Mikroservis Yapılandırma

Tek bir Spring Boot mikroservisini (order-service) baştan sona yapılandırmak: kendi giriş noktası, kendi application.yml'i (port, uygulama adı, veritabanı), REST controller ile API sözleşmesi, service katmanı, domain modeli ve health check.

Orta 18 dk

Son güncelleme: 18 Ağustos 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/spring-boot-microservice-basics

---

Mikroservis Yapılandırma

"Microservices Temelleri" dersinde mikroservislerin *neden* var olduğunu, servis sınırlarının nasıl belirlendiğini ve dağıtık sistemlerin getirdiği yeni zorlukları kavramsal olarak gördük -- hiç kod yazmadan. Bu derste ilk kez koda giriyoruz: **tek** bir mikroservisi (`order-service`), henüz başka hiçbir servisle konuşmadan, baştan sona yapılandıracağız -- kendi giriş noktası, kendi portu, kendi veritabanı bağlantısı, dışa açtığı REST API'si ve iç katmanlarıyla. Bu kategorinin bir sonraki dersinde (`inter-service-communication`), burada kuracağımız `order-service`'in yanına bir `inventory-service` daha ekleyip ikisini sade bir REST çağrısıyla konuşturacağız -- oradaki pratik proje de bu iki servisin gerçek, çalıştırılabilir hâli olacak. Bu ders boyunca kullanılan tüm örnekler, `spring-boot-starter-web`, `spring-boot-starter-data-jpa` ve `spring-boot-starter-actuator` gibi bu projenin de kullandığı gerçek Spring Boot API'lerine dayanıyor.

Mikroservis Yapılandırma Nedir?

"Bir mikroserviste yapılandırma" derken kastedilen, "Microservices Temelleri" dersindeki "Bir Mikroservisin Anatomisi" bölümünde saydığımız dört parçanın somut karşılığı: kendi giriş noktası (bir `main` metodu), kendi ağ kimliği (bir port ve bir uygulama adı), kendi veri bağlantısı (kendi veritabanına giden bir bağlantı dizesi) ve dışa açtığı API yüzeyi. Bunların hepsi tek bir dosyada değil -- bir kod dosyasında (giriş noktası, API yüzeyi) ve bir yapılandırma dosyasında (`application.yml`) yaşar; bu ders ikisini de tek tek kuracak.

Örneğimiz boyunca `order-service` adını kullanacağız -- "Microservices Temelleri" dersinde e-ticaret örneğinde geçen aynı servis. Bu dersin sonunda `order-service`, kendi başına `mvn spring-boot:run` ile ayağa kalkabilen, `POST /orders` ve `GET /orders/{id}` isteklerine yanıt veren, tam bağımsız bir Spring Boot uygulaması olacak.

Neden Var?

Bu projenin (`learning-platform`) kendisi TEK bir Spring Boot uygulaması -- tek bir `application.yml` , tek bir port (`8080`), tek bir veritabanı bağlantısı. "Microservices Temelleri" dersinin "Database per Service" bölümünde gördüğümüz gibi, `order-service` ve `inventory-service` gibi iki ayrı mikroservis kurduğumuzda bu artık yeterli değil -- her birinin **kendi** portu, **kendi** uygulama adı ve **kendi** veritabanı bağlantısı olmalı, aksi hâlde ikisini aynı makinede aynı anda çalıştıramayız (ikisi de `8080` 'i dinlemeye çalışır) ya da birinin şeması diğerininkini kirletir (aynı veritabanına yazarlarsa).

Bunun çözümü karmaşık bir mekanizma değil -- her mikroservis, kendi `application.yml` 'ine sahip, bağımsız bir Spring Boot projesidir. `learning-platform` 'un `application.yml` 'i nasıl bu projeye özgüyse, `order-service` 'in `application.yml` 'i de yalnızca `order-service` 'e özgü olacak; ikisi arasında hiçbir paylaşım yok.

