# Observability

| **Component** | **Role** | **Description** |
| --- | --- | --- |
| 🟩 **API / Microservice** | Source | Emits logs, traces, and metrics via OpenTelemetry SDK or Spring Actuator |
| 🟦 **OpenTelemetry Collector** | Collector | Collects telemetry, processes, and exports to backends |
| 🟨 **Prometheus** | Metrics DB | Scrapes and stores time-series metrics for performance analysis |
| 🟪 **Tempo** | Trace Storage | Stores and queries distributed traces |
| 🟥 **Jaeger** | Trace Viewer | UI for visualizing spans, dependencies, and performance bottlenecks |
| 🟧 **Grafana** | Dashboard | Unifies metrics (Prometheus) and traces (Tempo/Jaeger) for visualization |
| 👩‍💻 **User** | Viewer | Monitors health, latency, and bottlenecks through dashboards |

**🟦 OpenTelemetry Collector**

**What it does:**  
Acts as a **data router and processor** for all observability data — metrics, logs, and traces.

**Usage:**

* You configure your Spring Boot app with opentelemetry-exporter-otlp to send telemetry data (metrics/traces/logs) to the **Collector**.
* The collector then forwards:
  + **Metrics → Prometheus**
  + **Traces → Tempo/Jaeger**
  + **Logs → Loki (if you had it)**

**Think of it as:**  
A middleman that normalizes and routes your app’s monitoring data.

**How to check:**  
Go to your Collector logs — you should see messages like  
"Exporting metrics to prometheus" or "Exporting traces to tempo".

**🟨 Prometheus**

**What it does:**  
Stores **metrics** (numbers over time like CPU usage, memory, requests/sec).

**Usage:**

* Prometheus **scrapes metrics** from your app (or from the OpenTelemetry Collector).
* Metrics endpoint: http://<your-app>:8080/actuator/prometheus
* You can query metrics using **PromQL** (Prometheus Query Language).

**Try it:**  
Visit http://<prometheus-host>:9090 → Go to “Graph” → try queries like:

http\_server\_requests\_seconds\_count

system\_cpu\_usage

jvm\_memory\_used\_bytes

Prometheus must be connected to other microservice via actuator endpoint to pull this information this is defined in config.xml

global:  
 scrape\_interval: 5s  
 evaluation\_interval: 5s  
  
rule\_files:  
 - /etc/prometheus/alerts/prometheus-alerts.yml  
  
scrape\_configs:  
 - job\_name: 'order-service'  
 metrics\_path: /actuator/prometheus  
 static\_configs:  
 - targets: ['host.docker.internal:8080','order-service:8080']  
 - job\_name: 'customer-service'  
 metrics\_path: /actuator/prometheus  
 static\_configs:  
 - targets: ['host.docker.internal:8081','customer-service:8081']  
 - job\_name: 'gateway-service'  
 metrics\_path: /actuator/prometheus  
 static\_configs:  
 - targets: ['host.docker.internal:8085','gateway-service:8085']  
 - job\_name: 'discovery-service'  
 metrics\_path: /actuator/prometheus  
 static\_configs:  
 - targets: ['host.docker.internal:8761','discovery-service:8761']  
 - job\_name: 'config-service'  
 metrics\_path: /actuator/prometheus  
 static\_configs:  
 - targets: ['host.docker.internal:8888','config-service:8888']

And can be seen through targets page

A screenshot of a computer

AI-generated content may be incorrect.

**🟪 Tempo**

**What it does:**  
Stores **distributed traces** (used to follow a request through multiple services).

**Usage:**

* Traces come from OpenTelemetry (via Collector).
* Each trace shows how a request flows between services and how long each span took.

**Think of it as:**  
A **timeline** of your request journey across microservices.

**How to view:**

* Usually viewed **through Grafana**, not directly.
* In Grafana → add Tempo as a data source → open “Explore” → select Tempo → search for traces.

**🟥 Jaeger**

**What it does:**  
Also stores **traces**, like Tempo.  
You can use **either Tempo or Jaeger** (you have both, so maybe Tempo is backend and Jaeger is UI).

**Usage:**

* OpenTelemetry Collector can export traces to Jaeger.
* Each trace shows service name, operation, and duration.

**Try it:**  
Visit http://<jaeger-host>:16686 → Search by serviceName (e.g., spring-boot-app).

**In short:**  
Jaeger gives a **UI to visualize trace spans** (how long each call took, parent-child relationships, etc.).

**🟧 Grafana**

**What it does:**  
Acts as the **visualization and correlation hub**.  
You can connect all others (Prometheus, Tempo, Jaeger, Loki) to it.

**Usage:**

* Add **Prometheus**, **Tempo**, and **Jaeger** as data sources.
* Create dashboards for:
  + Metrics (CPU, JVM, requests/sec)
  + Traces (via Tempo)
  + Logs (if you add Loki later)
* Use **“Explore”** to correlate metrics ↔ traces ↔ logs.

