Servisler Arası İletişim

order-service'in yanına ikinci bir mikroservis (inventory-service) eklemek ve ikisini Spring'in RestClient'ıyla senkron bir REST çağrısıyla konuşturmak: stok kontrolü, hata yönetimi (servis ayakta değilse ne olur), ve servisler arası DTO ayrımı. Kategorinin gerçek, çalıştırılabilir Pratik Proje'sine buradan ulaşılıyor.

Orta 22 dk

Son güncelleme: 18 Ağustos 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/inter-service-communication

---

Servisler Arası İletişim

"Microservices Temelleri" ve "Mikroservis Yapılandırma" derslerinde tek bir mikroservisi (`order-service`) baştan sona kurduk -- ama gerçek dünyada mikroservisler tek başına yaşamaz, birbirleriyle konuşur. Bu derste, ilk dalganın (wave) son konusunda, `order-service` 'in yanına bir `inventory-service` daha ekleyip ikisini sade, senkron bir REST çağrısıyla konuşturacağız: bir sipariş oluşturulmadan önce `order-service` , `inventory-service` 'e "bu üründen yeterli stok var mı?" diye soracak. Bu dersin sonunda, bu iki servisin gerçek, çalıştırılabilir hâlini içeren bir Pratik Proje de göreceksin -- ayrı, izole bir repoda (bkz. dersin sonundaki "Pratik Proje" bölümü).

Servisler Arası İletişim Nedir?

"Microservices Temelleri" dersinin "Bir Mikroservisin Anatomisi" bölümünde her mikroservisin kendi API yüzeyi olduğunu görmüştük -- ama bir API yüzeyinin var olma sebebi, başka birilerinin onu çağırması. "Servisler arası iletişim" derken kastedilen tam olarak bu: bir mikroservisin, kendi işini tamamlamak için başka bir mikroservisten veri istemesi ya da ona bir komut vermesi. Bu derste somut örneğimiz: `order-service` , yeni bir sipariş oluşturmadan önce, o üründen yeterli stok olup olmadığını `inventory-service` 'e sormalı -- bu bilgi `order-service` 'in kendi veritabanında yok, çünkü "Database per Service" ilkesi gereği (Microservices Temelleri dersi) stok bilgisi yalnızca `inventory-service` 'e ait.

Bu iletişim iki temel şekilde olabilir: **senkron** (bir servis diğerine bir istek gönderir ve yanıtı bekler -- bu dersin konusu) ya da **asenkron** (bir servis bir mesaj/olay yayınlar, kim dinliyorsa dinler, gönderen yanıt beklemez -- kursun ilerleyen olası bir konusu olan Event-Driven Architecture/Kafka'nın odağı). Bu derste yalnızca senkron REST çağrılarını ele alıyoruz; "Senkron vs Asenkron: Bu Derste Neyi Kapsıyoruz?" bölümünde bu ayrımı netleştireceğiz.

Neden Var?

`order-service`, "Mikroservis Yapılandırma" dersinde gördüğümüz gibi kendi başına tamamen bağımsız çalışabilir -- kendi portu, kendi veritabanı, kendi API'si. Ama bağımsız çalışabilmesi, HER kararı tek başına verebileceği anlamına gelmez. Bir sipariş için "yeterli stok var mı?" sorusunun cevabı `order-service`'in kendi verisinde yok -- bu bilgi `inventory-service`'in bounded context'ine ait ("Domain-Driven Design'a Kısa Bakış: Bounded Context" bölümü, Microservices Temelleri dersi). Monolitik bir mimaride bu, aynı process içinde bir metot çağrısı kadar basit olurdu; mikroservislerde bu çağrı artık ağ üzerinden, HTTP ile yapılmak zorunda -- "Servis Sınırlarını (Service Boundaries) Belirlemek" bölümünde çizdiğimiz sınırların doğal bir sonucu: sınırın bir tarafındaki servis, diğer taraftaki bilgiye ancak bir istek göndererek ulaşabilir.

