Security

Closing the unconditional trust every lesson in this category has assumed so far: introducing Spring Security into the course for the first time, the authentication vs. authorization distinction, JWT's self-contained nature, validating JWTs INDEPENDENTLY at both api-gateway and order-service (zero trust), role-based authorization, and propagating identity across services the same way the correlation id already is.

Intermediate 26 min

Last updated: August 23, 2026
Read online: https://www.learnforgex.com/en/topics/security

Security

Every lesson in this category so far has assumed order-service and inventory-service simply trust whatever calls them. That's been a reasonable simplification while the focus was elsewhere -- but a real system needs to answer two questions this course hasn't touched yet: who is making this request, and are they allowed to do what they're asking? This lesson introduces Spring Security into the course for the first time, scoped specifically to this category's services.

What Does Security Mean for a Microservices System?

In a single application, security often means one login check at the front door. In a microservices system, EVERY service that receives a request -- not just the one facing the public internet -- needs its own answer to "who is this, and are they allowed to do this," because a request can reach an internal service (inventory-service) through paths that never touch the public-facing one (api-gateway) at all, whether by misconfiguration or by design.

Why Does It Exist?

Without any identity check, ANY request that can reach order-service's network can place orders, and any request that can reach inventory-service directly could bypass api-gateway entirely -- the routing and cross-cutting concerns api-gateway provides (see the API Gateway lesson) are convenience and structure, not a security boundary by themselves. A system that only checks identity at api-gateway and then trusts every internal call unconditionally is only as secure as its LEAST protected internal path.

History

Token-based authentication for HTTP APIs became the dominant pattern as single-page applications and mobile clients replaced server-rendered, session-cookie-based logins through the 2010s -- a stateless token that any service can verify independently fits a distributed system's shape far better than a shared server-side session ever could. JSON Web Token (JWT), standardized in RFC 7519 (2015), became the most common shape for that token specifically because it's self-contained and independently verifiable (see "JWT: A Self-Contained, Verifiable Identity") -- no service needs to call back to a central session store just to check who's making a request.

Authentication vs. Authorization: Two Different Questions

These two words are often used loosely, but they answer genuinely different questions. Authentication asks "who is this?" -- verifying an identity, typically by checking a token's signature. Authorization asks "is THIS identity allowed to do THIS specific thing?" -- a completely separate decision that happens AFTER authentication succeeds (see "Authorization: Restricting an Endpoint by Role"). A request can be authenticated (a real, valid identity) and still be unauthorized (that identity just isn't allowed to do what it's asking).

JWT: A Self-Contained, Verifiable Identity

A JWT carries its own claims (who issued it, who it identifies, what roles or scopes it grants, when it expires) and a cryptographic signature over all of that -- any service holding the issuer's public key can verify the signature and trust the claims, without ever contacting the issuer directly for THIS specific check. This is what lets api-gateway and order-service both verify the SAME token independently (see "Validating a JWT at the Gateway" and "Why the Gateway Alone Isn't Enough").

Validating a JWT at the Gateway

api-gateway is the first service to see an external request, so it's the natural first place to reject one carrying no valid token at all.

```
import org.springframework.context.annotation.Bean;
import org.springframework.security.config.annotation.web.reactive.EnableWebFluxSecurity;
import org.springframework.security.config.web.server.ServerHttpSecurity;
import org.springframework.security.web.server.SecurityWebFilterChain;

// api-gateway's FIRST security configuration -- this course has used no Spring
// Security anywhere until now (the AI ingestion endpoint in the quiz feature
// used a hand-written X-Api-Key check specifically BECAUSE Spring Security
// wasn't already a dependency, see that feature's own design notes). Spring
// Cloud Gateway runs on WebFlux (see the API Gateway lesson), so this uses the
// REACTIVE security config style (@EnableWebFluxSecurity, ServerHttpSecurity),
// not the servlet-based style order-service uses in OrderServiceSecurityConfig.
@EnableWebFluxSecurity
class ApiGatewaySecurityConfig {

    @Bean
    SecurityWebFilterChain securityFilterChain(ServerHttpSecurity http) {
        return http
                .authorizeExchange(exchanges -> exchanges
                        .pathMatchers("/actuator/health").permitAll()   // health
                                                                          // checks stay
                                                                          // public --
                                                                          // load
                                                                          // balancers
                                                                          // need to
                                                                          // reach them
                                                                          // unauthenticated
                        .anyExchange().authenticated())
                // oauth2ResourceServer + jwt() tells Spring Security that a valid
                // request carries a JWT in its Authorization header, and how to
                // VERIFY it (see ApiGatewayJwtConfig.yml for where the verification
                // key comes from) -- see "JWT: A Self-Contained, Verifiable Identity".
                .oauth2ResourceServer(oauth2 -> oauth2.jwt(jwt -> {}))
                .build();
    }
}
```

```
# Added to api-gateway's application.yml. Spring Security's resource server
# support needs to know WHERE to fetch the public key(s) it uses to verify a
# JWT's signature -- issuer-uri points at an identity provider (a dedicated
# authentication service; this course assumes one already exists and issues
# tokens, the same way it assumes a Kafka broker already exists for the
# Event-Driven Architecture & Kafka lesson -- building an identity provider
# itself is out of scope here).

spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: https://auth.example.com/   # Spring Security fetches this
                                                    # issuer's public signing keys
                                                    # automatically on startup --
                                                    # no key material is
                                                    # hardcoded anywhere in
                                                    # api-gateway's own config
```