Tarihçe

Her mikroservisin kendi portunu, kendi ortam değişkenlerini ve kendi bağlantı bilgilerini taşıması fikri, 2011'de Heroku mühendislerinin yayınladığı **"The Twelve-Factor App"** metodolojisine dayanır -- bulut ortamında çalışan uygulamaların nasıl inşa edilmesi gerektiğine dair on iki ilke. Mikroservislerle doğrudan ilgili üç ilkesi: **"Config"** (yapılandırma -- port, veritabanı bağlantısı, parolalar gibi ortama göre değişen her şey kod içine gömülmez, ortam değişkenlerinden okunur), **"Port Binding"** (bir servis kendi portunu kendi bağlar, harici bir web sunucusuna enjekte edilmeyi beklemez -- tam olarak Spring MVC Temelleri dersinin "Embedded Tomcat ve spring-boot-starter-web" bölümünde gördüğümüz embedded Tomcat modeli) ve **"Disposability"** (bir servis örneği hızlıca başlayıp durabilmeli, çünkü ölçeklenme ve deploy bu döngüye dayanır).

Spring Boot, bu ilkelerin çoğunu varsayılan olarak destekler: `application.yml` yapılandırmayı kod dışına taşır, `${ORDERS_DB_PASSWORD}` gibi ifadelerle ortam değişkenlerinden okunabilir hâle gelir, ve embedded Tomcat sayesinde her servis kendi portunu kendi bağlar. Bu ders boyunca yazacağımız `application.yml` , bu ilkelerin somut bir uygulaması.

Bir Mikroservisin Giriş Noktası: @SpringBootApplication

Her Spring Boot uygulaması gibi, `order-service` 'in de tek bir giriş noktası var:

```
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);
    }
}
```

Bu sınıf, Spring MVC Temelleri dersinin "Embedded Tomcat ve spring-boot-starter-web" bölümünde gördüğümüz bu projenin kendi `LearningPlatformApplication`'ıyla **yapısal olarak birebir aynı** -- aynı `@SpringBootApplication`, aynı `SpringApplication.run(...)` çağrısı. Aradaki fark kodda değil: bu sınıf kendi `.jar`'ında, kendi process'inde, diğer hiçbir serviste olmadan çalışıyor. `@SpringBootApplication`'ın örtük component scanning'i, Component Scanning dersinde gördüğümüz gibi, bu paketteki (default package) `@RestController` / `@Service` işaretli sınıfları otomatik bulacak -- birazdan yazacağımız `OrderController` ve `OrderService` de dahil.

Kendi `application.yml`'i: Port, Uygulama Adı ve Veritabanı

Giriş noktası neredeyse hiçbir şey söylemiyor -- asıl kimlik `application.yml`'de:

```
# 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")
```

Üç satıra dikkat et: `server.port: 8081`, `order-service`'i `learning-platform`'un kendi `8080`'inden ve ileride kuracağımız `inventory-service`'in (`8082`) portundan ayırıyor -- aynı makinede üçü birden çalışabilir. `spring.application.name: order-service`, bu servisin **kimliği** -- her log satırında görünecek, ve ileride (Service Discovery / Eureka) diğer servislerin `order-service`'i bulmasını sağlayacak isim tam olarak bu. `spring.datasource.url`, "Microservices Temelleri" dersinin "Database per Service" bölümündeki kuralın somut hâli -- `orders_db`, yalnızca `order-service`'in bildiği bir veritabanı; `inventory-service` kurulduğunda o da kendi `inventory_db`'sine bağlanacak, ikisi arasında hiçbir paylaşım olmayacak.

> `password: ${ORDERS_DB_PASSWORD}` bilerek düz metin bir parola **değil** -- bir ortam değişkeninden okunuyor. Bu, Tarihçe bölümündeki "Config" ilkesinin doğrudan uygulaması: parolalar, API anahtarları gibi ortama göre değişen ve gizli kalması gereken değerler asla `application.yml`'e düz metin yazılmaz.

