# Microservice Configuration

Configuring a single Spring Boot microservice (order-service) from the ground up: its own entry point, its own application.yml (port, application name, database), a REST controller as the API contract, the service layer, the domain model, and a health check.

**Intermediate** 18 min

Last updated: August 18, 2026
Read online: https://www.learnforgex.com/en/topics/spring-boot-microservice-basics

---

# Microservice Configuration

In "Microservices Fundamentals" we saw *why* microservices exist, how service boundaries are defined, and the new challenges distributed systems bring -- all conceptually, with no code. In this lesson we write our first real code: we'll configure a **single** microservice ( `order-service` ), before it talks to any other service, from top to bottom -- its own entry point, its own port, its own database connection, the REST API it exposes, and its internal layers. In the next lesson of this category ( `inter-service-communication` ), we'll add an `inventory-service` alongside the `order-service` we build here and have the two talk to each other with a plain REST call -- the practical project there will be the real, runnable version of both services. Every example in this lesson is based on real Spring Boot APIs this project itself already uses -- `spring-boot-starter-web` , `spring-boot-starter-data-jpa` , and `spring-boot-starter-actuator` .

## What Is Microservice Configuration?

"Configuring a microservice" means turning the four pieces we listed in "Microservices Fundamentals"'s "The Anatomy of a Microservice" section into something concrete: its own entry point (a `main` method), its own network identity (a port and an application name), its own data connection (a connection string pointing at its own database), and the API surface it exposes. None of this lives in one file -- some of it lives in code (the entry point, the API surface) and some in a configuration file ( `application.yml` ); this lesson builds both, one at a time.

We'll use the name `order-service` throughout -- the same service from the e-commerce example in "Microservices Fundamentals." By the end of this lesson, `order-service` will be a fully independent Spring Boot application that can start on its own with `mvn spring-boot:run` and respond to `POST /orders` and `GET /orders/{id}` .

## Why Does It Exist?

This project ( `learning-platform` ) itself is a SINGLE Spring Boot application -- one `application.yml` , one port ( `8080` ), one database connection. As we saw in "Microservices Fundamentals"'s "Database per Service" section, once we set up two separate microservices like `order-service` and `inventory-service` , that's no longer enough -- each needs its **own** port, its **own** application name, and its **own** database connection, or else we can't run both on the same machine at the same time (both would try to listen on `8080` ), or one would pollute the other's schema (if they wrote to the same database).

The fix isn't some elaborate mechanism -- each microservice is simply its own independent Spring Boot project with its own `application.yml` . Just as `learning-platform` 's `application.yml` is specific to this project, `order-service` 's `application.yml` will be specific only to `order-service` ; nothing is shared between the two.

## History

The idea that every microservice should carry its own port, its own environment variables, and its own connection details traces back to **"The Twelve-Factor App,"** a methodology published by Heroku engineers in 2011 -- twelve principles for how applications should be built to run well in the cloud. Three of its principles are directly relevant to microservices: **"Config"** (anything that varies between environments -- ports, database connections, passwords -- is never hardcoded into the code, it's read from environment variables), **"Port Binding"** (a service binds its own port itself, rather than expecting to be injected into an external web server -- exactly the embedded Tomcat model we saw in the Spring MVC Fundamentals lesson's "Embedded Tomcat and spring-boot-starter-web" section), and **"Disposability"** (a service instance should be able to start and stop quickly, since scaling and deployment depend on that cycle).

Spring Boot supports most of these principles by default: `application.yml` moves configuration out of the code, expressions like `${ORDERS_DB_PASSWORD}` let it be read from environment variables, and embedded Tomcat means each service binds its own port. The `application.yml` we'll write in this lesson is a concrete application of these principles.

## A Microservice's Entry Point: @SpringBootApplication

Like every Spring Boot application, `order-service` has a single entry point:

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

// The entry point of a single, independent microservice -- order-service. This class is
// structurally identical to this project's own LearningPlatformApplication (see the Spring
// MVC Fundamentals lesson's "Embedded Tomcat and spring-boot-starter-web" section) --
// @SpringBootApplication still triggers the exact same component scanning and
// auto-configuration covered in the Spring Core category. Nothing in THIS class is what
// makes it a microservice; what does is that this JAR is built, deployed, and run
// completely independently of any other service (see the Microservices Fundamentals
// lesson's "The Anatomy of a Microservice" section).
@SpringBootApplication
public class OrderServiceApplication {
    public static void main(String[] args) {
        SpringApplication.run(OrderServiceApplication.class, args);
    }
}
```

This class is **structurally identical** to this project's own `LearningPlatformApplication`, which we saw in the Spring MVC Fundamentals lesson's "Embedded Tomcat and spring-boot-starter-web" section -- the same `@SpringBootApplication`, the same `SpringApplication.run(...)` call. The difference isn't in the code: this class runs in its own `.jar`, in its own process, with no other service present. Its implicit component scanning, as we saw in the Component Scanning lesson, will automatically find the `@RestController` / `@Service`-annotated classes in this package (the default package) -- including the `OrderController` and `OrderService` we're about to write.

## Its Own `application.yml`: Port, Application Name, and Database

The entry point barely says anything on its own -- the real identity lives in `application.yml`:

```
# order-service's own application.yml -- completely separate from any other service's
# configuration file. Nothing in here is shared with inventory-service or any other
# service; each one ships with (and owns) a file exactly like this.

server:
  port: 8081                          # order-service's own port -- inventory-service would
                                       # use a different one (e.g. 8082), never this one

spring:
  application:
    name: order-service               # this service's identity -- shows up in every log
                                       # line and, later, is how other services find it
                                       # (Service Discovery / Eureka)
  datasource:
    url: jdbc:postgresql://localhost:5432/orders_db   # its OWN database -- no other
                                                       # service ever connects to this URL
                                                       # (see the Microservices
Fundamentals
                                                       # lesson's "Database per Service"
                                                       # section)
    username: orders_user
    password: ${ORDERS_DB_PASSWORD}   # read from the environment, never hardcoded

management:
  endpoints:
    web:
      exposure:
        include: health              # exposes GET /actuator/health (see "Sağlık Kontrolü")
```

Notice three lines: `server.port: 8081` separates `order-service` from `learning-platform`'s own `8080` and from the port ( `8082` ) we'll give `inventory-service` later -- all three can run on the same machine at once. `spring.application.name: order-service` is this service's **identity** -- it'll show up in every log line, and later (Service Discovery / Eureka) it's exactly the name other services will use to find `order-service`. `spring.datasource.url` is the concrete form of the rule from "Microservices Fundamentals"'s "Database per Service" section -- `orders_db` is a database only `order-service` knows about; once `inventory-service` is set up, it'll connect to its own `inventory_db`, with nothing shared between the two.

> `password: ${ORDERS_DB_PASSWORD}` is deliberately **not** a plaintext password -- it's read from an environment variable. This is a direct application of the "Config" principle from the History section: secrets like passwords and API keys should never be written as plain text into `application.yml`.

## The External Surface: A REST Controller as the API Contract

We now build, with a real controller, the API surface we mentioned in "Microservices Fundamentals"'s "The Anatomy of a Microservice" section:

```
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

// order-service's external "contract" -- everything another service or a client can ask
// order-service to do goes through these two endpoints, and nothing else. The controller
// itself makes no business decisions -- that's OrderService's job (see OrderService.java)
--
// exactly the Controller -> Service split from the Spring MVC Fundamentals lesson's "The
// Journey of an HTTP Request: Request Lifecycle" section.
@RestController
@RequestMapping("/orders")
class OrderController {

    private final OrderService orderService;

    OrderController(OrderService orderService) {
        this.orderService = orderService;
    }

    @PostMapping
    ResponseEntity<Order> create(@RequestBody CreateOrderRequest request) {
        Order created = orderService.create(request.productName(), request.quantity());
        return ResponseEntity.status(HttpStatus.CREATED).body(created);
    }

    @GetMapping("/{id}")
    ResponseEntity<Order> findById(@PathVariable String id) {
        return orderService.findById(id)
                .map(ResponseEntity::ok)
                .orElseGet(() -> ResponseEntity.status(HttpStatus.NOT_FOUND).build());
    }

    record CreateOrderRequest(String productName, int quantity) {
    }
}
```

These two endpoints ( `POST /orders` , `GET /orders/{id}` ) are `order-service` 's **entire** external contract -- another service or client can only ask `order-service` for something through these two paths. Notice that `OrderController` itself makes no business decisions (it doesn't answer questions like whether a quantity can be zero or negative, or how an order gets stored) -- it just takes the request and hands it off to `OrderService` . This is the same Controller -> Service split from the Spring MVC Fundamentals lesson's "The Journey of an HTTP Request: Request Lifecycle" section; we'll look at `OrderService` itself in the next section.

## Separating Business Logic from the Controller: The Service Layer

The business logic `OrderController` hands off lives here:

```
import org.springframework.stereotype.Service;

import java.util.Map;
import java.util.Optional;
import java.util.UUID;
import java.util.concurrent.ConcurrentHashMap;