> This course has used no Spring Security anywhere until this lesson -- the AI ingestion endpoint built for this project's quiz feature deliberately used a hand-written `X-Api-Key` check instead, specifically because adding a whole security framework for one internal endpoint would have been disproportionate. A public-facing gateway handling real user identity is exactly the kind of case that justifies bringing Spring Security in.

Why the Gateway Alone Isn't Enough: Zero Trust Between Services

If order-service simply trusted every request that reached it, ANY path that bypasses api-gateway -- a misconfigured route, a service reachable directly on an internal network, a future service someone forgets to route through the gateway -- would have no protection at all. Zero trust means order-service verifies the JWT itself too, independently, rather than assuming "it must have already been checked."

```
import org.springframework.context.annotation.Bean;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.web.SecurityFilterChain;

// order-service's OWN security configuration -- see "Why the Gateway Alone
// Isn't Enough: Zero Trust Between Services" for why order-service validates
// the SAME JWT api-gateway already validated, instead of trusting that a
// request reaching it must have already passed the gateway's check. This is
// the SERVLET-based config style (@EnableWebSecurity, HttpSecurity), matching
// order-service's blocking Spring MVC controllers -- unlike
// ApiGatewaySecurityConfig's reactive style, which matches api-gateway's
// WebFlux runtime.
@EnableWebSecurity
class OrderServiceSecurityConfig {

    @Bean
    SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        return http
                .authorizeHttpRequests(requests -> requests
                        .requestMatchers("/actuator/health").permitAll()
                        // Creating an order requires the "customer" role --
                        // see "Authorization: Restricting an Endpoint by Role".
                        // Every OTHER authenticated request just needs a valid
                        // JWT, no specific role.
                        .requestMatchers(org.springframework.http.HttpMethod.POST, "
/orders")
                        .hasRole("customer")
                        .anyRequest().authenticated())
                .oauth2ResourceServer(oauth2 -> oauth2.jwt(jwt -> {}))
                .build();
    }
}
```

```
# Added to order-service's application.yml -- the SAME issuer-uri api-gateway
# uses, since both are verifying the SAME identity provider's tokens. This is
# what "zero trust between services" looks like in configuration: order-service
# doesn't ask api-gateway "did you already check this?" -- it verifies the
# token itself, independently, every time.

spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: https://auth.example.com/
```

> Notice `OrderServiceJwtConfig.yml` points at the SAME `issuer-uri` as `ApiGatewayJwtConfig.yml` -- both services verify the SAME tokens from the SAME identity provider, completely independently. Neither service asks the other "did you already check this?"

Authorization: Restricting an Endpoint by Role

Once a request is authenticated, `OrderServiceSecurityConfig`'s `.hasRole("customer")` rule (see above) makes the SEPARATE decision of who's allowed to place an order specifically -- an authenticated identity without that role gets a `403 Forbidden`, not the `401 Unauthorized` a missing or invalid token would produce.