Tarihçe

Servisler arası senkron iletişim için tarih boyunca birçok teknoloji denendi: 1990'ların CORBA ve Java RMI'ı (uzak bir nesnenin metodunu yerel bir metot gibi çağırmayı hedefleyen, karmaşık ve platform bağımlı RPC -- Remote Procedure Call -- mekanizmaları), 2000'lerin SOAP/WSDL'i (XML tabanlı, katı bir sözleşmeye dayanan ama ağır ve yavaş bir protokol) ve 2010'lardan itibaren REST-over-HTTP'nin (bu kursun REST API Tasarımı dersinde detaylıca gördüğümüz yaklaşım) baskın hâle gelmesi. REST'in kazanmasının temel sebebi karmaşıklık değil sadelik: HTTP zaten her yerde var, JSON insan tarafından okunabilir, ve bir servisin API'sini anlamak için özel bir istemci kütüphanesi kurmana gerek yok -- `curl` bile yeterli (tıpkı "Sağlık Kontrolü (Health Check): Servis Ayakta mı?" bölümündeki `curl http://localhost:8081/actuator/health` örneğinde gördüğümüz gibi, Mikroservis Yapılandırma dersi).

Bu dersteki `RestClient`, Spring Framework 6.1 (2023) ile gelen, senkron HTTP çağrıları için önerilen modern API -- eski `RestTemplate`'in (hâlâ çalışan ama artık "maintenance mode"da olan) yerini, reaktif `WebClient`'ın (ek bir `spring-webflux` bağımlılığı ve reaktif programlama modeli gerektiren) getirdiği karmaşıklık olmadan alıyor. `spring-boot-starter-web` (`order-service`'in zaten sahip olduğu tek bağımlılık) `RestClient`'ı otomatik olarak sağlıyor -- yeni bir bağımlılık eklemeye gerek yok.

Senkron vs Asenkron: Bu Derste Neyi Kapsıyoruz?

"Microservices Temelleri" dersinin "CAP Teoremine Kısa Bir Bakış" bölümü, tam olarak bu dersin senaryosunu önceden işaret etmişti: "order-service ile inventory-service arasındaki iletişim kesildiğinde, sistemin ne yapacağına dair bu tür kararlar, mikroservis mimarisinin kaçınılmaz bir parçasıdır." Bu derste `order-service`, bir sipariş oluşturmadan önce `inventory-service`'i senkron olarak çağırıyor ve yanıtı bekliyor -- yani Tutarlılık (Consistency)

tarafını seçiyoruz: `inventory-service` yanıt veremiyorsa, `order-service` siparişi oluşturmaz (bkz. "Ağ Güvenilmez: inventory-service Ayakta Değilse Ne Olur?" bölümü), stoktan emin olmadan asla ilerlemez.

Bunun bir alternatifi de var: `order-service`, siparişi hemen oluşturup "stok kontrolü bekleniyor" durumuna alabilir, `inventory-service`'e bir olay (event) yayınlayabilir, ve stok yetersizse siparişi sonradan iptal edebilirdi -- bu, Erişilebilirlik (Availability) tarafını seçen, asenkron bir yaklaşım olurdu (kursun ilerleyen olası bir konusu olan Event-Driven Architecture/Kafka'nın ve Distributed Transactions'ın -- Saga pattern gibi -- odağı). Bu ders bilinçli olarak senkron/tutarlılık tarafını seçiyor, çünkü konsept olarak daha basit ve mevcut REST API Tasarımı bilgisinin üzerine doğrudan inşa ediliyor; asenkron yaklaşımı kursun ilerleyen aşamalarında, farklı bir altyapıyla (mesaj kuyruğu) ele alacağız.