**Try it:**  
Visit http://<grafana-host>:3000  
→ Log in (default admin/admin)  
→ Add data sources  
→ Import dashboard ID 4701 (Spring Boot Micrometer) from Grafana.com

**🧩 Quick Recap Table**

| **Tool** | **Type** | **Example UI** | **What You Get** |
| --- | --- | --- | --- |
| 🟦 OpenTelemetry Collector | Pipeline | — | Routes metrics/traces/logs |
| 🟨 Prometheus | Metrics DB | :9090 | CPU, JVM, request rates |
| 🟪 Tempo | Traces backend | via Grafana | Distributed traces |
| 🟥 Jaeger | Trace viewer | :16686 | Trace visualization |
| 🟧 Grafana | Dashboard/Correlation | :3000 | Unified dashboards |

# Spring

Which class to make as componenet which one not

| **Method** | **Depends on Config?** | **Stateful?** | **Should be Static?** | **Reason** |
| --- | --- | --- | --- | --- |
| generateAccessToken | ✅ Yes | ✅ | ❌ | Needs key + expiration |
| generateRefreshToken | ✅ Yes | ✅ | ❌ | Needs key + expiration |
| validateToken | ✅ Yes | ✅ | ❌ | Uses secret key |
| constantTimeEquals | ❌ No | ❌ | ✅ | Pure utility |
| hashPassword | ❌ No | ❌ | ✅ | Pure crypto function |

# Security

## CORS

How CORS work? Is OPTIONS sending some cookie or something which is getting sent in POST call?

✅ **No, OPTIONS never sends cookies or credentials.**  
It’s a *pre-check*. When the server’s OPTIONS response includes:

Access-Control-Allow-Credentials: true

Access-Control-Allow-Origin: <your-origin>

then the browser *decides* it’s safe to include credentials in the next (actual) request. So, when you set: you’re not *sending* credentials in the preflight you’re telling the browser **it’s allowed to include them later**.

corsConfig.setAllowCredentials(true);

The **browser**, not your code, decides when to send credentials (cookies or auth headers). The **preflight (OPTIONS)** request itself never includes credentials — but its *result* determines whether the *actual request* will include them.

**🧠 What happens step by step**

Let’s say you make this Angular call:

this.http.post('http://localhost:8085/customers/login', body, {

withCredentials: true

});

1. **Browser sends a preflight OPTIONS request:**

OPTIONS /customers/login HTTP/1.1

Origin: http://localhost:4200

Access-Control-Request-Method: POST

Access-Control-Request-Headers: content-type

➡️ This does NOT include cookies or Authorization headers.It’s a simple request to ask: “Hey server, can I make a POST request from this origin and include credentials?”

1. **Server (Gateway) responds:**

HTTP/1.1 200 OK

Access-Control-Allow-Origin: http://localhost:4200

Access-Control-Allow-Methods: GET,POST,PUT,DELETE,OPTIONS

Access-Control-Allow-Headers: Authorization, Content-Type

Access-Control-Allow-Credentials: true

Access-Control-Max-Age: 3600

✅ This tells the browser: “Yes, you (the origin http://localhost:4200) may send a POST request and include credentials.”

1. **Browser sends the actual POST:**

POST /customers/login HTTP/1.1

Origin: http://localhost:4200

Cookie: JSESSIONID=XYZ123 ← or any cookies for this domain

Authorization: Bearer abcdef ← if your JS adds this header

Content-Type: application/json

✅ Only **now** are credentials actually sent (because of withCredentials: true and because the server allowed it).

1. **Server responds**

HTTP/1.1 200 OK

Access-Control-Allow-Origin: http://localhost:4200

Access-Control-Allow-Credentials: true

Set-Cookie: JSESSIONID=XYZ123; Path=/; HttpOnly

The browser accepts this response and stores cookies, if any.

**🔬 When credentials get included**

Browser includes cookies or Authorization headers only if:

1. **Frontend request** sets withCredentials: true (Angular/Fetch)

login(email*:* string, password*:* string)*:* Observable<{ success*:* boolean; message*:* string }> {  
 *return* this.http.post<{ success*:* boolean; message*:* string }>(  
 `${BASE}/customers/login`,  
 { email, password },  
 {  
 headers: this.jsonHeaders,  
 withCredentials: true,  
 }  
 );  
}

1. **Server’s CORS response** includes:

*@Bean  
public* CorsConfigurationSource corsConfigurationSource() {  
 CorsConfiguration corsConfig = *new* CorsConfiguration();  
 corsConfig.setAllowedOrigins(List.of("http://localhost:4200"));  
 corsConfig.setAllowedMethods(List.of("GET", "POST", "PUT", "DELETE", "OPTIONS"));  
 corsConfig.setAllowedHeaders(List.of("Authorization", "Content-Type", "X-Requested-With"));  
 corsConfig.setExposedHeaders(List.of("Authorization"));  
 corsConfig.setAllowCredentials(*true*);  
 corsConfig.setMaxAge(3600L);  
  
 UrlBasedCorsConfigurationSource source = *new* UrlBasedCorsConfigurationSource();  
 source.registerCorsConfiguration("/\*\*", corsConfig);  
 *return* source;  
}

* Access-Control-Allow-Credentials: true
* Access-Control-Allow-Origin: http://your-ui-domain

1. **AllowedOrigin is not "\*"** — wildcard is forbidden with credentials.

