Inter-Service Communication

Adding a second microservice (inventory-service) next to order-service and connecting the two with a synchronous REST call via Spring's RestClient: stock checking, error handling (what happens when a service is down), and separating DTOs between services. Links to the category's real, runnable Practical Project.

Intermediate 22 min

Last updated: August 18, 2026
Read online: https://www.learnforgex.com/en/topics/inter-service-communication

---

Inter-Service Communication

In "Microservices Fundamentals" and "Microservice Configuration" we built a single microservice (`order-service`) from the ground up -- but in the real world, microservices don't live alone, they talk to each other. In this lesson, the last topic of the first wave, we're adding a second service, `inventory-service`, next to `order-service`, and connecting the two with a simple, synchronous REST call: before creating an order, `order-service` will ask `inventory-service` "is there enough stock of this product?". By the end of this lesson you'll also see a real, runnable version of these two services -- in a separate, isolated repo (see the "Practical Project" section at the end of this lesson).

What Is Inter-Service Communication?

In "Microservices Fundamentals"'s "The Anatomy of a Microservice" section we saw that every microservice has its own API surface -- but the whole reason an API surface exists is for someone else to call it. "Inter-service communication" means exactly that: one microservice asking another microservice for data, or telling it to do something, in order to finish its own job. Our concrete example in this lesson: before creating a new order, `order-service` has to ask `inventory-service` whether there's enough stock of that product -- that information doesn't live in `order-service`'s own database, because under the "Database per Service" principle (Microservices Fundamentals) stock information belongs to `inventory-service` alone.

This kind of communication comes in two basic flavors: **synchronous** (one service sends a request to another and waits for the response -- this lesson's topic) or **asynchronous** (one service publishes a message/event, whoever is listening picks it up, and the sender never

waits for a response -- the focus of a possible later topic in this course, Event-Driven Architecture/Kafka). This lesson covers synchronous REST calls only; we'll draw that line clearly in "Synchronous vs. Asynchronous: What Does This Lesson Cover?".

Why Does It Exist?

`order-service`, as we saw in "Microservice Configuration", can run completely independently on its own -- its own port, its own database, its own API. But being able to run independently doesn't mean it can make EVERY decision on its own. The answer to "is there enough stock?" for an order doesn't live in `order-service`'s own data -- that information belongs to `inventory-service`'s bounded context (see "A Quick Look at Domain-Driven Design: Bounded Context", Microservices Fundamentals). In a monolithic architecture this would be as simple as a method call in the same process; in microservices, that call now has to happen over the network, over HTTP -- a direct consequence of the boundaries we drew in "Defining Service Boundaries": a service on one side of a boundary can only reach information on the other side by sending a request.

History

Many technologies have been tried, over the decades, for synchronous inter-service communication: CORBA and Java RMI in the 1990s (complex, platform-dependent RPC -- Remote Procedure Call -- mechanisms aiming to make calling a remote object's method feel like calling a local one), SOAP/WSDL in the 2000s (an XML-based protocol built on a rigid contract, but heavy and slow), and, from the 2010s onward, REST-over-HTTP (the approach this course covers in depth in REST API Design) becoming dominant. REST won not because it was more sophisticated, but because it was simpler: HTTP is already everywhere, JSON is human-readable, and you don't need a special client library just to understand a service's API -- even `curl` is enough (exactly like the `curl http://localhost:8081/actuator/health` example in "Health Checks: Is the Service Up?", Microservice Configuration).

The `RestClient` used in this lesson is the modern API Spring Framework 6.1 (2023) introduced for synchronous HTTP calls -- it replaces the older `RestTemplate` (still functional, but now in "maintenance mode") without the extra complexity the reactive `WebClient` brings (an additional `spring-webflux` dependency and a reactive programming model). `spring-boot-starter-web` (the only dependency `order-service` already has) provides `RestClient` automatically -- no new dependency needed.

Synchronous vs. Asynchronous: What Does This Lesson Cover?

