Microservices Fundamentals

The limits of monolithic architecture, the core characteristics of microservice architecture, defining service boundaries (bounded context), database per service, and a quick look at the CAP theorem and Conway's Law -- a conceptual introduction with no code.

Intermediate 20 min

Last updated: August 17, 2026
Read online: https://www.learnforgex.com/en/topics/microservices-fundamentals

---

Microservices Fundamentals

This is the first lesson of the new "Microservices" category in the `spring-boot` course. We won't write any code in this lesson -- unlike the lessons in the Spring Core and Spring MVC categories, the goal here isn't to demonstrate an API or a mechanism through code, but to understand, conceptually, *why* microservice architecture exists, *what problem* it solves, and *what new problems* it brings with it. In the next lesson of this category we'll configure our first single microservice with Spring Boot; in the lesson after that we'll see how two services talk to each other with a plain REST call, through a real, runnable example. The ideas in this lesson are the foundation those two lessons will build on.

What Are Microservices?

Microservices (microservice architecture) is an approach to building an application not as a single, large unit that's deployed all together, but as a collection of small, independent services, each of which **can be deployed on its own, can own its own database, and is responsible for a narrow, well-defined piece of the business.** These services usually talk to each other over the network (HTTP/REST, message queues, and similar) -- they are not classes calling each other directly through method calls within the same process; they are separate programs, deployed and run independently of one another.

Think of an e-commerce system: order management, inventory tracking, payment processing, and user accounts -- all of this could live inside a single application, a single codebase, a single deployment unit (we call this a "monolith," which we'll get into in the next section), or each could be a separate service with its own codebase, its own deployment cycle, and its own database (`order-service` , `inventory-service` , `payment-service` , `user-service`). Microservice architecture describes the second approach.

Why Do They Exist? (The Limits of the Monolith)

The easiest way to understand microservices is to understand the problem they're trying to solve -- and that problem barely shows up at all in small applications. For a small or medium-sized application, monolithic architecture (where all functionality lives in a single deployment unit) is usually the **right** choice: a single codebase, a single build, a single deployment -- simple and fast.

The trouble shows up as the application, and the team building it, **grows**. In a monolith, a small change (say, a fix to order-confirmation logic), even if it technically only concerns that one module, still requires the **entire** application to be rebuilt and redeployed -- and that carries the risk of breaking something that has nothing to do with the change. Dozens of developers working in the same codebase at the same time increases merge conflicts and uncertainty about "whose change broke what." And if one part (say, order creation) gets heavy traffic, scaling it requires scaling out the **whole** application horizontally -- even though, say, the reporting module might never see anywhere near that much traffic.

Microservices target exactly these three problems by splitting the application into independently deployable pieces: a change to one service only requires redeploying that service, different teams can work in different services without blocking each other, and only the service under heavy traffic needs to be scaled. In "Characteristics of Monolithic Architecture" and "Monolith or Microservices? Decision Criteria" we'll see that this trade-off is never free -- microservices bring their own new problems along with them.

History

The term "microservices" is newer than the underlying idea of small, independent services itself. The idea's roots go back to **SOA** (Service-Oriented Architecture) in the mid-2000s -- SOA also advocated splitting an application into services, but it typically communicated through a heavy, centralized "Enterprise Service Bus" (ESB), using complex enterprise standards (SOAP, the WS-* protocols).

The term "microservices," in the sense we use it today, was first used around 2011 at a software architects' workshop, but it really took off in **2014**, with the article "Microservices" published jointly by Martin Fowler and James Lewis -- unlike SOA's heavy, centralized infrastructure, this article emphasized **lightweight** communication mechanisms (usually plain HTTP/REST), each service owning its own database, and automated deployment processes (the forerunner of what we now call "CI/CD").

Around the same time, large-scale tech companies like Netflix and Amazon shared, in public talks, how they had gradually broken up their own massive monoliths into microservices -- turning the idea from an academic discussion into a concrete, industry-adopted practice. Today, tools like Spring Cloud (the ecosystem that Eureka and Spring Cloud Gateway -- which we'll cover later in this category -- belong to) and Kubernetes turn many of the problems that large companies used to solve by hand back then (service discovery, load balancing, deployment automation) into standard, off-the-shelf solutions.

Characteristics of Monolithic Architecture

Before we go further, let's be clear that "monolith" isn't a negative label -- monolithic architecture is a valid approach in its own right, defined by the following characteristics:

- **A single codebase:** All of the application's modules (order, inventory, payment, user, and so on) live in the same repository, in the same project structure -- exactly the way this project (`learning-platform`) itself has been built so far.
- **A single deployment unit:** The application is built and packaged as a single runnable unit (a `.jar`, a container image) and deployed that way.
- **A single process, direct method calls:** Communication between modules happens through direct Java method calls within the same process, not over the network -- no network latency, no network failures, no distributed transactions (we'll go into why this difference matters in "The New Challenges of Distributed Systems").
- **Usually a single, shared database:** All modules use the same database schema and can join its tables directly.