**⚠️ Why disabling setAllowCredentials breaks it**

When allowCredentials = false, browser sees:

Access-Control-Allow-Credentials: false

→ It won’t send any cookies or auth headers even if you ask it to.  
→ So your POST /login request lacks session cookies or Authorization header → backend rejects it → looks like a CORS failure (but it’s actually blocked credential use).

**✅ Summary**

| **Step** | **Includes Cookies/Auth?** | **Notes** |
| --- | --- | --- |
| Preflight (OPTIONS) | ❌ No | Browser checks permission only |
| Actual Request (POST/GET) | ✅ Yes (if allowed) | Browser includes cookies/headers |
| Response | ✅ May set cookies | Must echo Access-Control-Allow-Credentials: true |

## Returning sensitive info

Is it okay to set JWT token in the request body? How does it compare to setting the token in response header, or in the cookie, or any other way?

* **⚙️ 1️⃣ Option A — Return JWT in response body (your current approach)**

Example:

{

"success": true,

"token": "eyJhbGciOiJIUzI1NiIs..."

}

**✅ Pros:**

* + **Simple** — Easy for SPA or mobile clients to parse.
  + **Stateless** — No need for cookies or session.
  + **Flexible** — Works across different frontends (web, mobile, APIs).

**⚠️ Cons:**

* + If the client stores it in **localStorage** or **sessionStorage**, it’s **vulnerable to XSS** (cross-site scripting).
  + Requires custom code in Angular to attach it to headers (Authorization: Bearer ...).
  + Token is not sent automatically — each request must explicitly add it.

**💡 When to use:**

* + SPAs or native apps where you control the JS.
  + You’ve hardened the app against XSS (e.g., strict CSP, sanitization).
  + You prefer manual token handling (you already wrote an interceptor — perfect).

✅ **This is currently the most common pattern in SPAs (Angular, React, Vue).**

* **⚙️ 2️⃣ Option B — Return JWT in response header**

Example response:

HTTP/1.1 200 OK

Authorization: Bearer eyJhbGciOiJIUzI1NiIs...

**✅ Pros:**

* + - Avoids exposing the token inside response body (less likely to be logged accidentally).
    - Useful when the frontend framework automatically reads headers.

**⚠️ Cons:**

* + - Browsers block JS from reading Authorization header **unless you explicitly expose it in CORS**:  
      You must add
    - corsConfig.setExposedHeaders(List.of("Authorization"));
    - Still must store token in localStorage/sessionStorage or memory.
    - Doesn’t improve XSS safety.

**💡 When to use:**

* + - API-to-API login flows (machine clients).
    - Microservice login (non-browser consumers).

✅ It’s slightly cleaner, but for browser apps it adds CORS complexity.

* **⚙️ 3️⃣ Option C — Return JWT in HttpOnly cookie**

Example:

Set-Cookie: jwt=eyJhbGciOiJIUzI1NiIs...; Path=/; HttpOnly; Secure; SameSite=Strict

**✅ Pros:**

* + - **Most secure** for browser clients.
      * HttpOnly → JS cannot access it (protected from XSS).
      * Secure + SameSite → mitigates CSRF attacks.
    - Token automatically sent with every request (no need to attach headers).

**⚠️ Cons:**

* + - Requires **CORS with credentials**:
      * Access-Control-Allow-Credentials: true
      * Access-Control-Allow-Origin: https://your-ui.com (cannot use \*)
    - More complex in distributed microservices (cookie domain scoping).
    - Refresh tokens must also be handled carefully (cookie expiration).

**💡 When to use:**

* + - Public web apps (user login sessions).
    - Apps where **maximum browser security** is desired (like banking portals).

✅ **This is the enterprise-recommended pattern for web apps with a backend login flow.**

* **⚙️ 4️⃣ Option D — JWT in both Access Token (header) and Refresh Token (cookie)**

Example:

* + - Access token returned in body or header (short-lived)
    - Refresh token stored as HttpOnly cookie (long-lived)

Then when the access token expires:

* + - Frontend calls /refresh-token
    - Browser automatically sends the cookie
    - Server verifies it and issues a new short-lived JWT.

**✅ Pros:**

* + - Perfect balance between security and usability.
    - Refresh token hidden from JS (safe from XSS).
    - Short-lived access token limits exposure window.

**⚠️ Cons:**

* + - Slightly more complex to implement.
    - Requires managing token rotation and revocation.

