API Security for Microservices — Detailed Breakdown

1️⃣ Why API Security Is Critical

- Protects against **data breaches**, **unauthorized access**, and **service misuse**.
- Ensures **integrity**, **confidentiality**, and **availability** of APIs.
- Prevents OWASP API Top 10 threats (e.g., Broken Object Level Authorization, Injection, Excessive Data Exposure).

2️⃣ Common Security Threats

- **Broken Authentication** → attackers bypass login or steal tokens.
- **Broken Authorization** → users access data they shouldn't.
- **Injection Attacks** → SQL, XSS, or command injections.
- **Excessive Data Exposure** → APIs returning unnecessary sensitive fields.
- **DoS Attacks** → overload server via unthrottled requests.
- **Insecure Direct Object References (IDOR)** → manipulating IDs to access others' data.

3️⃣ Multi-Layered Security Approach

- **Network Layer:** Use HTTPS, firewall, VPN, private subnets.
- **Gateway Layer:** Use API Gateway (Apigee, Kong, AWS API Gateway) → throttling, IP whitelisting, token validation.
- **Application Layer:** Authentication, Authorization, Input validation.
- **Data Layer:** Encrypt sensitive data (AES, RSA). Avoid storing plain text.

4️⃣ Authentication & Authorization

- **Authentication** → verifies user identity (login, token).
- **Authorization** → grants access to specific APIs based on roles/permissions.
- **Principle:** "Authenticate once, authorize always."

Common Methods:

- Basic Auth (not recommended for production).
- JWT Tokens.
- OAuth2 / OpenID Connect.
- API Keys (internal service-to-service).

5️⃣ JWT (JSON Web Token)

- Stateless, lightweight, easy for distributed systems.
- Contains claims → `sub`, `iat`, `exp`, `roles`.
- Sent in header:

```
Authorization: Bearer <token>
```

- Signed using secret key (HS256) or private key (RS256).
- No need for session storage.

Spring Boot Implementation Steps:

1. Login API authenticates user and generates JWT.
2. `JwtRequestFilter` validates token on each request.
3. If valid → sets authentication context.
4. Access control via `@PreAuthorize("hasRole('ADMIN')")`.

6️⃣ OAuth2 & OpenID Connect

- Best for **enterprise and 3rd-party integrations**.

- Uses **Access Tokens** and **Refresh Tokens**.
- Tools: Keycloak, Auth0, Okta, Azure AD.
- Microservices act as **Resource Servers**, validating tokens from Auth Server.
- Supports **SSO**, **scopes**, and **token introspection**.

7️⃣ Input Validation

- Always validate incoming request data.
- Use **Spring Validation** (`@Valid`, `@NotBlank`, `@Size`, etc.).
- Sanitize data to avoid injection attacks.
- Validate:
  - Request Body (JSON fields)
  - Query Params
  - Path Variables
  - Headers

8️⃣ Global Exception Handling

- Use `@RestControllerAdvice` + `@ExceptionHandler`.
- Centralize error handling for consistency.
- Avoid leaking stack traces or system info.
- Custom error response with timestamp, error code, and message.

Example:

```
@RestControllerAdvice public class GlobalExceptionHandler
{ @ExceptionHandler(AccessDeniedException.class) public
ResponseEntity<?> handleAccessDenied(AccessDeniedException ex)
{ return
ResponseEntity.status(HttpStatus.FORBIDDEN) .body(Map.of("error",
"Access Denied")); } }
```

9️⃣ HTTPS / TLS Encryption

- Always serve APIs over **HTTPS**.
- Use **SSL certificates** (Let's Encrypt, AWS ACM).
- Terminate SSL at API Gateway or Load Balancer.
- Redirect all HTTP to HTTPS.

🔟 Rate Limiting & Throttling

- Prevent DDoS and brute-force attacks.
- Implement via:
  - API Gateway policies (Apigee, Kong).
  - In-service tools: **Bucket4j**, **Resilience4j RateLimiter**.
- Example: Limit to `100 req/min` per IP/user.

1️⃣1️⃣ Cross-Origin Resource Sharing (CORS)

- Restrict allowed origins, headers, and methods.
- Example Spring config:

```
@Bean public WebMvcConfigurer corsConfigurer() { return new
WebMvcConfigurer() { @Override public void
addCorsMappings(CorsRegistry registry)
{ registry.addMapping("/api/**") .allowedOrigins("https://myfrontend.c
om") .allowedMethods("GET","POST"); } }; }
```

1️⃣2️⃣ Secrets Management

- Never hardcode passwords, keys, or tokens.
- Use secret stores:
  - AWS Secrets Manager
  - Azure Key Vault

    - HashiCorp Vault
- Inject via environment variables or vault integration.