İkinci Servis: inventory-service

`order-service`'i nasıl kurduğumuzu "Mikroservis Yapılandırma" dersinde adım adım gördük -- `inventory-service` de birebir aynı iskeletle başlıyor:

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

// The second microservice this course builds -- inventory-service. Structurally identical
// to order-service's own entry point (see Spring Boot Microservice Basics' "A
// Microservice's Entry Point" section) -- what makes the two DIFFERENT services isn't this
// class, it's that each is built, deployed, and run as its own independent JAR, with its
// own application.yml (see InventoryServiceConfig.yml) and its own port.
@SpringBootApplication
public class InventoryServiceApplication {
    public static void main(String[] args) {
        SpringApplication.run(InventoryServiceApplication.class, args);
    }
}
```

Giriş noktası, "Bir Mikroservisin Giriş Noktası: @SpringBootApplication" bölümünde gördüğümüz `OrderServiceApplication` ile yapısal olarak birebir aynı -- şaşırtıcı değil, çünkü bir mikroservisi mikroservis yapan şey giriş noktasının kodu değil, bağımsız çalışması. Kimliği, kendi `application.yml`'inde:

```
# inventory-service's own application.yml -- its own port, its own identity, its own
# database, none of it shared with order-service. inventory-service knows NOTHING about
# order-service's existence: it just exposes a plain REST endpoint and has no idea who
# calls it. In this lesson, knowledge flows one way -- order-service calls inventory-
service,
# never the other way around.

server:
  port: 8082                          # never 8081 (order-service) or 8080 (learning-
platform)

spring:
  application:
    name: inventory-service
  datasource:
    url: jdbc:postgresql://localhost:5432/inventory_db   # its OWN database -- order-service
                                                            # never connects to this URL
                                                            # (Database per Service)
    username: inventory_user
    password: ${INVENTORY_DB_PASSWORD}   # read from the environment, never hardcoded

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

`server.port: 8082`, `order-service`'in `8081`'inden ve `learning-platform`'un kendi `8080`'inden ayrı -- üçü aynı makinede aynı anda çalışabilir. `spring.datasource.url`'deki `inventory_db`, `order-service`'in `orders_db`'sinden tamamen ayrı bir veritabanı -- "Database per Service" bölümünde gördüğümüz kuralın bu derste ikinci bir örneği.

inventory-service'in API'si: Stok Sorgulama

`inventory-service`'in dışa açtığı tek şey, bir ürünün stok durumunu sorgulayan tek bir endpoint:

```
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.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

// inventory-service's ENTIRE external contract: one endpoint, read-only. Same
// Controller -> Service split as order-service's OrderController -- the controller makes no
// decisions, it just asks InventoryService and translates the answer into an HTTP response.
@RestController
@RequestMapping("/inventory")
class InventoryController {

    private final InventoryService inventoryService;

    InventoryController(InventoryService inventoryService) {
        this.inventoryService = inventoryService;
    }

    @GetMapping("/{productName}")
    ResponseEntity<InventoryItem> findByProductName(@PathVariable String productName) {
        return inventoryService.findByProductName(productName)
                .map(ResponseEntity::ok)
                .orElseGet(() -> ResponseEntity.status(HttpStatus.NOT_FOUND).build());
    }
}
```

"Dış Yüzey: REST Controller ile API Sözleşmesi" bölümünde `OrderController`'da gördüğümüz aynı desen -- controller hiçbir karar vermiyor, isteği `InventoryService`'e devrediyor:

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

import java.util.Map;
import java.util.Optional;

// inventory-service's business logic. A real deployment would back this with its own
// database (inventory_db, see InventoryServiceConfig.yml); a fixed in-memory map stands in
// for that here, the same simplification order-service's OrderService already made -- this
// lesson's focus is the CALL between the two services, not persistence.
@Service
class InventoryService {

    private final Map<String, Integer> stock = Map.of(
            "Keyboard", 42,
            "Monitor", 7,
            "Mouse", 0
    );