✅ **This is what enterprise IdPs (like Keycloak, Auth0, Okta) do internally.**

**🧩 5️⃣ TL;DR — Summary Table**

| **Delivery Location** | **Pros** | **Cons** | **Ideal For** |
| --- | --- | --- | --- |
| **Response Body** | Simple, easy to parse | Exposed to JS (XSS risk) | Learning, internal SPAs |
| **Response Header** | Clean separation, no body changes | CORS config needed | API-based clients |
| **HttpOnly Cookie** | Secure, automatic | CORS+cookie complexity | Enterprise browser apps |
| **Header + Cookie Split** | Most secure, refreshable | Complex | Enterprise production setups |

## Secrets Key Management

**🧩 1️⃣ Why key management is a big deal**

In enterprise systems, a JWT secret key is a **sensitive cryptographic material**:

* It can **sign** tokens (→ impersonate users if leaked)
* It can **verify** tokens (→ trust boundary across services)
* It should be **rotatable, auditable, and access-controlled**

So, it must never be:

* hardcoded in source code ❌
* stored in version control ❌
* emailed or shared manually ❌

Instead, it’s managed through **secret management systems** that enforce access control, versioning, rotation, and auditing.

**🏗️ 2️⃣ Common approaches — summary table**

| **Environment** | **How keys are stored** | **Who manages** | **How applications access** | **Rotation method** |
| --- | --- | --- | --- | --- |
| **Cloud (AWS / GCP / Azure)** | Secret Manager / Parameter Store / KMS | Cloud security / DevOps | SDK or injected env var | Automated or scheduled |
| **On-prem / In-house servers** | Vault (HashiCorp), file store, or env vars | Infra / Ops team | Mounted volume / env var | Manual or scripted |
| **CI/CD (Jenkins, GitLab CI, Argo)** | Secret plugin / Vault integration | DevSecOps | Injected during build/deploy | Automated with pipeline |
| **Containers / Kubernetes** | Kubernetes Secrets or sealed secrets | Platform team | Mounted or env var in Pod | Rotated via re-deploy |
| **Local dev** | .env file or IDE secret store | Developer | Local env var | Manual reset |

**☁️ 3️⃣ Cloud-native key management (best practice)**

**Example: AWS**

**🔐 Secret storage**

* Store JWT secret in **AWS Secrets Manager** or **SSM Parameter Store** (encrypted with AWS KMS key).
* IAM policies define **which services can access** that secret.

**🏃 Retrieval (runtime)**

* Spring Boot apps use:
  + AWS SDK to load the secret programmatically, or
  + Secrets pulled into env vars at deployment time (via ECS, Lambda, or EKS config).

jwt:

secret: ${JWT\_SECRET}

Then ECS or EKS injects that env var from Secrets Manager:

aws secretsmanager get-secret-value --secret-id jwt-secret

✅ Benefits:

* Rotation can be scheduled (AWS can auto-rotate)
* Fine-grained IAM access
* Centralized audit trail

**Example: GCP**

* Store in **Google Secret Manager**
* Grant access to only the service account of your app
* Inject via environment variable or read at runtime using Google SDK
* Optional automatic rotation policies

**Example: Azure**

* Store in **Azure Key Vault**
* Access controlled by **Managed Identities**
* Retrieved at app startup via Spring Cloud Azure or SDK

**🏢 4️⃣ On-prem / in-house environments**

**Option 1: HashiCorp Vault (de facto standard)**

* Vault centrally stores all secrets (JWT keys, DB creds, API keys)
* Access is via API or short-lived tokens
* Can auto-generate and rotate keys periodically

Typical flow:

1. Infra or Security team generates keys using OpenSSL or Vault’s built-in generator.
2. Key is stored in Vault under a namespace like secret/app/jwt-key.
3. Each service fetches it at startup (via Vault agent or Spring Vault).

spring:

cloud:

vault:

host: vault.company.local

scheme: https

authentication: token

application-name: jwt-service

✅ Vault handles:

* Encryption at rest
* Audit logs
* Dynamic key rotation
* Policy-based access

**Option 2: Env vars + file-based secrets**

For smaller on-prem setups:

* Keys stored in encrypted files on secure servers (/etc/secrets/jwt.key)
* Mounted as read-only volumes into containers
* Decrypted at runtime or pre-startup
* Managed by Infra team (restricted permissions)

Example (systemd service):

EnvironmentFile=/etc/myapp/secrets.env

**🔄 5️⃣ Key rotation (enterprise must-have)**

Rotation means periodically changing the key used to sign tokens.

**How it’s done:**

1. **Dual-key strategy**
   1. Maintain **current key (K₁)** and **previous key (K₀)**
   2. New tokens signed with K₁
   3. Old tokens verified with both K₀ and K₁
   4. After TTL expires → retire K₀
2. **Stored with versioning**
   1. Vault / Secret Manager stores secrets with versions
   2. Your app can be configured to fetch latest key version
3. **Rolling deployment**
   1. CI/CD pipeline updates key in Vault → triggers redeploys
   2. New pods fetch new key automatically

**⚙️ 6️⃣ Example: Cloud enterprise setup (JWT signing key)**

**Diagram (conceptually):**

┌──────────────────────────┐

│ AWS Secrets Manager │

│ secret: jwt-key │

│ version: v2 │

└──────────┬───────────────┘

│ (secure fetch via IAM role)