1️⃣3️⃣ Security Headers

- Add HTTP response headers to prevent browser attacks.
    - `Strict-Transport-Security`: enforce HTTPS
    - `X-Frame-Options`: prevent clickjacking
    - `X-Content-Type-Options`: avoid MIME sniffing
    - `Content-Security-Policy`: restrict script sources

Spring Example:

```
http .headers() .contentSecurityPolicy("default-src
'self'") .frameOptions().deny() .httpStrictTransportSecurity().include
SubDomains(true);
```

1️⃣4️⃣ Dependency & Vulnerability Scanning

- Scan for insecure dependencies continuously.
- Tools:
    - **OWASP Dependency Check**
    - **SonarQube Security Plugin**
    - **Snyk**
    - **Checkmarx**
    - **Fortify**
- Integrate scans in CI/CD pipelines.

1️⃣5️⃣ Logging, Monitoring & Auditing

- Log API activity with correlation ID.
- Avoid logging sensitive data (PII, tokens).
- Use centralized logging:

    - ELK Stack (Elasticsearch, Logstash, Kibana)
    - Splunk
    - Datadog
- Configure alerts for failed logins, token misuse, or unusual traffic.

1️⃣6️⃣ Security Testing Tools

- **OWASP ZAP** → automated vulnerability scanner.
- **Burp Suite** → penetration testing.
- **Postman** → API security tests.
- **K6 / JMeter** → performance & load testing (to ensure resilience).

1️⃣7️⃣ Code & Deployment Best Practices

- Apply **Principle of Least Privilege**.
- Use **immutable infrastructure** (containers).
- Regularly **rotate tokens, passwords, certificates**.
- Enforce **strong password policies** and **multi-factor authentication** (MFA).
- Keep dependencies **patched and updated**.

☑️ Summary — Secure API Checklist

- ✅ HTTPS enabled
- ✅ JWT or OAuth2 Auth
- ✅ Input validation
- ✅ Global exception handling
- ✅ Rate limiting
- ✅ CORS restrictions
- ✅ Secret management
- ✅ Security headers
- ✅ Static and dynamic scanning

- ✅ Centralized logging and alerting

==========

🚀 Performance Improvement of Services (Complete Guide)

We'll organize this like a **real architecture review** or **interview answer**, with **clear sections**, **actionable techniques**, and **example snippets**.

📑 Table of Contents

1️⃣ Understanding Service Performance

Definition:

Service performance = **how efficiently a service handles requests** in terms of:

- **Latency (Response Time)**
- **Throughput (Requests per second)**
- **Resource usage (CPU, Memory, IO, DB connections)**

Objective:

Improve:

- **Speed** ⏱️
- **Scalability** 📈
- **Reliability under load** 🧱

2️⃣ Performance Metrics

Metric	Description	Ideal Range
Latency (p50, p95, p99)	Time taken to respond to requests	p95 < 200ms
Throughput (TPS/QPS)	Requests handled per second	As high as infra allows
CPU Utilization	Percent of CPU used	< 70%
Memory Usage	Heap/Stack space used	Stable over time
GC Pause Time	Garbage collection delays	< 100ms
Error Rate	% of failed requests	< 0.1%
Thread Pool Saturation	% of max pool size	< 80%

3️⃣ Performance Bottlenecks (Common Areas)

1. **Slow DB queries**
2. **Blocking I/O calls**

3. **Synchronous downstream service calls**
4. **Large payloads / serialization**
5. **Inefficient loops or data structures**
6. **Excessive object creation**
7. **Thread contention / deadlocks**
8. **Poor caching strategy**

4️⃣ Application-Level Improvements

🧩 4.1 Connection Pooling

Problem: Creating new DB connections for each request.
Solution: Use connection pools (`HikariCP`, `C3P0`).

```
# application.yml spring.datasource.hikari.maximum-pool-size: 20
spring.datasource.hikari.connection-timeout: 30000
```

🧠 **Benefit:** Reuses existing connections, avoids expensive setup time.

⚡ 4.2 Enable HTTP/2 or gRPC

HTTP/2 multiplexes multiple requests over one connection.
Use **gRPC** for inter-service communication to reduce overhead.

🔄 4.3 Use Asynchronous or Non-Blocking Calls

Use:

- `CompletableFuture`
- `WebClient` (Spring Reactive)
- `@Async`

```
@Async public CompletableFuture<User> getUserDetails(Long id) { return
CompletableFuture.supplyAsync(() ->
userRepository.findById(id).orElseThrow()); }
```

🔒 4.4 Use Bulkheads, Circuit Breakers

Use **Resilience4j** or **Spring Cloud CircuitBreaker**:

```
@CircuitBreaker(name = "userService", fallbackMethod = "fallbackUser")
public User getUser(Long id) { ... }
```

🧠 Avoid cascading failures in high-load scenarios.

5️⃣ Threading, Concurrency & Async

🧵 Thread Pool Optimization

- Use `Executors.newFixedThreadPool()` or `ThreadPoolTaskExecutor` in Spring.
- Tune core and max threads based on CPU cores and workload.

```
@Bean public Executor taskExecutor() { ThreadPoolTaskExecutor executor
= new ThreadPoolTaskExecutor(); executor.setCorePoolSize(10);
executor.setMaxPoolSize(50); executor.setQueueCapacity(100);
executor.initialize(); return executor; }
```

🧠 Monitor

- Pool exhaustion → leads to rejected tasks.
- Too many threads → context-switching overhead.