    Optional<InventoryItem> findByProductName(String productName) {
        Integer quantity = stock.get(productName);
        return quantity == null ? Optional.empty() : Optional.of(new InventoryItem
(productName, quantity));
    }
}
```

Ve controller/service'in paylaştığı domain modeli:

```
// inventory-service's own view of "how much stock does a product have" -- deliberately NOT
// the same type as order-service's Order (see Spring Boot Microservice Basics' domain model
// section). Two services can both have an opinion about the same real-world product
// ("Keyboard") and model it completely differently, because each only needs to know what
// its OWN job requires.
record InventoryItem(String productName, int quantityInStock) {
}
```

`InventoryItem`, "Domain Modeli: Bu Serviste 'Sipariş' Ne Demek?" bölümünde `Order` için söylediğimiz her şeyin aynısını taşıyor -- yalnızca `inventory-service`'in bilmesi gereken iki alan (`productName`, `quantityInStock`). Dikkat et: burada da `productName` var, `order-service`'in `Order` tipinde de `productName` var -- ama bu iki ayrı alan, iki ayrı sınıfta. Bu tekrar kasıtlı; "Kendi Sözleşmen: StockCheckResponse Neden InventoryItem Değil?" bölümünde neden bu kadar önemli olduğunu göreceğiz.

order-service'ten inventory-service'e: RestClient ile Senkron Çağrı

`order-service`'in `inventory-service`'e açtığı tek pencere:

```
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;
import org.springframework.web.client.HttpClientErrorException;
import org.springframework.web.client.ResourceAccessException;
import org.springframework.web.client.RestClient;

// order-service's ONLY window into inventory-service -- everything OrderService needs to
// know about stock goes through this one method. The base URL is read from application.yml
// via @Value (see the Autoconfiguration & Properties lesson's "Injecting a Single Property
// with @Value" section) instead of being hardcoded, for the same "Config" reason
// order-service's own database URL isn't hardcoded (see Spring Boot Microservice Basics'
// "History" section).
@Component
class StockClient {

    private final RestClient restClient;

    StockClient(@Value("${services.inventory-service.url}") String inventoryServiceUrl) {
        this.restClient = RestClient.builder().baseUrl(inventoryServiceUrl).build();
    }

    StockCheckResponse checkStock(String productName) {
        try {
            return restClient.get()
                    .uri("/inventory/{productName}", productName)
                    .retrieve()
                    .body(StockCheckResponse.class);
        } catch (HttpClientErrorException.NotFound e) {
            // inventory-service is UP and answered -- it just doesn't know this product.
            // A well-formed "no" is not a failure, so this does NOT become an exception.
            return new StockCheckResponse(productName, 0);
        } catch (ResourceAccessException e) {
            // inventory-service never answered at all (timeout, connection refused, DNS
            // failure) -- a completely different kind of problem than the 404 above, and
            // one order-service's own code needs to be able to tell apart from it.
            throw new InventoryServiceUnavailableException("inventory-service is not
reachable", e);
        }
    }
}

// A deliberate, meaningful exception -- not the raw ResourceAccessException RestClient
// throws. Catching that at the boundary (see StockClient.checkStock above) and translating
// it into something order-service's own code can reason about keeps inventory-service's
// failure mode from leaking, unexplained, into order-service.
class InventoryServiceUnavailableException extends RuntimeException {
    InventoryServiceUnavailableException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

`RestClient.builder().baseUrl(...).build()`, "Tarihçe" bölümünde bahsettiğimiz modern, senkron istemci -- `.get().uri(...).retrieve().body(...)` zinciri, tıpkı `OrderController`'ın kendi isteklerini işlediği gibi (ama bu kez `order-service`, isteği alan değil gönderen taraf). `@Value("${services.inventory-service.url}")`, `inventory-service`'in adresini `order-service`

'in kodunda sabit kodlamak yerine `application.yml`'den okuyor -- Autoconfiguration & Properties dersinin "@Value ile Tekil Property Enjeksiyonu" bölümünde gördüğümüz mekanizmanın aynısı, "Config" ilkesinin (Mikroservis Yapılandırma dersinin "Tarihçe" bölümü) bir başka uygulaması. `order-service`'in `application.yml`'ine eklenen tek satır:

```
services:
  inventory-service:
    url: http://localhost:8082
```

Kendi Sözleşmen: StockCheckResponse Neden InventoryItem Değil?

