# 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 |

# 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 |

## 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).