▼

┌──────────────────────────┐

│ Spring Boot Microservice│

│ (Customer Service) │

│ loads key via AWS SDK │

│ sign JWT using HS256 │

└──────────┬───────────────┘

│

▼

┌──────────────────────────┐

│ API Gateway │

│ loads same key via IAM │

│ validates JWT signature │

└──────────────────────────┘

✅ Keys never touch the file system  
✅ No developer ever sees the secret  
✅ Full audit trail maintained

**🧰 7️⃣ Jenkins / CI/CD integration**

In CI/CD:

* Secrets (JWT keys, DB passwords, etc.) are not stored in Jenkins itself.
* Jenkins connects to Vault or cloud secret store and **fetches secrets at build or deploy time**.
* Injected as environment variables or Kubernetes Secrets for deployment.

environment {

JWT\_SECRET = credentials('vault-jwt-key')

}

✅ Jenkins never logs the secret (redacted)  
✅ Secrets rotate without pipeline change

**🧾 8️⃣ Typical enterprise lifecycle**

| **Phase** | **Action** | **Who owns it** |
| --- | --- | --- |
| **Initial generation** | openssl rand -base64 32 or via Vault API | Security / Infra |
| **Storage** | Secret Manager / Vault / Key Vault | Infra / DevSecOps |
| **Access setup** | IAM roles / Vault policies | Security team |
| **Application config** | Refer to env var or secret name | Developer |
| **Rotation** | Automated every 90 days | Security automation |
| **Audit & alerts** | Monitor access patterns | SOC / Compliance |

**🔒 9️⃣ TL;DR — The enterprise-grade pattern**

| **Area** | **Best Practice** |
| --- | --- |
| Key source | Generated via Vault or OpenSSL |
| Storage | Cloud Secret Manager / Vault |
| Access | IAM roles or Vault tokens |
| Injection | Environment variable or runtime fetch |
| Rotation | Automated (90 days or less) |
| Audit | Enabled via secret manager logs |
| No hardcoding | ✅ Never in code or repo |
| Automation | CI/CD handles injection & redeploy |

**✅ Example (Cloud-native summary)**

| **Component** | **Where the key lives** | **How it’s accessed** |
| --- | --- | --- |
| Customer Service | AWS Secrets Manager: jwt-sign-key | IAM Role fetches at startup |
| Gateway | Same secret ID | IAM Role fetches at startup |
| Jenkins | Injects secret via Vault plugin | Vault token |
| Local dev | .env file or dummy key | Manual |

### ****Base64****, ****Hex****, and ****HMAC-SHA256 (HS256)****

**🧠 Core Concepts Summary**

**1️⃣ HMAC-SHA256 (HS256)**

* It’s the algorithm used to **sign and verify** JWT tokens.
* **HMAC** = Hash-based Message Authentication Code.
* **SHA-256** = Secure Hash Algorithm, 256-bit digest.

🧩 **HS256 requires a secret key that is at least 256 bits (32 bytes).**

* The key is used both for:
  + Signing the token (in Customer Service)
  + Verifying the token (in Gateway)

⚠️ If even one byte of the key differs → signature mismatch.

**2️⃣ Byte length & cryptographic strength**

| **Algorithm** | **Required key length** | **In bytes** | **In bits** |
| --- | --- | --- | --- |
| HS256 | 32 | 32 bytes | 256 bits |
| HS384 | 48 | 48 bytes | 384 bits |
| HS512 | 64 | 64 bytes | 512 bits |

💡 Using shorter keys weakens security;  
longer keys are allowed but will be truncated.

**3️⃣ Hexadecimal vs Base64**

| **Concept** | **Purpose** | **Valid Characters** | **Example** | **Decodes to** |
| --- | --- | --- | --- | --- |
| **Hexadecimal (Hex)** | Human-readable form of binary | 0–9, A–F | 0C8FAB1234... | Each **2 chars = 1 byte** |
| **Base64** | Text-safe encoding of binary | A–Z, a–z, 0–9, +, /, = | u3YzB2+n7zDGMB7... | Each **4 chars = 3 bytes** |

🧩 **Important difference:**  
Base64 is safe for storage in text configs;  
Hex is not interchangeable with Base64.

If you take a **Hex string** and try to **Base64-decode** it,  
you get **garbage bytes** → signature mismatch.

**4️⃣ Your Case**

You had this key:

0CBC458ACABBE3F02EA737C7DF416FA216FBA9B287932C38AB8AC61A69671CE4

This looks random, but it’s **Hexadecimal**, not Base64.

* ❌ Base64.getDecoder().decode(secret) → interprets it wrongly.
* ✅ secret.getBytes(StandardCharsets.UTF\_8) → works (same visible text bytes).
* ✅ (Better) Generate a **real Base64 key**.

**5️⃣ Why openssl rand -base64 32 matters**

This command:

openssl rand -base64 32

➡️ Generates **32 cryptographically secure random bytes**  
➡️ Then encodes them in **Base64** (text-safe format).