// order-service's business logic -- the only place that decides what a valid order looks
// like. A real deployment would back this with its own database (see OrderServiceConfig.yml
// and the Microservices Fundamentals lesson's "Database per Service" section); an
// in-memory map stands in for that here, so this example stays focused on the
// controller/service split rather than JPA/repository plumbing (already covered in the
// Spring MVC category's REST API Design topic).
@Service
class OrderService {

    private final Map<String, Order> orders = new ConcurrentHashMap<>();

    Order create(String productName, int quantity) {
        if (quantity <= 0) {
            throw new IllegalArgumentException("quantity must be positive");
        }
        String id = UUID.randomUUID().toString();
        Order order = new Order(id, productName, quantity);
        orders.put(id, order);
        return order;
    }

    Optional<Order> findById(String id) {
        return Optional.ofNullable(orders.get(id));
    }
}
```

The `quantity <= 0` check lives here, not in `OrderController` -- the answer to "what counts as a valid order" is a rule only `OrderService` should know, exactly like the "what business decisions can this service make on its own?" question from "Microservices Fundamentals"'s "Defining Service Boundaries" section. A real `order-service` would store orders in the `orders_db` we saw in "Its Own `application.yml`: Port, Application Name, and Database" -- here, a `ConcurrentHashMap` stands in for that persistence, because this lesson's focus isn't JPA /repository details (already covered in the Spring MVC category's REST API Design topic), it's the controller/service split itself.

## The Domain Model: What Does "Order" Mean in This Service?

The `Order` type shared by `OrderController` and `OrderService`:

```
// order-service's own view of what an "order" is -- just enough to satisfy THIS service's
// responsibility. If inventory-service or payment-service also had an "order"-shaped
// concept, they could model it completely differently (different fields, different rules)
// -- each service's model only has to make sense within its own bounded context, the idea
// covered in the Microservices Fundamentals lesson.
record Order(String id, String productName, int quantity) {
}
```

It's deliberately this short -- these three fields are everything `order-service` needs to know when it says "order." As we saw in "Microservices Fundamentals"'s "A Quick Look at Domain-Driven Design: Bounded Context" section, this model only has to make sense within `order-service`'s **own bounded context** -- if `inventory-service` also needed an "order"-shaped concept later (say, to see how much quantity to subtract from stock), it could model it completely differently, for its own purposes; there is **no** shared `Order` class between the two services.

## Health Checks: Is the Service Up?

When a single service is running, you rarely need to ask "is it up?" -- but whoever is watching an independently running service like `order-service` (a load balancer, an API Gateway we'll build later, or an orchestrator like Kubernetes) has to keep asking that question. The `management.endpoints.web.exposure.include: health` line from "Its Own `application.yml`: Port, Application Name, and Database" automatically exposes an endpoint that responds to `GET /actuator/health` once the `spring-boot-starter-actuator` dependency is added to the project -- with no code required:

```
$ curl http://localhost:8081/actuator/health
{"status":"UP"}
```

`{"status":"UP"}` tells you `order-service` is running and (since Actuator's default health indicators also check the database connection) can reach its database. If the service has crashed or can't connect to its database, the same request returns `{"status":"DOWN"}`. This simple-looking endpoint is the most basic building block that a possible later topic in this category, Observability (and infrastructure pieces like an API Gateway or Service Discovery -- we'll separately discuss when we'll need real verification from you for those), will be built on top of.

## Logging and Correlation: Which Log Came from Which Service?

In a single application like `learning-platform`, it's obvious where every console log line comes from -- there's only one application. Once `order-service` and `inventory-service` are both running, each writes to its own console (or its own log file) -- but thanks to the `spring.application.name: order-service` line from "Its Own `application.yml`: Port, Application

Name, and Database," a properly configured log format can tag every line with the service it came from:

```
2026-08-15 10:03:12 [order-service] INFO  OrderController - Creating order for "Keyboard"
2026-08-15 10:03:12 [inventory-service] INFO  InventoryController - Checking stock for
"Keyboard"
```

This distinction was unnecessary in a single application -- this is the simplest possible form of the observability challenge we saw in "Microservices Fundamentals"'s "The New Challenges of Distributed Systems" section: once you want to trace a request spread across ten services, you first need to be able to tell which log line came from which service. This is something a possible later topic in this category, Observability (with distributed tracing and correlation IDs), will go into much more deeply -- for now, what matters is that `spring.application.name` isn't just a label; it's the foundational identity that all future monitoring /logging infrastructure will be built on.

## Best Practices

- **Give every microservice its own `application.yml`, never share one** -- as we saw in "Its Own `application.yml`: Port, Application Name, and Database", the port, application name, and database connection should all differ between services.
- **Never write secrets (passwords, API keys) as plain text in `application.yml`** -- per the "Config" principle from the History section, these should always be read from environment variables ( `${...}` ).
- **Choose `spring.application.name` carefully, from the start** -- it's not just a log label; as we saw in "Logging and Correlation: Which Log Came from Which Service?", service discovery and distributed tracing will depend on this name later.
- **Keep the controller thin, and put business rules in the service layer** -- the `OrderController` / `OrderService` split from "Separating Business Logic from the Controller: The Service Layer" is the same principle from the Spring MVC Fundamentals lesson, applied in a microservices context.
- **Design your domain model around your own service's needs only -- it doesn't need to resemble another service's model** -- as we saw in "The Domain Model: What Does 'Order' Mean in This Service?", every service has its own bounded context.
- **Turn on the health check endpoint from day one** -- as we saw in "Health Checks: Is the Service Up?", this isn't something you add later as the service grows; it's a baseline you need from the start.

## Common Mistakes

**1. Trying to set up multiple microservices to share the same `application.yml` (or the same port).** As we saw in "Why Does It Exist?", this makes it impossible to run both at the same time, or lets one service's data pollute the other's.

**2. Writing passwords or connection strings as plain text in `application.yml`.** The `${...}` syntax in "Its Own `application.yml`: Port, Application Name, and Database" exists exactly to prevent this -- a plaintext password is compromised the moment the file is pushed to git.

**3. Putting business logic in the controller (answering "what decision can this service make?" inside the controller).** The `OrderController` in "The External Surface: A REST Controller as the API Contract" just hands the request off to `OrderService` -- writing decision logic into the controller breaks the Controller -> Service split from the Spring MVC Fundamentals lesson.

**4. Forgetting `spring.application.name`, or naming it randomly/inconsistently.** As we saw in "Logging and Correlation: Which Log Came from Which Service?", this name isn't just cosmetic -- service discovery and monitoring will depend on it later, and inconsistent naming breaks that infrastructure too.

**5. Putting off the health check endpoint as something to "add later."** This endpoint is the only way to tell from the outside whether a microservice is up -- as we saw in "Health Checks: Is the Service Up?", it might seem unimportant while a service runs on its own, but it becomes indispensable the moment a load balancer or orchestrator enters the picture.

## Summary, Cheat Sheet, and Glossary

Configuring a Spring Boot microservice means turning the conceptual anatomy from "Microservices Fundamentals" (entry point, its own port/identity, its own database, API surface) into concrete files. Key points:

- `@SpringBootApplication` gives the entry point, structurally identical to `learning-platform` 's own main class -- the difference is that it runs independently
- `application.yml` carries the service's identity ( `server.port` , `spring.application.name` ) and its data connection ( `spring.datasource.url` ) -- none of it shared with any other service
- Secrets (like `${ORDERS_DB_PASSWORD}` ) are read from environment variables, never written as plain text (the Twelve-Factor App's "Config" principle)
- The controller ( `OrderController` ) represents the external surface, the service ( `OrderService` ) the business logic, and the domain model ( `Order` ) the data shape specific to this service -- three separate responsibilities
- `GET /actuator/health` , provided automatically by `spring-boot-starter-actuator` , is the basic endpoint that tells the outside world whether the service is up
- `spring.application.name` isn't just a label -- service discovery and distributed logging /tracing will depend on this name later

Quick reference:

```
server:
  port: 8081

spring:
  application:
    name: order-service
  datasource:
    url: jdbc:postgresql://localhost:5432/orders_db
    password: ${ORDERS_DB_PASSWORD}

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

```
@SpringBootApplication
public class OrderServiceApplication {
    public static void main(String[] args) {
        SpringApplication.run(OrderServiceApplication.class, args);
    }
}

@RestController
@RequestMapping("/orders")
class OrderController {
    private final OrderService orderService;
    // ... constructor injection, the controller just hands off the request
}

@Service
class OrderService {
    // ... business logic and persistence live here
}
```

### Glossary

`application.yml` — The configuration file that carries a Spring Boot application's/service's environment-specific settings, like its port, identity, and connection details.

`server.port` — The port number a service's embedded server (Tomcat, in this project) listens on.

`spring.application.name` — A service's identity; it shows up in logs, and service discovery and distributed tracing depend on this name.

**Twelve-Factor App** — A twelve-principle methodology, published by Heroku engineers in 2011, for how applications should be built to run well in the cloud.

**Config (a Twelve-Factor principle)** — The principle that environment-specific values (ports, passwords, connection strings) should be read from environment variables rather than hardcoded into the code.

**Health check** — A mechanism, usually exposed as an endpoint like `GET /actuator/health`, that tells the outside world whether a service is up and running.

**Spring Boot Actuator** — The Spring Boot starter that automatically provides operational endpoints like health checks.