Propagating Identity: The Correlation Id's Security Counterpart

order-service's own JWT doesn't automatically travel along when it calls inventory-service through `ResilientStockClient` (see the Resilience4j lesson) -- exactly the same gap the Observability lesson closed for the correlation id, now for identity instead.

```
import org.springframework.http.HttpRequest;
import org.springframework.http.client.ClientHttpRequestExecution;
import org.springframework.http.client.ClientHttpRequestInterceptor;
import org.springframework.http.client.ClientHttpResponse;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.oauth2.jwt.Jwt;

import java.io.IOException;

// The security counterpart of the Observability lesson's
// RestClientCorrelationIdInterceptor -- same shape, different header. Without
// this, order-service's call to inventory-service (see ResilientStockClient in
// the Resilience4j lesson) would carry NO identity at all, and inventory-
// service's own SecurityFilterChain (built the same way as
// OrderServiceSecurityConfig) would have nothing to authenticate -- either the
// call fails outright, or inventory-service has to trust it unconditionally,
// which is exactly the gap "Why the Gateway Alone Isn't Enough" warns against
// one level further down the chain.
//
// Registered on the SAME @LoadBalanced RestClient.Builder bean the Service
// Discovery & Eureka lesson introduced -- adding this interceptor (alongside
// RestClientCorrelationIdInterceptor) doesn't change anything else about how
// the call is made, only which headers travel with it.
class RestClientBearerTokenInterceptor implements ClientHttpRequestInterceptor {

    @Override
    public ClientHttpResponse intercept(HttpRequest request, byte[] body,
ClientHttpRequestExecution execution)
            throws IOException {
        // The Jwt Spring Security already validated for THIS incoming request
        // (see OrderServiceSecurityConfig) is available from the security
        // context -- forwarding its original, already-verified token value is
        // simpler and more honest than order-service minting a new one on
        // inventory-service's behalf.
        if (SecurityContextHolder.getContext().getAuthentication() instanceof
                org.springframework.security.oauth2.server.resource.authentication.
JwtAuthenticationToken jwtAuth) {
            Jwt jwt = jwtAuth.getToken();
            request.getHeaders().setBearerAuth(jwt.getTokenValue());
        }
        return execution.execute(request, body);
    }
}
```

Best Practices

- **Verify identity at every service that receives a request, not only the one facing the public internet** -- see "Why the Gateway Alone Isn't Enough".
- **Keep authentication and authorization as separate concerns**, even when they're configured close together (as in `OrderServiceSecurityConfig`) -- see "Authentication vs. Authorization".

- **Propagate identity across service boundaries deliberately**, the same way the correlation id is propagated -- see `RestClientBearerTokenInterceptor`, and the Observability lesson's `RestClientCorrelationIdInterceptor` it mirrors.
- **Keep health check endpoints public** (see `.pathMatchers("/actuator/health").permitAll()` above) -- load balancers and orchestrators need to reach them without a token.

Common Mistakes

- **Trusting api-gateway's authentication check as the system's ONLY security boundary.** Any internal service reachable by another path has no protection at all unless it verifies identity itself -- see "Why the Gateway Alone Isn't Enough".
- **Confusing a `401` with a `403`.** A `401 Unauthorized` means authentication itself failed (no token, or an invalid one); a `403 Forbidden` means authentication succeeded but authorization didn't -- conflating the two makes debugging a real access problem much harder.
- **Forgetting to propagate identity to a downstream service call.** Without `RestClientBearerTokenInterceptor`, inventory-service would receive a completely unauthenticated request from order-service, even though the ORIGINAL external request was properly authenticated.
- **Putting authorization logic inside a controller method as scattered `if` statements**, instead of declaring it where the rest of a service's security rules already live (`OrderServiceSecurityConfig`) -- scattering it makes a service's actual access rules hard to audit in one place.

Summary, Cheat Sheet, and Glossary

Security in a microservices system means every service that can receive a request needs its own answer to who's asking (authentication) and what they're allowed to do (authorization) -- trusting api-gateway's check alone leaves every other path unprotected. JWTs carry a verifiable identity independently checkable by any service holding the issuer's public key, which is what lets both api-gateway and order-service validate the SAME token without calling back to a central store. Identity needs to be deliberately propagated across service boundaries, the same way a correlation id is.

Quick reference:

```
@EnableWebSecurity
class SomeServiceSecurityConfig {
    @Bean
    SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        return http
                .authorizeHttpRequests(requests -> requests
                        .requestMatchers("/actuator/health").permitAll()
                        .anyRequest().authenticated())
                .oauth2ResourceServer(oauth2 -> oauth2.jwt(jwt -> {}))
                .build();
    }
}

// application.yml
// spring.security.oauth2.resourceserver.jwt.issuer-uri: https://auth.example.com/
```

Glossary

Authentication — Verifying who is making a request, typically by checking a token's signature.

Authorization — Deciding whether an already-authenticated identity is allowed to do a specific thing.

JWT (JSON Web Token) — A self-contained, cryptographically signed token carrying its own identity claims, independently verifiable by any party holding the issuer's public key.

Zero Trust — The principle that no service should assume a request has already been validated elsewhere, and should verify identity itself.

Resource Server — A service (like order-service or api-gateway here) configured to validate JWTs and enforce access rules based on their claims.