API Gateway

order-service and inventory-service becoming reachable to external clients through a single entry point: setting up a separate api-gateway application with Spring Cloud Gateway, the route predicate/URI/filter trio, discovery-based routing through Eureka with lb://, writing a custom GlobalFilter (the course's first reactive code), and why system-wide concerns like correlation ids belong at the gateway while business logic stays in the service.

Intermediate 24 min

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

---

API Gateway

So far, every call into this course's microservices has come from ANOTHER microservice -- order-service calling inventory-service, first with a hardcoded URL (see the "Inter-Service Communication" lesson), then by name through Eureka (see the "Service Discovery & Eureka" lesson's "Calling a Service by Name with a Load-Balanced RestClient" section). But real systems also have EXTERNAL clients -- a browser, a mobile app -- and those clients don't run inside the Eureka network, don't know service names, and shouldn't need to know that "orders" and "inventory" are even separate applications. This lesson introduces the piece that sits between external clients and the whole microservices system: the API Gateway.

What Is an API Gateway?

An API Gateway is a SINGLE entry point that sits in front of a set of microservices. An external client sends every request to ONE address (the gateway); the gateway looks at the request and ROUTES it to whichever internal service actually handles it -- order-service, inventory-service, or any service added later. The client never talks to order-service or inventory-service directly, and never needs their real addresses.

Why Does It Exist?

Without a gateway, an external client would need to know the address of EVERY microservice individually -- and that address list would change every time a service moved, scaled, or was renamed. Worse, without a single entry point, concerns that apply to the

WHOLE system (authentication, rate limiting, request logging) would need to be reimplemented inside EVERY service separately. An API Gateway solves both problems at once: ONE address for clients to remember, and ONE place to apply system-wide concerns -- instead of order-service and inventory-service each reinventing the same logic.

History

Netflix built one of the first widely-used API gateways, Zuul, around 2013, for the same internal infrastructure that produced Eureka (see the Service Discovery & Eureka lesson's "History" section). Zuul 1 was blocking, thread-per-request, built on the older Servlet model. Spring Cloud Gateway, introduced in 2018, is the modern Spring Cloud answer -- built on Spring WebFlux (reactive, non-blocking) from the start, since a gateway sits on the path of EVERY request in the system and benefits the most from handling many concurrent connections without tying up a thread per request. This lesson uses Spring Cloud Gateway.

Setting Up the Gateway Application

api-gateway is its OWN Spring Boot application -- a fourth one in this course, next to order-service, inventory-service, and eureka-server. It contains no business logic and no database of its own; its only job is receiving requests and forwarding them.

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

// A FOURTH Spring Boot application in this course, next to order-service,
// inventory-service, and eureka-server -- but api-gateway isn't a business
// microservice OR a registry. It's the SINGLE entry point external clients (a
// browser, a mobile app) talk to (see "What Is an API Gateway?") -- it never
// contains business logic of its own, it only ROUTES requests to the right
// internal service.
//
// No special annotation is needed here, unlike @EnableEurekaServer on
// EurekaServerApplication (see the Service Discovery & Eureka lesson's "Eureka
// Server: A Central Registry" section) -- adding the
// spring-cloud-starter-gateway dependency is what turns a plain Spring Boot app
// into a gateway; routing itself is configured, not annotated (see "Route
// Configuration: Predicates and Filters").
@SpringBootApplication
public class ApiGatewayApplication {
    public static void main(String[] args) {
        SpringApplication.run(ApiGatewayApplication.class, args);
    }
}
```

```
# api-gateway's own application.yml -- runs as its own independent process, on
# its own port, exactly like eureka-server and every business microservice in
# this course (see the Spring Boot Microservice Basics lesson's "Its Own
# `application.yml`: Port, Application Name, and Database" section).

server:
  port: 8090                          # never 8080 (learning-platform), 8081
                                       # (order-service), 8082 (inventory-service),
                                       # or 8761 (eureka-server)

spring:
  application:
    name: api-gateway                 # not just a log label -- other tools
                                       # (and eventually Observability, see the
                                       # upcoming lesson) will use this to tell
                                       # requests inside the gateway apart from
                                       # requests inside order-service/
                                       # inventory-service

eureka:
  client:
    service-url:
      defaultZone: http://localhost:8761/eureka/   # api-gateway is ALSO a
                                                       # Eureka client -- it needs
                                                       # the registry to resolve
                                                       # lb://order-service and
                                                       # lb://inventory-service
                                                       # (see "Discovery-Based
                                                       # Routing with lb://")

management:
  endpoints:
    web:
      exposure:
        include: health
```

> api-gateway is ALSO a Eureka client, for the same reason order-service and inventory-service are -- it needs to resolve service names to real addresses to route requests to them (see "Discovery-Based Routing with lb://" below).

Route Configuration: Predicates and Filters

A ROUTE is the gateway's core building block -- three things together: a PREDICATE (a condition that decides whether this route applies to an incoming request, most commonly matching on the request path), a URI (where a matching request gets forwarded), and optionally one or more FILTERS (transformations applied to the request or response along the way).