Dış Yüzey: REST Controller ile API Sözleşmesi

"Microservices Temelleri" dersinin "Bir Mikroservisin Anatomisi" bölümünde bahsettiğimiz API yüzeyini şimdi gerçek bir controller'la kuruyoruz:

```
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) {
    }
}
```

Bu iki endpoint (`POST /orders`, `GET /orders/{id}`), `order-service`'in dışarıya verdiği **sözleşmenin tamamı** -- başka bir servis ya da istemci, `order-service`'ten yalnızca bu iki yolla bir şey isteyebilir. Dikkat edersen `OrderController`'ın kendisi hiçbir iş kararı vermiyor (miktar sıfır ya da negatif olabilir mi, sipariş nasıl saklanır gibi sorulara cevap vermiyor) -- yalnızca isteği alıp `OrderService`'e devrediyor. Bu, Spring MVC Temelleri dersinin "Bir HTTP İsteğinin Yolculuğu: Request Lifecycle" bölümündeki Controller -> Service ayrımının aynısı; bir sonraki bölümde `OrderService`'in kendisini göreceğiz.

İş Mantığını Controller'dan Ayırmak: Service Katmanı

`OrderController`'ın devrettiği iş mantığı burada yaşıyor:

```
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));
    }
}
```

`quantity <= 0` kontrolü, `OrderController`'da değil burada -- "hangi sipariş geçerli" sorusunun cevabı yalnızca `OrderService`'in bileceği bir kural, tıpkı "Microservices Temelleri" dersinin "Servis Sınırlarını (Service Boundaries) Belirlemek" bölümündeki "bu servis hangi iş kararını tek başına verebilir?" sorusunun cevabı gibi. Gerçek bir `order-service`, siparişleri "Kendi `application.yml`'i: Port, Uygulama Adı ve Veritabanı" bölümünde gördüğümüz `orders_db`'de saklardı -- burada bir `ConcurrentHashMap` bu kalıcılığın yerini tutuyor, çünkü bu dersin odağı JPA/repository detayları değil (bunlar zaten Spring MVC kategorisinin REST API Design konusunda işlendi), controller/service ayrımının kendisi.

Domain Modeli: Bu Serviste "Sipariş" Ne Demek?

`OrderController` ve `OrderService`'in paylaştığı `Order` tipi:

```
// 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) {
}
```

Bu kadar kısa olması kasıtlı -- `order-service`'in "sipariş" derken bilmesi gereken her şey bu üç alan. "Microservices Temelleri" dersinin "Domain-Driven Design'a Kısa Bakış: Bounded Context" bölümünde gördüğümüz gibi, bu model yalnızca `order-service`'in **kendi bounded context'i** içinde anlamlı -- `inventory-service` ileride kendi "sipariş"e ihtiyaç duysa (örneğin stoktan düşülecek miktarı görmek için), bunu kendi amacına göre, tamamen farklı alanlarla modelleyebilir; iki servis arasında paylaşılan, ortak bir `Order` sınıfı **yok**.

Sağlık Kontrolü (Health Check): Servis Ayakta mı?

Tek bir servis çalışırken "ayakta mı?" sorusunu sormaya pek gerek duymazsın -- ama `order-service` gibi bağımsız çalışan bir servisi kimin izleyeceği (bir yük dengeleyici, ileride kuracağımız bir API Gateway, ya da Kubernetes gibi bir orkestratör) bu soruyu sürekli sormak zorunda. "Kendi `application.yml`'i: Port, Uygulama Adı ve Veritabanı" bölümündeki `management.endpoints.web.exposure.include: health` satırı, `spring-boot- starter-actuator` bağımlılığı projeye eklendiğinde `GET /actuator/health` isteğine yanıt veren bir endpoint'i **otomatik olarak** açar -- hiç kod yazmana gerek kalmadan:

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

`{"status":"UP"}`, `order-service`'in çalıştığını ve (Actuator'ın varsayılan health indicator'ları veritabanı bağlantısını da kontrol ettiği için) veritabanına ulaşabildiğini söylüyor. Servis çökmüşse ya da veritabanına bağlanamıyorsa, aynı istek `{"status":"DOWN"}` döner. Bu basit görünen endpoint, bu kategorinin ilerleyen olası bir konusu olan Observability'nin (ve API Gateway/Service Discovery gibi altyapı parçalarının, ki hangisinin ne zaman senden gerçek bir doğrulama isteyeceğimizi ayrıca konuşacağız) üzerine kurulacağı en temel yapı taşı.