6️⃣ Network & API Layer Optimization

🚀 Use GZIP / Brotli Compression

Reduce payload size via:

```
server.compression.enabled: true server.compression.mime-types: application/json,application/xml
```

🧠 Keep Responses Lightweight

- Use projections (`@JsonIgnore`, DTOs)
- Remove unused fields in API response

🧰 Pagination

Never fetch entire dataset.
Use:

```
Pageable pageable = PageRequest.of(0, 10);
repository.findAll(pageable);
```

7️⃣ Database & Caching Layer Optimization

🗃️ 7.1 Use Indexing

Analyze queries with **EXPLAIN PLAN**.

💾 7.2 Add Caching Layer (Redis / Caffeine)

```
@Cacheable(value = "userCache", key = "#id") public User getUser(Long id) { ... }
```

🧠 Caches frequent reads and reduces DB load.

7.3 Optimize ORM

- Use lazy loading carefully
- Avoid N+1 query problem via `@EntityGraph` or `fetch join`
- Use batch inserts:

```
spring.jpa.properties.hibernate.jdbc.batch_size: 30
```

8 Code-Level Performance Improvements

Use Efficient Data Structures

- Prefer `HashMap` over `List` search loops.
- Stream optimizations:

```
users.stream().parallel().filter(u ->
u.isActive()).collect(Collectors.toList());
```

Reduce Object Creation

- Reuse immutable objects
- Use primitive types where possible

Avoid Synchronization Overuse

- Lock only critical sections
- Use **ConcurrentHashMap**, **AtomicInteger**, etc.

9️⃣ Microservices Communication & Scalability

🧭 API Gateway Caching

Use **Spring Cloud Gateway + Redis Cache**.

📡 Asynchronous Messaging

Offload heavy operations via **Kafka**, **RabbitMQ**, or **SQS**.

☁️ Load Balancing & Autoscaling

Use:

- Kubernetes **HPA (Horizontal Pod Autoscaler)**
- AWS ECS autoscaling policies

🔟 Container & Deployment Optimization

🧰 Docker

- Use **Alpine base images**
- Reduce image size (multi-stage builds)

🧱 Kubernetes

- Set CPU/memory **requests & limits**
- Enable **liveness/readiness probes**

🌐 CDN for static assets

Use CDN (CloudFront, Akamai) to offload bandwidth.

1️⃣1️⃣ Performance Testing & Tools

Tool	Usage	Example
JMeter	Load testing	Simulate 1000 concurrent users
Gatling	Scalable performance testing (Scala-based)	Define request scenarios
Apache Benchmark (ab)	CLI benchmarking	`ab -n 1000 -c 100 http://localhost:8080/api/test`
wrk	Modern load testing tool	`wrk -t12 -c400 -d30s http://localhost:8080`
Postman Runner	Light API test runner	Small-scale load testing
Locust	Python-based distributed load tests	Web UI to simulate traffic

🗄️ Example JMeter Test Plan

- Thread Group: 500 users
- Ramp-up time: 30 seconds
- Throughput: 200 requests/sec
- Assertions: p95 latency < 250ms

1️⃣2️⃣ Monitoring, Profiling & Observability

Category	Tool	Usage
APM	New Relic, Datadog, AppDynamics	End-to-end latency tracing
Metrics	Prometheus + Grafana	Track CPU, memory, request time
Logs	ELK Stack (Elasticsearch, Logstash, Kibana)	Log analysis
Profiling	VisualVM, JProfiler, Java Flight Recorder	Find CPU/memory hotspots
Distributed Tracing	OpenTelemetry, Zipkin, Jaeger	Trace requests across microservices

Example Spring Boot Actuator:

```
management.endpoints.web.exposure.include: health, metrics, prometheus
management.metrics.export.prometheus.enabled: true
```

1️⃣3️⃣ Sample Strategy: Complete Service Optimization Flow

Example Scenario:
A product microservice has high response times (~2s).
Approach:

Step	Action	Result
Step 1	Add logs + actuator metrics	Found DB query taking 1.2s
Step 2	Added Redis cache for product data	Response ↓ 1.2s → 200ms
Step 3	Enabled async calls for inventory check	Parallelized external calls
Step 4	Used HikariCP for DB pool tuning	Reduced connection overhead
Step 5	Added Prometheus & Grafana	Continuous latency tracking

Result:

- ✅ 4x faster response time
- ✅ 40% lower CPU usage
- ✅ System scales to 10x load under 500ms latency

☑️ Summary

Layer	Optimization Technique	Tools
Application	Async, caching, circuit breaker	Resilience4j, Redis
Database	Query tuning, indexing	PgHero, MySQL EXPLAIN
Code	Stream & data structure tuning	JProfiler, Flight Recorder
Infra	Autoscaling, load balancing	Kubernetes HPA, Nginx

Monitoring	Metrics + tracing	Prometheus, Grafana, Zipkin
Testing	Load & stress tests	JMeter, Gatling