`StockClient`'ın deserialize ettiği tip, `InventoryItem` değil, ayrı bir sınıf:

```
// order-service's OWN shape for "what inventory-service told us about a product" -- NOT the
// same class as inventory-service's InventoryItem. This lesson's two services don't share a
// single class, not even a DTO -- the field names happen to look similar here only because
// JSON keys have to match to deserialize; inventory-service is free to change InventoryItem
// internally without ever breaking this contract, exactly why a DTO is safer than
// deserializing straight into a shared/domain type (see the REST API Design lesson's "The
// DTO Pattern: Separating Request/Response with Records" section).
record StockCheckResponse(String productName, int quantityInStock) {
}
```

Bu, kopyala-yapıştır tembelliği değil -- REST API Tasarımı dersinin "Entity'yi Doğrudan Dışarı Vermenin Riskleri: Neden DTO?" bölümünde gördüğümüz mantığın servisler arası hâli. `InventoryItem`, `inventory-service`'in İÇ domain modeli -- yarın bir alan eklenebilir, bir alan kaldırılabilir, adı değişebilir, çünkü bu sınıf yalnızca `inventory-service`'in kendi kod tabanında yaşıyor. `StockCheckResponse` ise `order-service`'in, `inventory-service`'in JSON yanıtını NASIL yorumladığına dair KENDİ kararı -- bu iki sınıf şu an aynı iki alanı taşısa bile, aralarında hiçbir kod bağımlılığı yok, ikisi de kendi servisinde bağımsız olarak değişebilir. Eğer `order-service`, `InventoryItem`'ı doğrudan import edip kullansaydı (mikroservisler arasında bu zaten mümkün bile değil -- iki servis ayrı JAR, ayrı process, ayrı classpath), `inventory-service`'teki her değişiklik `order-service`'i de derlemekten alıkoyardı; tam olarak mikroservislerin kaçınmaya çalıştığı sıkı bağlılık (tight coupling).

Ağ Güvenilmez: inventory-service Ayakta Değilse Ne Olur?

"order-service'ten inventory-service'e: RestClient ile Senkron Çağrı" bölümünde gördüğün `StockClient.checkStock(...)` metodu iki farklı başarısızlığı BİLE BİLE birbirinden ayırıyor:

- **`HttpClientErrorException.NotFound` (HTTP 404):** `inventory-service` AYAKTA, isteği aldı, cevap verdi -- sadece o ürünü tanımıyor. Bu bir arıza değil, düzgün bir "hayır"; bu yüzden bir exception fırlatmak yerine `quantityInStock = 0` olan bir `StockCheckResponse` döndürülüyor.

- `ResourceAccessException` **(zaman aşımı, bağlantı reddi, DNS hatası):** `inventory-service` hiç cevap VERMEDİ -- servis çökmüş olabilir, ağ kopmuş olabilir, ya da o an yeniden başlıyor olabilir. Bu, "Microservices Temelleri" dersinin "Dağıtık Sistemlerin Getirdiği Yeni Zorluklar" bölümünde saydığımız ağ güvenilmezliği ve partial failure (kısmi arıza) risklerinin canlı hâli -- monolitte bir metot çağrısının neredeyse hiç başarısız olmayacağı bir yerde, burada gerçek bir olasılık.