```
# Route definitions, ADDED on top of api-gateway's own application.yml (see
# ApiGatewayConfig.yml) -- this is where the gateway's actual job (matching an
# incoming request to the right internal service) is configured. A "route" is
# three things together: a PREDICATE (when does this route apply?), a URI
# (where does it forward to?), and optionally one or more FILTERS (what does it
# do to the request/response along the way?).

spring:
  cloud:
    gateway:
      routes:
        - id: orders-route
          uri: lb://order-service                # "lb://" -- NOT a real host,
                                                   # a service NAME resolved
                                                   # through Eureka, exactly
                                                   # like @LoadBalanced
                                                   # RestClient (see "Discovery-
                                                   # Based Routing with lb://")
          predicates:
            - Path=/orders/**                     # this route only matches
                                                   # requests whose path starts
                                                   # with /orders/
          filters:
            - StripPrefix=0                       # order-service's own
                                                   # controllers already expect
                                                   # paths starting with
                                                   # /orders -- StripPrefix=0
                                                   # means "forward the path
                                                   # exactly as received"

        - id: inventory-route
          uri: lb://inventory-service
          predicates:
            - Path=/inventory/**
          filters:
            - StripPrefix=0
```

Discovery-Based Routing with lb://

Notice the `lb://order-service` URI in the route configuration above -- this is the SAME idea as `@LoadBalanced RestClient` calling `http://inventory-service` in the Service Discovery & Eureka lesson (see its "Calling a Service by Name with a Load-Balanced RestClient" section): `lb://` is not a real protocol, it tells Spring Cloud Gateway to resolve `order-service` through Eureka and load-balance across however many instances are currently registered under that name. The gateway never needs order-service's real host and port hardcoded anywhere.

Writing a Custom Filter

Built-in filters like `StripPrefix` (used above) cover common cases, but a gateway can also run custom code on every request through a `GlobalFilter`. This is the first REACTIVE code

in this course -- Spring Cloud Gateway runs on Spring WebFlux, not the blocking Spring MVC used by order-service and inventory-service's controllers.

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.cloud.gateway.filter.GatewayFilterChain;
import org.springframework.cloud.gateway.filter.GlobalFilter;
import org.springframework.core.Ordered;
import org.springframework.stereotype.Component;
import org.springframework.web.server.ServerWebExchange;
import reactor.core.publisher.Mono;

// Spring Cloud Gateway is built on Spring WebFlux (reactive), NOT the same
// blocking Spring MVC used by order-service/inventory-service's controllers --
// this is the FIRST reactive code in this course. A GlobalFilter runs for
// EVERY route, unlike a filter attached to one specific route (see
// GatewayRoutesConfig.yml's per-route "filters:" list) -- implementing
// Ordered lets several global filters run in a defined sequence.
//
// The chain.filter(exchange) call is what actually forwards the request
// toward its destination (order-service or inventory-service, picked by the
// matching route) -- code BEFORE that call runs before forwarding, code
// inside .then(...) runs AFTER the downstream response comes back. Nothing
// here blocks a thread waiting for that response, unlike a plain
// @RestController method.
@Component
class RequestLoggingGlobalFilter implements GlobalFilter, Ordered {

    private static final Logger log = LoggerFactory.getLogger(RequestLoggingGlobalFilter.
class);

    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {
        long startedAt = System.currentTimeMillis();
        String method = exchange.getRequest().getMethod().name();
        String path = exchange.getRequest().getPath().value();

        return chain.filter(exchange)
                .then(Mono.fromRunnable(() -> {
                    long tookMillis = System.currentTimeMillis() - startedAt;
                    int status = exchange.getResponse().getStatusCode() != null
                            ? exchange.getResponse().getStatusCode().value()
                            : 0;
                    log.info("{} {} -> {} ({} ms)", method, path, status, tookMillis);
                }));
    }

    @Override
    public int getOrder() {
        return Ordered.LOWEST_PRECEDENCE;   // run last among global filters --
                                             // logs the FINAL outcome, after any
                                             // other filter has already run
    }
}
```

> A `GlobalFilter`'s `filter(...)` method must return a `Mono<Void>` and must never BLOCK the calling thread (no JDBC calls, no `Thread.sleep`, no blocking I/O) -- Spring WebFlux runs a small, fixed number of threads shared across every concurrent request; blocking even one of them stalls unrelated requests too.