Example output:

u3YzB2+n7zDGMB7PlA1QXH0Wq1e67soxhOVN6YjNrcE=

✅ Perfect for HS256:

* 32 bytes (256 bits)
* Secure randomness
* Easy to store and decode in Java:
* Keys.hmacShaKeyFor(Base64.getDecoder().decode(secret));

**6️⃣ In your system now**

| **Service** | **Old behavior** | **Correct fix** |
| --- | --- | --- |
| Customer Service | Base64-decoded a non-Base64 hex string → ❌ | Use UTF-8 bytes directly for now |
| Gateway | Used UTF-8 bytes of same string → ✅ | Keep consistent |
| Long-term (secure) | Use openssl rand -base64 32 and decode properly | ✅ Recommended |

**7️⃣ End Result**

✅ Matching secret key bytes → JWT signatures verify successfully  
✅ No more “Invalid JWT signature” errors  
✅ Security-grade randomness ready for production later

### Key Rotation invalidate JWT

**🧩 1️⃣ JWTs are stateless by design**

JWT (JSON Web Token) = **self-contained credential**. It carries all info (claims, expiration, signature) and doesn’t need to be stored on the server.

✅ **Advantage:** No database lookup → fast, scalable  
❌ **Disadvantage:** Once issued, can’t be revoked unless you track it somewhere

This design directly impacts both key rotation and user-specific invalidation.

**🔐 2️⃣ Key Rotation — What it is**

“Key rotation” = replacing the cryptographic key used to sign/verify tokens. Typical reasons:

* Regular rotation policy (e.g., every 90 days)
* Compromised key
* Moving to stronger algorithms
* Security compliance (SOC2, ISO, NIST)

**🧠 3️⃣ What happens during key rotation**

If you simply **replace** the secret key and restart services:

* All existing JWTs (signed with the old key) will fail verification.
* Every user will get 401 Unauthorized → effectively **force-logged out**.

That’s the default behavior of naïve key rotation.

**✅ 4️⃣ Enterprise-grade key rotation: dual-key strategy**

To prevent global logout during rotation, enterprises use a **“Key Versioning”** or **“Key ID (kid)”** strategy.

**🔸 Step 1 — Maintain multiple active keys**

Example:

| **Key ID** | **Secret** | **Status** |
| --- | --- | --- |
| kid\_1 | old key (Base64) | ✅ still valid |
| kid\_2 | new key (Base64) | 🆕 used for signing |

JWT header contains a kid (Key ID) claim:

{

"alg": "HS256",

"typ": "JWT",

"kid": "kid\_1"

}

**🔸 Step 2 — During rotation**

* New tokens are **signed** using kid\_2
* Gateway verifies against **both keys** (kid\_1 and kid\_2)
* Old tokens remain valid until expiration
* After TTL passes → kid\_1 is retired

✅ No mass logout  
✅ Gradual rotation  
✅ Predictable expiration cutoff

**🔸 Step 3 — Implementation pattern**

In your config:

jwt:

active-key-id: kid\_2

keys:

kid\_1: Base64OldKey==

kid\_2: Base64NewKey==

When signing:

String kid = "kid\_2";

return Jwts.builder()

.setHeaderParam("kid", kid)

.setSubject(userId)

.signWith(keys.get(kid), SignatureAlgorithm.HS256)

.compact();

When verifying:

String kid = jwtHeader.get("kid", String.class);

Key key = keys.get(kid);

Jwts.parserBuilder().setSigningKey(key).build().parseClaimsJws(token);

✅ Result: Users continue working until their old JWT expires naturally.

**🕒 5️⃣ Token expiration (TTL) is part of rotation safety**

Enterprise systems always issue **short-lived access tokens** (e.g., 15 min) and use **refresh tokens** (valid for days/weeks). That way:

* You only need to keep the old signing key alive for ~15 minutes.
* Within that window, all old tokens will expire naturally.

No visible user logout during key rotation.

**🚫 6️⃣ How to invalidate JWTs for a specific user**

Since JWTs are stateless, you can’t “pull back” a token from a client. But you can implement **blacklisting** or **token invalidation checks**.

**✅ Option 1 — Maintain a “revoked token store”**

* Keep a Redis or database table of revoked tokens or jti (JWT IDs).
* When a JWT is issued, store its unique ID claim (jti) and expiry.
* When you want to invalidate a user:
  + Add their JWT’s jti or userId to the blacklist.
* Gateway or auth filter checks this store on each request.

if (redis.contains("revoked:" + jti)) {

rejectRequest("Token revoked");

}

✅ Works well for user-specific logout  
⚠️ Adds DB/Redis lookup per request (small performance cost)

**✅ Option 2 — Maintain “user lastLogoutAt” timestamp**

Simpler and scalable.