İkinci durumda `StockClient`, ham `ResourceAccessException`'ı `order-service`'in geri kalanına sızdırmak yerine kendi anlamlı istisnasına (`InventoryServiceUnavailableException`) çeviriyor. Bu fark önemli: "Yaygın Hatalar" bölümünde göreceğimiz gibi, bir servisin çağırdığı başka bir servisin ham hatasını olduğu gibi yukarı fırlatmak, çağıran tarafın kodunu (ve onu kullanan herkesi) `inventory-service`'in iç detaylarına (hangi HTTP istemcisi kullanıldığı gibi) bağımlı hâle getirir.

Parçaları Birleştirmek: OrderService'i Güncellemek

`StockClient` artık hazır -- `OrderService`'in tek yapması gereken, bir sipariş oluşturmadan önce onu çağırmak:

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

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

// The same order-service, the same OrderController in front of it -- but now, before
// creating an order, it asks inventory-service whether enough stock exists. This one method
// is the whole lesson: a same-service business rule (quantity <= 0, already covered in
// Spring Boot Microservice Basics) sitting right next to a call to another service.
@Service
class OrderService {

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

    OrderService(StockClient stockClient) {
        this.stockClient = stockClient;
    }

    Order create(String productName, int quantity) {
        if (quantity <= 0) {
            throw new IllegalArgumentException("quantity must be positive");
        }
        // Unlike every other line in this method, this one is a network call -- it can be
        // slow, time out, or fail because a completely different process crashed (see the
        // Microservices Fundamentals lesson's "New Challenges Introduced by Distributed
        // Systems" section).
        StockCheckResponse stock = stockClient.checkStock(productName);
        if (stock.quantityInStock() < quantity) {
            throw new IllegalArgumentException("insufficient stock for " + productName);
        }
        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));
    }
}
```

"İş Mantığını Controller'dan Ayırmak: Service Katmanı" bölümünde gördüğümüz `quantity <= 0` kontrolü hâlâ burada, değişmedi -- yeni eklenen tek şey, `stockClient.checkStock(productName)` çağrısı ve onun döndürdüğü stok miktarının istenen miktardan az olup olmadığının kontrolü. Dikkat et: `OrderController` hiç değişmedi (bu dosyaya bu derste dokunmadık) -- `OrderService` constructor'ına yeni bir bağımlılık (`StockClient`) eklenmesi, Spring'in component scanning'i (Component Scanning dersi) sayesinde otomatik olarak `@Component` işaretli `StockClient` bean'ini bulup enjekte ediyor, `OrderController`'ın bundan haberi bile yok.

Best Practices

- **Senkron bir servis çağrısını her zaman bir sınıfın (bir "client") arkasına gizle, `RestClient`'ı iş mantığının içine dağıtma** -- "order-service'ten inventory-service'e: RestClient ile Senkron Çağrı" bölümündeki `StockClient`, `OrderService`'in `inventory-service`'in var olduğunu bile bilmesine gerek bırakmıyor.
- **Base URL'i asla sabit kodlama, `application.yml`'den oku** -- "order-service'ten inventory-service'e: RestClient ile Senkron Çağrı" bölümündeki `@Value` kullanımı, aynı "Config" ilkesinin (Mikroservis Yapılandırma dersi) bir başka uygulaması; farklı ortamlarda (yerel, test, production) `inventory-service`'in adresi değişebilir.
- **Servisler arası çağrılarda 404 ("bulunamadı") ile bağlantı hatasını ("ulaşılamadı") birbirinden ayır** -- "Ağ Güvenilmez: inventory-service Ayakta Değilse Ne Olur?" bölümünde gördüğümüz gibi, bunlar tamamen farklı anlamlar taşır ve farklı şekilde ele alınmalı.