Where Cross-Cutting Concerns Belong

A gateway is the natural place for concerns that apply to EVERY request, regardless of which service ultimately handles it -- assigning a correlation id, for instance, so one external request can be traced across multiple internal services later.

```
import org.springframework.cloud.gateway.filter.GatewayFilterChain;
import org.springframework.cloud.gateway.filter.GlobalFilter;
import org.springframework.core.Ordered;
import org.springframework.http.server.reactive.ServerHttpRequest;
import org.springframework.stereotype.Component;
import org.springframework.web.server.ServerWebExchange;
import reactor.core.publisher.Mono;

import java.util.UUID;

// A cross-cutting concern that genuinely BELONGS at the gateway (see "Where
// Cross-Cutting Concerns Belong"): every request entering the system gets a
// correlation id -- one shared value that order-service, inventory-service,
// and any future service can log alongside their own messages, making it
// possible to trace ONE external request across MULTIPLE internal services.
// This filter does NOT invent request tracing itself -- it only ensures the
// header exists as early as possible; an upcoming Observability lesson covers
// actually propagating and using it downstream.
@Component
class CorrelationIdGatewayFilter implements GlobalFilter, Ordered {

    private static final String CORRELATION_ID_HEADER = "X-Correlation-Id";

    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {
        ServerHttpRequest request = exchange.getRequest();
        if (request.getHeaders().containsKey(CORRELATION_ID_HEADER)) {
            // A caller (or a previous hop) already set one -- keep it AS IS, don't
            // overwrite an id a client may already be tracking.
            return chain.filter(exchange);
        }

        String correlationId = UUID.randomUUID().toString();
        ServerHttpRequest mutatedRequest = request.mutate()
                .header(CORRELATION_ID_HEADER, correlationId)
                .build();

        return chain.filter(exchange.mutate().request(mutatedRequest).build());
    }

    @Override
    public int getOrder() {
        return Ordered.HIGHEST_PRECEDENCE;   // run FIRST -- every later filter
                                              // (including RequestLoggingGlobalFilter)
                                              // and every downstream service should
                                              // see the header already set
    }
}
```

> This lesson only ASSIGNS the correlation id -- actually propagating it into outgoing calls between order-service and inventory-service, and using it to tie log lines together, is covered by the upcoming Observability lesson.

What a Gateway Should NOT Do

It's tempting to put business logic in the gateway too, since it already sees every request -- but that pulls business rules OUT of the services that own them and INTO a piece of infrastructure that has no domain knowledge. A gateway should route, and apply concerns that are genuinely about the REQUEST itself (authentication, rate limiting, logging, correlation ids) -- not decide, for example, whether an order is valid. That decision belongs to order-service, exactly as before.

Best Practices

- **Route by service NAME (`lb://service-name`), never by a hardcoded host:port** -- the same reasoning as `@LoadBalanced RestClient` (see the Service Discovery & Eureka lesson).
- **Keep filters focused on request/response concerns** (logging, correlation ids, headers) -- push anything resembling a business decision back into the owning service.
- **Register api-gateway with Eureka under its own `spring.application.name`** -- even though nothing routes TO the gateway itself through discovery, it keeps it visible in the registry alongside every other service.
- **Order global filters deliberately with `Ordered`** -- a filter that assigns a correlation id needs to run before a filter that logs requests using it.

Common Mistakes

- **Adding business logic (validation, calculations) directly inside a gateway filter.** That logic belongs in the owning microservice, not in shared infrastructure every request passes through.
- **Writing a `GlobalFilter` that blocks the thread** (a JDBC call, `Thread.sleep`) -- Spring WebFlux's threading model makes this far more damaging than the same mistake in a blocking Spring MVC controller.
- **Hardcoding a downstream service's host:port in a route's `uri`** instead of using `lb://service-name` -- reintroduces exactly the problem Service Discovery was meant to solve.
- **Expecting the gateway to aggregate responses from multiple services into one.** Plain Spring Cloud Gateway routes ONE request to ONE service; combining multiple calls into a single response is a different pattern (often called Backend for Frontend), out of scope for this lesson.

Summary, Cheat Sheet, and Glossary

An API Gateway is a single entry point external clients talk to instead of individual microservices. Spring Cloud Gateway routes are made of predicates (when a route applies),

a URI (where it forwards to, usually `lb://service-name` resolved through Eureka), and filters (transformations along the way) -- both built-in filters like `StripPrefix` and custom `GlobalFilter`s, which run on Spring WebFlux and must never block. The gateway is the right place for request-level, system-wide concerns (correlation ids, logging) -- never for business logic, which stays in the service that owns it.

Quick reference:

```
@SpringBootApplication
public class ApiGatewayApplication { ... }   // its own Spring Boot app, no
                                              // business logic of its own

// application.yml
// spring.cloud.gateway.routes:
//   - id: orders-route
//     uri: lb://order-service               // service NAME, resolved via Eureka
//     predicates:
//       - Path=/orders/**
//     filters:
//       - StripPrefix=0

@Component
class SomeGlobalFilter implements GlobalFilter, Ordered {
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {
        return chain.filter(exchange);       // forwards the request onward
    }
}
```

Glossary

API Gateway — A single entry point that routes external requests to the correct internal microservice.

Route — A gateway's core building block: a predicate, a destination URI, and optional filters.

Predicate — A condition (most commonly a path pattern) that decides whether a route applies to an incoming request.

GlobalFilter — Custom code that runs on every request passing through the gateway, written reactively on Spring WebFlux.

`lb://` — A pseudo-protocol telling Spring Cloud Gateway to resolve a service name through Eureka and load-balance across its instances, instead of using a fixed address.