__)
oauth = OAuth(app)

oauth.register(
    'social_media_provider',
    client_id='YOUR_CLIENT_ID',
    client_secret='YOUR_CLIENT_SECRET',
    access_token_url='https://provider.com/oauth/token',
    authorize_url='https://provider.com/oauth/authorize',
    api_base_url='https://provider.com/api',
    client_kwargs={'scope': 'post'}
)

@app.route('/login')
def login():
    redirect_uri = 'http://your-app.com/callback'
    return oauth.social_media_provider.authorize_redirect(redirect_uri)

# ... (Handle the callback to get the access token and use it to make API calls)
```

Practical Example

mTLS for Secure Service Communication:

Microservices that exchange sensitive financial data can use mTLS to ensure that the data is not intercepted or tampered with.

See the Example code in the next slide

NEXT

mTLS for Secure Service Communication (Financial Microservices - Nginx Configuration Example):

```
server
{ listen 443 ssl; server_name  financial-service.example.com;

ssl_certificate /path/to/server.crt; ssl_certificate_key /path/to/server.key;

 ssl_client_certificate /path/to/client-ca.crt;

 ssl_verify_client on;

location / { proxy_pass http://financial-service-backend; } }
```

Key Considerations

- Security First: Always prioritize security in your design. Implement input validation, HTTPS, and best practices for your chosen authentication method.
- Token Management: Store and transmit tokens securely. Consider token expiration and revocation mechanisms.
- Error Handling: Design for graceful error handling and provide helpful feedback to clients when authentication fails.
- Documentation: Thoroughly document your authentication scheme to make integration easier for other developers.

Libraries and Frameworks

Many tools can streamline authentication in microservices:

- Spring Security: Widely used Java framework for securing web applications and microservices.
- Keycloak: Open-source identity and access management solution that supports OAuth, OIDC, and more.
- Auth0: Cloud-based authentication and authorization platform.

Created by Shailesh Shakya

@BEGINNERSBLOG.ORG

Did you find this post helpful?
Please...