1. Each user has a field lastLogoutAt in the DB.
2. When a JWT is issued → embed "iat" (issued at).
3. When verifying JWT → check:

if (token.iat < user.lastLogoutAt)

→ reject (token issued before logout)

1. When you force logout a user (e.g., credentials compromised) → update lastLogoutAt.

✅ No need to track token IDs  
✅ Stateless tokens remain stateless  
✅ Works perfectly with short TTLs

**✅ Option 3 — Combine with Refresh Tokens**

* Access tokens → short-lived (e.g., 15 min)
* Refresh tokens → long-lived, stored in DB
* If user is compromised → delete refresh token record
* Effect: user’s access token expires soon, can’t refresh again

✅ Practically revokes user access  
✅ Lightweight and efficient

**🧱 7️⃣ Summary Table**

| **Problem** | **Naïve Behavior** | **Enterprise Fix** |
| --- | --- | --- |
| Rotate JWT key | All users logged out | Use key versioning (kid) + overlap |
| Prevent user logout during rotation | Impossible | Keep old key until expiry window passes |
| Revoke 1 user’s JWT | Stateless → no revoke | Use lastLogoutAt or jti blacklist |
| Token reuse after password change | Token still valid | Compare iat vs. passwordChangedAt |
| Global logout | Rotate all keys instantly | Optional security reset |

**☁️ 8️⃣ Cloud-level automation of key rotation**

In **AWS / GCP / Vault**, this can be automatic:

* AWS Secrets Manager → auto-rotate every 90 days
* Vault → versioned key storage with TTLs
* Jenkins or ArgoCD pipeline → detect rotation → redeploy services with new active key
* Gateway → always loads both latest and previous versions (from Secrets Manager)

No downtime, no manual restarts, no user logout.

## Info

**1) High-level enterprise architecture (recommended)**

Client apps ↔ **API Gateway** ↔ Microservices (Customer, Order, …)

* **Dedicated Identity Provider (IdP)** (OAuth2 / OIDC) — e.g. Keycloak (self-hosted), Okta/Auth0 (managed), or an enterprise IdP.
* **API Gateway** (Kong / Apigee / NGINX / Spring Cloud Gateway with Oauth2) enforces perimeter auth, rate limits, TLS, WAF, and preliminary authZ.
* **Service Mesh** (Istio / Linkerd) for service-to-service mTLS, observability, and policy enforcement.
* **Policy Engine** (OPA/Gatekeeper) for fine-grained authorization (RBAC/ABAC).
* **Secrets Manager** (Vault / Azure Key Vault / AWS Secrets Manager) for signing keys, DB creds.
* **Audit & SIEM** for logging auth events, revocations, failed logins.

Diagram (text):

[Client] -> [IdP (Auth Server)] (login, token)

Client + JWT -> [API Gateway (validate JWT, ACLs, rate-limit)]

Gateway -> [Service Mesh (mTLS)] -> [Order Service, Customer Service, ...]

OPA policies consult data stores (groups, attributes)

Secrets in Vault, keys rotated via CI/CD

Audit logs -> SIEM

**2) Authentication pattern (enterprise)**

Use **OAuth2 / OIDC** (industry standard) with flow choices by client type:

* **Web clients**: Authorization Code + PKCE (OIDC)
* **Native/mobile**: Authorization Code + PKCE
* **SPAs**: Authorization Code + PKCE (no implicit)
* **Server-to-server**: Client Credentials (machine tokens)
* **Backend for frontends (BFF)**: use BFF to avoid exposing refresh tokens to browsers

Tokens:

* **Access tokens**: short lived (minutes). Prefer JWT (self-contained) for performance OR opaque tokens + introspection for revocation control (tradeoffs below).
* **Refresh tokens**: long-lived, rotate on use; stored securely on client (or not used for machine clients).
* Use **refresh token rotation** and **revocation lists**.

Preferred enterprise IdP: **Keycloak** (good for on-prem), or managed providers (Okta/Auth0/Azure AD) if you want SaaS and enterprise SSO/SCIM.

**3) Authorization pattern (enterprise)**

* **Gateway** performs authentication (verify token signature via JWKS) and simple authZ (scopes).
* **OPA (Open Policy Agent)** for centralized fine-grained policies (example: "customer can view orders only for their customerId").
* **Microservices** enforce resource-level authorization using claims in token (sub, roles, scopes) + consult OPA for ABAC as needed.
* **Roles & scopes model**:
  + Scopes = coarse capabilities (read:orders, write:orders)
  + Roles = business roles (admin, sales, customer)
  + Claims include tenantId/customerId for multi-tenancy.

**4) Token strategy — JWT vs Opaque**

* **JWT (self-contained)**: fast (gateway and services validate locally using JWKS). Use short expiry (e.g., 5–15 min) and refresh tokens. Rotate signing keys (kid).
* **Opaque tokens**: tokens are random strings and require introspection against IdP — enables immediate revocation but adds IdP load and latency. Cache introspection results in gateway with TTL.
* Enterprise approach: **use JWT for typical traffic + introspection for critical endpoints** OR use opaque for highly-sensitive scopes. Or use **reference tokens + caching**.