"Microservices Fundamentals"'s "A Quick Look at the CAP Theorem" section already pointed straight at this lesson's scenario: "when communication between order-service and inventory-service breaks down, decisions about what the system does are an unavoidable part of microservice architecture." In this lesson, `order-service` calls `inventory-service`

synchronously before creating an order, and waits for the response -- in other words, we're choosing the Consistency side: if `inventory-service` can't answer, `order-service` does not create the order (see "The Network Is Unreliable: What If inventory-service Is Down?"), it never proceeds without being sure about stock.

There's an alternative: `order-service` could create the order immediately in a "stock check pending" state, publish an event for `inventory-service`, and cancel the order later if stock turns out to be insufficient -- that would be an asynchronous approach choosing the Availability side (the focus of possible later topics in this course, Event-Driven Architecture /Kafka, and Distributed Transactions -- with approaches like the Saga pattern). This lesson deliberately picks the synchronous/consistency side, because it's conceptually simpler and builds directly on the REST API Design knowledge you already have; we'll cover the asynchronous approach later in the course, with a different kind of infrastructure (a message queue).

The Second Service: inventory-service

We walked through setting up `order-service` step by step in "Microservice Configuration" -- `inventory-service` starts from the exact same skeleton:

```
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);
    }
}
```

The entry point is structurally identical to the `OrderServiceApplication` we saw in "A Microservice's Entry Point: @SpringBootApplication" -- unsurprising, since what makes a microservice a microservice isn't the code in its entry point, it's that it runs independently. Its identity lives in its own `application.yml`:

```
# 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` keeps it separate from `order-service`'s `8081` and `learning-platform`'s own `8080` -- all three can run on the same machine at the same time. The `inventory_db` in `spring.datasource.url` is a completely separate database from `order-service`'s `orders_db` -- a second concrete example of the rule from "Database per Service".

inventory-service's API: Checking Stock

The only thing `inventory-service` exposes to the outside world is a single endpoint for checking a product's stock level:

```
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());
    }
}
```

The same pattern we saw in `OrderController` in "The External Surface: A REST Controller as the API Contract" -- the controller makes no decisions at all, it just hands the request to `InventoryService`:

```
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));
    }
}
```

And the domain model the controller and service share:

```
// 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` carries the exact same idea we described for `Order` in "The Domain Model: What Does 'Order' Mean in This Service?" -- just the two fields `inventory-service` actually needs (`productName`, `quantityInStock`). Notice that `productName` shows up here too, and it also shows up in `order-service`'s `Order` type -- but these are two separate fields, in two separate classes. That duplication is deliberate; we'll see exactly why it matters in "Your Own Contract: Why StockCheckResponse Instead of InventoryItem?".

From order-service to inventory-service: A Synchronous Call with RestClient

The one window `order-service` opens into `inventory-service`:

```
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()` is the modern, synchronous client we mentioned in "History" -- the `.get().uri(...).retrieve().body(...)` chain, except this time `order-service` is the one SENDING a request instead of receiving one, like `OrderController` does. `@Value("${services.inventory-service.url}")` reads `inventory-service`'s address from

`application.yml` instead of hardcoding it in `order-service`'s code -- the same mechanism we saw in the Autoconfiguration & Properties lesson's "Injecting a Single Property with @Value" section, another application of the "Config" principle (see "History", Microservice Configuration). The single line added to `order-service`'s `application.yml`:

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

Your Own Contract: Why StockCheckResponse Instead of InventoryItem?

The type `StockClient` deserializes into isn't `InventoryItem` -- it's a separate class:

```
// 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) {
}
```

This isn't copy-paste laziness -- it's the inter-service version of the reasoning in REST API Design's "The Risks of Returning an Entity Directly: Why a DTO?" section. `InventoryItem` is `inventory-service`'s INTERNAL domain model -- a field can be added, a field can be removed, a name can change tomorrow, because that class only lives inside `inventory-service`'s own codebase. `StockCheckResponse`, on the other hand, is `order-service`'s OWN decision about how it interprets `inventory-service`'s JSON response -- even though the two classes happen to carry the same two fields right now, there's no code dependency between them at all, and each can change independently inside its own service. If `order-service` had imported and used `InventoryItem` directly (which isn't even possible between microservices in the first place -- two services are two separate JARs, two separate processes, two separate classpaths), every change in `inventory-service` would risk breaking `order-service`'s own build -- exactly the tight coupling microservices try to avoid.

The Network Is Unreliable: What If inventory-service Is Down?

The `StockClient.checkStock(...)` method you saw in "From order-service to inventory-service: A Synchronous Call with RestClient" deliberately separates two different kinds of failure:

- `HttpClientErrorException.NotFound` **(HTTP 404):** `inventory-service` is UP, it received the request, it answered -- it just doesn't recognize that product. This isn't a failure, it's a well-formed "no"; so instead of throwing an exception, a `StockCheckResponse` with `quantityInStock = 0` is returned.
- `ResourceAccessException` **(timeout, connection refused, DNS failure):** `inventory-service` never answered at all -- it might have crashed, the network might be down, or it might be restarting at that exact moment. This is the live version of the network-unreliability and partial-failure risks we listed in Microservices Fundamentals's "The New Challenges of Distributed Systems" section -- a real possibility here, in a place where a method call inside a monolith would almost never fail.

In the second case, `StockClient` translates the raw `ResourceAccessException` into its own meaningful exception (`InventoryServiceUnavailableException`) instead of letting it leak into the rest of `order-service` . That difference matters: as we'll see in "Common Mistakes", letting a raw error from a service you call bubble straight up ties the calling code (and everyone who uses it) to `inventory-service` 's internal details -- like which HTTP client it happens to use.

Putting the Pieces Together: Updating OrderService

`StockClient` is ready -- all `OrderService` has to do now is call it before creating an order:

```
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));
    }
}
```

The `quantity <= 0` check from "Separating Business Logic from the Controller: The Service Layer" is still right there, unchanged -- the only new thing is the call to `stockClient.checkStock(productName)` and checking whether the stock it returns is less than the requested quantity. Notice that `OrderController` didn't change at all (we never touched that file in this lesson) -- adding a new dependency (`StockClient`) to `OrderService`'s constructor is automatically picked up and injected by Spring's component scanning (Component Scanning lesson), which finds the `@Component`-annotated `StockClient` bean on its own; `OrderController` doesn't even know it exists.

Best Practices

- **Always hide a synchronous service call behind a class (a "client"), never scatter `RestClient` calls through your business logic** -- the `StockClient` in "From order-service to inventory-service: A Synchronous Call with RestClient" means `OrderService` doesn't even have to know `inventory-service` exists.
- **Never hardcode a base URL, always read it from `application.yml`** -- the `@Value` usage in "From order-service to inventory-service: A Synchronous Call with RestClient" is another application of the same "Config" principle (Microservice Configuration); `inventory-service`'s address can change across environments (local, test, production).
- **Tell a 404 ("not found") apart from a connection failure ("unreachable") in inter-service calls** -- as we saw in "The Network Is Unreliable: What If inventory-service Is Down?", these mean completely different things and need to be handled differently.
- **Never deserialize another service's JSON response directly into your own domain model -- define a separate DTO** -- the `StockCheckResponse` in "Your Own Contract: Why StockCheckResponse Instead of InventoryItem?" keeps the two services' domain models independent of each other.
- **A hard dependency on another service (a synchronous call) also inherits that service's Availability** -- in this lesson, if `inventory-service` goes down, `order-service` can no longer create orders either; that's a real architectural trade-off (see "Synchronous vs. Asynchronous: What Does This Lesson Cover?") that shouldn't be ignored.

Common Mistakes

1. Trying to use one service's domain model (`Order`, `InventoryItem`) directly inside another service, even where that's technically possible. As we saw in "Your Own Contract: Why StockCheckResponse Instead of InventoryItem?", this locks the two services together at the code level -- changing one breaks the other.

2. Hardcoding `inventory-service`'s address inside `order-service`'s code (something like `"http://localhost:8082"`). The `@Value` usage in "From order-service to inventory-service: A Synchronous Call with RestClient" exists specifically to prevent this -- the address changes with the environment, the code shouldn't have to.

3. Confusing a service being down ("unreachable") with that same service giving a normal answer ("not found"). As we saw in "The Network Is Unreliable: What If inventory-service Is Down?", `StockClient` deliberately handles these in two separate `catch` blocks -- treating them the same leads to wrong conclusions, like assuming a product "always has 0 stock."

4. Leaving a synchronous service call with no `try`/`catch` at all. When `RestClient` throws a `ResourceAccessException` and nothing catches it, the exception rises all the way to

`OrderController`, and the client making the request gets a bare HTTP 500 with no idea what actually happened to `inventory-service` -- the translation step in "The Network Is Unreliable: What If inventory-service Is Down?" exists specifically to prevent that.

5. Making every inter-service call synchronous without ever asking which data is actually needed "right now." As we saw in "Synchronous vs. Asynchronous: What Does This Lesson Cover?", the stock check in this lesson HAS to be synchronous (it must be known before the order exists) -- but not every inter-service interaction is like that; we'll see the alternative in a possible later topic in this course, Event-Driven Architecture.

Summary, Cheat Sheet, and Glossary

In this lesson we added a second, independent microservice (`inventory-service`) next to `order-service`, with its own port (`8082`), its own database (`inventory_db`), and its own API, and connected `order-service` to it with a synchronous REST call. Key points:

- Two independent services can talk over synchronous HTTP with `RestClient` -- no extra dependency needed, `spring-boot-starter-web` is enough
- The base URL always comes from `application.yml` (via `@Value`), never hardcoded into the code
- An inter-service call is hidden behind a "client" class (`StockClient`) -- the business logic (`OrderService`) doesn't even need to know another service exists
- A 404 ("not found") and a connection failure ("unreachable") are completely different and must be handled separately
- Another service's JSON response is deserialized into the calling service's own DTO (`StockCheckResponse`) -- never directly into the other service's domain model
- This lesson deliberately covers SYNCHRONOUS communication (the Consistency side); asynchronous (the Availability side) is a possible later topic in this course

Quick reference:

```
@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);
        }
    }
}
```

Glossary

Synchronous communication — Communication where one service sends a request to another and waits for the response; this lesson's topic.

Asynchronous communication — Communication where one service publishes a message /event and doesn't wait for a response; a possible later topic in this course (Event-Driven Architecture/Kafka).

`RestClient` — The modern client API Spring Framework 6.1 introduced for synchronous HTTP requests; included out of the box in `spring-boot-starter-web`.

`ResourceAccessException` — The exception `RestClient` throws when it can't reach the other service at all (timeout, connection refused).

DTO (Data Transfer Object) — A class a service defines to interpret another service's (or a client's) response, independent of its own domain model.

CAP Theorem — The theorem stating that a distributed system can guarantee at most two of Consistency, Availability, and Partition Tolerance at the same time (Microservices Fundamentals).

Practical Project

There's a real, runnable example project that brings together everything we learned in this category (Microservices Fundamentals, Microservice Configuration, Inter-Service Communication): **Inter-Service Communication Demo**.

The project contains two independent Spring Boot microservices -- `order-service` (port `8081`) and `inventory-service` (port `8082`) -- with `order-service` calling `inventory-service` synchronously via `RestClient` to check stock before creating an order. You can download it and run both services yourself, and read through the code line by line:

```
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 (a separate terminal)
cd order-service
mvn spring-boot:run
```

Unlike `react-course-projects`, the `microservices-course-projects` repo doesn't use npm workspaces -- Maven has no equivalent, so each project keeps its own independent `pom.xml` as a sibling folder; there's no shared setup step, each service is built and run on its own. The project's own `README.md` has more detail, including ready-to-run `curl` commands to try it out.