Loglama ve Korelasyon: Hangi Log Hangi Servisten?

`learning-platform` gibi tek bir uygulamada, konsoldaki her log satırının nereden geldiği bellidir -- tek bir uygulama var. `order-service` ve `inventory-service` aynı anda çalışmaya başladığında, ikisi de kendi konsoluna (ya da kendi log dosyasına) yazar -- ama "Kendi `application.yml`'i: Port, Uygulama Adı ve Veritabanı" bölümündeki `spring.application.name: order-service` satırı sayesinde, doğru yapılandırılmış bir log formatı her satıra hangi servisten geldiğini ekleyebilir:

```
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"
```

Tek bir uygulamada bu ayrım gereksizdi -- "Microservices Temelleri" dersinin "Dağıtık Sistemlerin Getirdiği Yeni Zorluklar" bölümünde gördüğümüz gözlemlenebilirlik zorluğunun en basit hâli tam olarak bu: on servise yayılmış bir isteği takip etmek istediğinde, önce hangi log satırının hangi servisten geldiğini ayırt edebilmen gerekir. Bu, bu kategorinin ilerleyen olası bir konusu olan Observability'de (dağıtık izleme, correlation ID'ler ile) çok daha derinlemesine ele alınacak bir konu -- şimdilik bilmen gereken tek şey, `spring.application.name`'in yalnızca bir etiket değil, gelecekteki tüm izleme/loglama altyapısının üzerine kurulacağı temel kimlik olduğu.

Best Practices

- **Her mikroservise kendi `application.yml`'ini ver, hiçbirini paylaşma** -- "Kendi `application.yml`'i: Port, Uygulama Adı ve Veritabanı" bölümünde gördüğümüz gibi, port, uygulama adı ve veritabanı bağlantısı her serviste farklı olmalı.
- **Gizli değerleri (parola, API anahtarı) asla `application.yml`'e düz metin yazma** -- "Tarihçe" bölümündeki "Config" ilkesi gereği, bunlar her zaman ortam değişkenlerinden (`${...}`) okunmalı.
- **`spring.application.name`'i baştan, dikkatli seç** -- yalnızca bir log etiketi değil; "Loglama ve Korelasyon: Hangi Log Hangi Servisten?" bölümünde gördüğümüz gibi, ileride service discovery ve dağıtık izleme bu isme dayanacak.
- **Controller'ı ince tut, iş kurallarını service katmanına bırak** -- "İş Mantığını Controller'dan Ayırmak: Service Katmanı" bölümünde gördüğümüz `OrderController` / `OrderService` ayrımı, Spring MVC Temelleri dersindeki aynı prensibin mikroservis bağlamındaki hâli.
- **Domain modelini yalnızca kendi servisinin ihtiyacına göre tasarla, başka bir servisin modeline benzemesi gerekmez** -- "Domain Modeli: Bu Serviste 'Sipariş' Ne Demek?" bölümünde gördüğümüz gibi, her servisin kendi bounded context'i vardır.
- **Health check endpoint'ini en baştan aç** -- "Sağlık Kontrolü (Health Check): Servis Ayakta mı?" bölümünde gördüğümüz gibi, servis büyüdükçe eklenecek bir şey değil, ilk günden itibaren gereken bir temel.

Yaygın Hatalar

1. Birden fazla mikroservisi aynı `application.yml`'i (ya da aynı portu) paylaşacak şekilde kurmaya çalışmak. "Neden Var?" bölümünde gördüğümüz gibi bu, ikisini aynı anda çalıştırmayı imkânsız kılar ya da birinin verisinin diğerini kirletmesine yol açar.

2. Parolaları ya da bağlantı dizelerini `application.yml`'e düz metin yazmak. "Kendi `application.yml`'i: Port, Uygulama Adı ve Veritabanı" bölümündeki `${...}` söz dizimi tam

olarak bunu önlemek için var -- düz metin bir parola, dosya git'e gönderildiği anda sızmış demektir.