**5) Service-to-service security**

* **Mutual TLS (mTLS)** between services (use service mesh — Istio or Linkerd provide automatic mTLS). This ensures only legitimate services talk to each other.
* **SPIFFE / SPIRE** for workload identity (short-lived certs).
* Service accounts + client credentials for back-end calls (no static secrets).

**6) Where “Customer Service (login APIs)” fits**

* In an enterprise design, prefer a **dedicated Auth server** rather than embedding auth in Customer Service:
  + Move auth responsibilities to IdP (Keycloak).
  + Use Customer Service as a user management/profile service (connect to IdP via SCIM or LDAP sync).
  + If you must keep Customer Service as auth provider initially, it should act as an **adapter** to the IdP or be reworked into the Auth Server eventually.

**7) Gateway responsibilities (enterprise)**

* Validate JWT via JWKS (or introspect opaque tokens).
* Enforce global authZ rules (scopes) and rate limits, CORS, header hygiene.
* Offload TLS termination (with cert management).
* Inject identity headers for downstream services (e.g., x-user-id, x-roles) **only** over mTLS / internal network.
* Integrate with WAF and API analytics.

**8) Operational & security hardening checklist**

* TLS everywhere (ingress, internal, egress).
* Rotate keys & certificates automatically (short lived).
* Store secrets in Vault / KMS; do not bake into images.
* JWKS endpoint + automated key rotation and discovery.
* Token lifetime policy: short access tokens, rotate refresh tokens.
* Implement anomaly detection: repeated failed logins, token misuse.
* Audit trails for all auth events sent to SIEM (who logged in, token issuance, revocations, failed attempts).
* Regular pen tests and vulnerability scanning.
* Rate limits & WAF at gateway.
* Implement logging & metrics for token validation failures, latency, policy denials.
* Implement SSO and SCIM if integrating with corporate directories (AD/LDAP).

**9) Deployment & CI/CD practices**

* Deploy IdP in HA across AZs / regions (or use managed).
* Automate config via GitOps (sealed secrets for sensitive values).
* Automate secrets rotation and revocation in CI/CD.
* Roll key rotations gracefully — publish new jwks and support kid fallback.
* Canary deployments + automated security tests.
* Use infrastructure policy checks (OPA/Gatekeeper) for cluster admission.

**10) Phased rollout plan (practical)**

**Phase 0 — Design & PoC**

* Choose IdP (Keycloak or managed).
* Build a small PoC: Keycloak + Spring Cloud Gateway + 2 services (Customer, Order).
* Use JWT validation at Gateway, Resource Server in services.

**Phase 1 — Expand & Integrate**

* Move all login flows to IdP, integrate SSO and SCIM sync for users.
* Add OPA for fine-grained policies and basic RBAC.

**Phase 2 — Harden**

* Add service mesh for mTLS.
* Implement secrets manager and rotate keys.
* Implement refresh token rotation and revocation.

**Phase 3 — Monitoring & Compliance**

* Enable SIEM, audit logs, and penetration testing.
* Document SSO, onboarding/offboarding (SCIM), and incident response.

**11) Practical tech stack suggestions**

* **IdP**: Keycloak (on-prem) or Okta/Auth0/Azure AD (managed).
* **Gateway**: Kong / Apigee / NGINX / Spring Cloud Gateway (+ OAuth2 plugin).
* **Service Mesh**: Istio or Linkerd.
* **Policy**: OPA (Rego) or Keycloak authorization services.
* **Secrets**: HashiCorp Vault / Cloud KMS.
* **Auth Libs**: Spring Security OAuth2 Resource Server for each Spring service.
* **Monitoring**: Prometheus, Grafana, Jaeger, ELK/Loki; SIEM for audit.

**12) Recommended minimal enterprise rules (quick)**

1. Token validation at gateway + resource server check at service if sensitive.
2. Keep access tokens short; use refresh tokens securely.
3. Use JWKS discovery; avoid hardcoded keys.
4. mTLS between services.
5. OPA for policies and avoid putting business authorization only in app code.
6. Centralize user management (SCIM) with IdP.

**13) Example mapping to your components**

* **Discovery** (Eureka): internal service discovery; ensure it’s accessible only inside mesh (mTLS).
* **Gateway**: OAuth2/OIDC client to IdP; validate tokens; enforce global policies.
* **Config**: store non-sensitive config; sensitive keys in Vault; restrict access.
* **Customer Service**: become a user/profile service (no longer responsible for token issuance).
* **Order Service**: Resource server with claims-based checks (customerId matches token claim or consult OPA).

**14) Ready-to-use artifacts I can produce for you**

I can create in this session (pick any/all):

* A **detailed implementation plan** (phases, tickets, roles) for your org.
* A **PoC repo** structure (Keycloak + Spring Cloud Gateway + two Spring Boot services) with config and Docker Compose or k8s manifests.
* **Sample OPA Rego policies** for common rules (tenant isolation, role checks).
* **CI/CD snippets** to rotate keys and publish JWKS.
* **Checklist and runbook** for identity operations (onboard/offboard, incident response).