- **Başka bir servisin JSON yanıtını asla doğrudan kendi domain modeline deserialize etme, ayrı bir DTO tanımla** -- "Kendi Sözleşmen: StockCheckResponse Neden InventoryItem Değil?" bölümünde gördüğümüz `StockCheckResponse`, iki servisin domain modelini birbirinden bağımsız tutuyor.
- **Bir servisin başka bir servise sabit bağımlılığı (senkron çağrı), o servisin Availability'sini de miras alır** -- `inventory-service` çökerse, bu derste `order-service` da sipariş oluşturamaz hâle gelir; bu, gerçek bir mimari trade-off (bkz. "Senkron vs Asenkron: Bu Derste Neyi Kapsıyoruz?" bölümü), göz ardı edilmemeli.

Yaygın Hatalar

1. Bir servisin domain modelini (`Order`, `InventoryItem` gibi) başka bir serviste doğrudan kullanmaya çalışmak (mümkün olsa bile). "Kendi Sözleşmen: StockCheckResponse Neden InventoryItem Değil?" bölümünde gördüğümüz gibi, bu iki servisi kod seviyesinde birbirine kilitler -- birini değiştirmek diğerini bozar.

2. `inventory-service`'in adresini `order-service`'in kodunda sabit kodlamak (`"http://localhost:8082"` gibi). "order-service'ten inventory-service'e: RestClient ile Senkron Çağrı" bölümündeki `@Value` kullanımı tam olarak bunu önlemek için var -- adres ortama göre değişir, kod değişmemeli.

3. Bir servisin çökmesini ("ulaşılamadı") o servisin normal bir cevabıyla ("bulunamadı") karıştırmak. "Ağ Güvenilmez: inventory-service Ayakta Değilse Ne Olur?" bölümünde gördüğümüz gibi, `StockClient` bu ikisini bilerek iki ayrı `catch` bloğunda ele alıyor -- ikisini aynı şekilde ele almak, örneğin bir üründen "her zaman 0 stok var" sanmak gibi yanlış sonuçlara yol açar.

4. Senkron bir servis çağrısını hiç `try/catch` olmadan bırakmak. `RestClient` bir `ResourceAccessException` fırlattığında, yakalanmazsa bu istisna doğrudan `OrderController`'a kadar yükselir ve isteği yapan istemci, `inventory-service`'in aslında ne olduğunu hiç

anlamayan, çıplak bir HTTP 500 alır -- "Ağ Güvenilmez: inventory-service Ayakta Değilse Ne Olur?" bölümündeki `InventoryServiceUnavailableException`'a çevirme adımı tam olarak bunu önlüyor.

5. Servisler arası her çağrıyı senkron yapmak, hangi verinin gerçekten "hemen" gerekli olduğunu hiç sorgulamamak. "Senkron vs Asenkron: Bu Derste Neyi Kapsıyoruz?" bölümünde gördüğümüz gibi, bu derste stok kontrolü senkron olmak ZORUNDA (sipariş oluşmadan ÖNCE bilinmeli) -- ama her servisler arası etkileşim böyle değildir; kursun ilerleyen olası konularında (Event-Driven Architecture) bunun alternatifini göreceğiz.

Özet, Cheat Sheet ve Terimler Sözlüğü

Bu derste `order-service`'in yanına, kendi portu (`8082`), kendi veritabanı (`inventory_db`) ve kendi API'siyle bağımsız ikinci bir mikroservis (`inventory-service`) ekledik, ve `order-service` 'ten `inventory-service`'e senkron bir REST çağrısı kurduk. Öne çıkan noktalar:

- İki bağımsız servis, `RestClient` ile senkron HTTP üzerinden konuşabilir -- ek bir bağımlılık gerekmez, `spring-boot-starter-web` yeterli
- Base URL her zaman `application.yml`'den (`@Value` ile) okunur, koda gömülmez
- Servisler arası çağrı bir "client" sınıfının (`StockClient`) arkasına gizlenir, iş mantığı (`OrderService`) başka bir servisin var olduğunu bile bilmez
- 404 ("bulunamadı") ile bağlantı hatası ("ulaşılamadı") birbirinden tamamen farklıdır ve ayrı ele alınmalıdır
- Bir servisin JSON yanıtı, çağıran servisin kendi DTO'suna (`StockCheckResponse`) deserialize edilir -- asla karşı servisin domain modeline değil
- Bu ders bilinçli olarak SENKRON iletişimi (Tutarlılık/Consistency tarafı) kapsıyor; asenkron (Erişilebilirlik/Availability tarafı) kursun ilerleyen olası bir konusu

Hızlı referans:

```
@Component
class StockClient {
    private final RestClient restClient;

    StockClient(@Value("${services.inventory-service.url}") String url) {
        this.restClient = RestClient.builder().baseUrl(url).build();
    }

    StockCheckResponse checkStock(String productName) {
        try {
            return restClient.get()
                    .uri("/inventory/{productName}", productName)
                    .retrieve()
                    .body(StockCheckResponse.class);
        } catch (HttpClientErrorException.NotFound e) {
            return new StockCheckResponse(productName, 0);
        } catch (ResourceAccessException e) {
            throw new InventoryServiceUnavailableException("unreachable", e);
        }
    }
}
```

Terimler Sözlüğü

Senkron iletişim — Bir servisin diğerine bir istek gönderip yanıtı beklediği iletişim şekli; bu dersin konusu.

Asenkron iletişim — Bir servisin bir mesaj/olay yayınlayıp yanıt beklemediği iletişim şekli; kursun ilerleyen olası bir konusu (Event-Driven Architecture/Kafka).

`RestClient` — Spring Framework 6.1 ile gelen, senkron HTTP istekleri için önerilen modern istemci API'si; `spring-boot-starter-web` içinde hazır gelir.

`ResourceAccessException` — `RestClient`'ın, karşı servise hiç ulaşamadığında (zaman aşımı, bağlantı reddi) fırlattığı istisna.

DTO (Data Transfer Object) — Bir servisin, başka bir servisin (ya da bir istemcinin) yanıtını yorumlamak için tanımladığı, o servisin kendi domain modelinden bağımsız sınıf.

CAP Teoremi — Dağıtık bir sistemin Tutarlılık, Erişilebilirlik ve Bölünme Toleransı'ndan aynı anda en fazla ikisini garanti edebileceğini söyleyen teorem (Microservices Temelleri dersi).

Pratik Proje

Bu kategoride (Microservices Temelleri, Mikroservis Yapılandırma, Servisler Arası İletişim) öğrendiğimiz kavramları bir arada kullanan, gerçek ve çalıştırılabilir bir örnek proje var: **Inter-Service Communication Demo**.

Proje iki bağımsız Spring Boot mikroservisi içeriyor -- `order-service` (port `8081`) ve `inventory-service` (port `8082`) -- ve `order-service`, bir sipariş oluşturmadan önce `RestClient` ile `inventory-service`'i senkron olarak çağırıp stok kontrolü yapıyor. Bilgisayarına indirip her iki servisi de ayrı ayrı çalıştırabilir, kodunu satır satır inceleyebilirsin:

```
git clone https://github.com/cdurgun/microservices-course-projects.git
cd microservices-course-projects/projects/inter-service-communication

# Terminal 1
cd inventory-service
mvn spring-boot:run

# Terminal 2 (ayrı bir terminalde)
cd order-service
mvn spring-boot:run
```

`microservices-course-projects` deposu, `react-course-projects`'in aksine npm workspaces kullanmıyor -- Maven'de bunun bir karşılığı yok, bu yüzden her proje kendi bağımsız `pom.xml`'iyle, kardeş klasörler (sibling folders) hâlinde tutuluyor; ortak bir kurulum adımı yok, her servis kendi başına derlenip çalıştırılıyor. Projenin kendi `README.md`'si, denemek için hazır `curl` komutları da dahil, daha fazla ayrıntı içeriyor.