3. Controller'a iş mantığı yazmak ("bu servis hangi kararı verebilir" sorusunu Controller'da cevaplamak). "Dış Yüzey: REST Controller ile API Sözleşmesi" bölümündeki `OrderController`, isteği yalnızca `OrderService`'e devrediyor -- karar mantığını controller'a yazmak, Spring MVC Temelleri dersindeki Controller -> Service ayrımını bozar.

4. `spring.application.name`'i unutmak ya da rastgele/tutarsız isimlendirmek. "Loglama ve Korelasyon: Hangi Log Hangi Servisten?" bölümünde gördüğümüz gibi, bu isim yalnızca kozmetik değil -- ileride service discovery ve izleme bu isme dayanacak, tutarsız bir isimlendirme o altyapıyı da bozar.

5. Health check'i "sonra eklerim" diye ertelemek. Bir mikroservisin ayakta olup olmadığını dışarıdan anlamanın tek yolu bu endpoint -- "Sağlık Kontrolü (Health Check): Servis Ayakta mı?" bölümünde gördüğümüz gibi, servis tek başına çalışırken önemsiz görünse de, bir yük dengeleyici ya da orkestratör devreye girdiğinde vazgeçilmez hâle gelir.

Özet, Cheat Sheet ve Terimler Sözlüğü

Bir Spring Boot mikroservisini yapılandırmak, "Microservices Temelleri" dersindeki kavramsal anatomiyi (giriş noktası, kendi portu/kimliği, kendi veritabanı, API yüzeyi) somut dosyalara dönüştürmek. Öne çıkan noktalar:

- `@SpringBootApplication` ile giriş noktası, `learning-platform`'un kendi ana sınıfıyla yapısal olarak aynı -- fark, bağımsız çalışmasında
- `application.yml`, servisin kimliğini (`server.port`, `spring.application.name`) ve veri bağlantısını (`spring.datasource.url`) taşır -- hiçbiri başka bir serviste paylaşılmaz
- Gizli değerler (`${ORDERS_DB_PASSWORD}` gibi) ortam değişkenlerinden okunur, asla düz metin yazılmaz (Twelve-Factor App'in "Config" ilkesi)
- Controller (`OrderController`) dış yüzeyi, Service (`OrderService`) iş mantığını, domain modeli (`Order`) bu servise özgü veri şeklini temsil eder -- üçü ayrı sorumluluklar
- `GET /actuator/health`, `spring-boot-starter-actuator` ile otomatik gelen, servisin ayakta olup olmadığını dışarıya bildiren temel endpoint
- `spring.application.name`, yalnızca bir etiket değil -- ileride service discovery ve dağıtık loglama/izleme bu isme dayanır

Hızlı referans:

```
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, controller yalnızca isteği devreder
}

@Service
class OrderService {
    // ... iş mantığı ve kalıcılık burada yaşar
}
```

Terimler Sözlüğü

`application.yml` — Bir Spring Boot uygulamasının/servisinin portu, kimliği ve bağlantı bilgileri gibi ortama göre değişen ayarlarını taşıyan yapılandırma dosyası.

`server.port` — Bir servisin embedded sunucusunun (bu projede Tomcat) dinleyeceği port numarası.

`spring.application.name` — Bir servisin kimliği; loglarda görünür, service discovery ve dağıtık izleme bu isme dayanır.

Twelve-Factor App — 2011'de Heroku mühendislerince yayınlanan, bulut ortamında çalışan uygulamaların nasıl inşa edilmesi gerektiğine dair on iki ilkelik metodoloji.

Config (Twelve-Factor ilkesi) — Ortama göre değişen değerlerin (port, parola, bağlantı dizesi) koda gömülmek yerine ortam değişkenlerinden okunması gerektiğini söyleyen ilke.

Health check — Bir servisin ayakta ve çalışır durumda olup olmadığını dışarıya bildiren, genellikle `GET /actuator/health` gibi bir endpoint'le sunulan mekanizma.

Spring Boot Actuator — Health check gibi operasyonel endpoint'leri otomatik olarak sağlayan Spring Boot starter'ı.