For many small and medium-sized applications, this is still the **right** starting point -- for its simplicity, its low operational overhead (keeping a single application running takes far less work than keeping dozens of services running), and its fast development cycle. In "'Modular Monolith': A Middle Ground" we'll look at an approach that keeps this simplicity while organizing the monolith's internals with more discipline.

Core Characteristics of Microservice Architecture

Microservice architecture argues for the opposite of all four characteristics from "Characteristics of Monolithic Architecture":

- **Independent deployability:** Each service can be built and deployed on its own, without the others knowing. A change to `order-service` doesn't require redeploying `inventory-service`.
- **A narrow, well-defined responsibility:** A well-designed microservice focuses on a single business capability -- we'll see how to define this in "Defining Service Boundaries".
- **Ownership of its own data:** Each service is the sole owner of its own data -- it never reaches directly into another service's database (see "Database per Service").
- **Independent scalability:** Only the service under heavy traffic needs to be scaled out, and only that one.
- **Technology diversity (polyglot):** In theory, different services could be written in different languages/frameworks -- in practice, every service in this project will be Spring Boot/Java, but that flexibility is something the architecture allows.
- **Team autonomy:** Each service can be owned by a separate team, developed and deployed at its own pace -- when we cover Conway's Law (in "Conway's Law") we'll see how this autonomy is tied to organizational structure.

None of this comes for free -- each of these characteristics also brings a new source of complexity, which we'll cover in "The New Challenges of Distributed Systems."

The Anatomy of a Microservice

Let's make this concrete: picture a service called `order-service`, responsible for orders. A service like this typically has:

- **Its own API surface:** The REST endpoints it exposes to the outside world (to other services or to a front end) -- for example `POST /orders`, `GET /orders/{id}`. This is the service's externally visible, agreed-upon "contract."
- **Its own business logic:** Rules about creating an order, updating an order's status, and so on -- inventory checks or payment logic do **not** live here, they live in the relevant services.
- **Its own database (or schema):** A database that holds order data, accessed only by `order-service`.
- **Its own runtime:** Its own process, its own port (say, `8081`), its own `application.yml` -- we'll set this up in a real Spring Boot project in the next lesson of this category.

Here's a simple sketch of how two services might sit side by side (this isn't code, just a conceptual sketch):

```
order-service (port 8081)          inventory-service (port 8082)
├── OrderController                ├── InventoryController
├── OrderService (business logic)  ├── InventoryService (business logic)
└── its own database (orders_db)   └── its own database (inventory_db)
```

Both are independent Spring Boot applications -- if one crashes or gets redeployed, the other keeps running unaffected (but as we'll see in "The New Challenges of Distributed Systems," that independence alone isn't enough for the moments when they actually need each other).

Defining Service Boundaries

The most critical, and most commonly mishandled, decision in adopting microservice architecture is deciding **where to split** the application. Draw the boundary in the wrong place, and, as we'll see in "Common Mistakes", you can lose all the benefits of microservices while keeping only the cost of a distributed system.

The right approach is to split the application not along technical layers (for example "all controllers in one service, all repositories in another" -- this is never the right split), but along **business capabilities**. "Order management," "inventory tracking," "payment processing" -- each of these is a meaningful responsibility in its own right, one that maps onto something the business actually does. A good service boundary answers this question clearly: "What business decisions can this service make on its own, without asking another

service?" `order-service` alone knows and applies the business rules around whether an order can be created (a minimum amount, say) -- but it doesn't know whether an item is in stock; for that, it asks `inventory-service`.

In the next section ("A Quick Look at Domain-Driven Design: Bounded Context") we'll tie this idea to a more formal framework -- Domain-Driven Design's concept of a "bounded context."

> A common rule of thumb: if you keep needing the word "and" to describe a "service" (like "the order-and-payment service"), that's usually a sign that the service is actually taking on two separate responsibilities.

A Quick Look at Domain-Driven Design: Bounded Context

Domain-Driven Design (DDD), an approach to turning complex business domains into software defined by Eric Evans in 2003, predates microservices by a long way -- but it's become one of the most commonly used tools for answering the question raised in "Defining Service Boundaries".

DDD's most important concept in this context is the **bounded context**: a sub-part of a large business domain within which a particular language and model stays consistent, with clear boundaries. For example, the word "product" might mean "how many units are in stock, in which warehouse" in the context of inventory, while in the context of orders it might mean "how many units were ordered, at what price" -- the same word can carry different meanings and different data models in different contexts. In a well-designed microservice architecture, each microservice usually maps onto one (or a few closely related) bounded context.

Another DDD concept, the **aggregate**, is a group of objects within a bounded context that must stay consistent together -- for example, an `Order` and its `OrderLine`s form an aggregate; one shouldn't change while leaving the other in an inconsistent state. We won't go deep into DDD in this course -- it's enough to know that, when the DDD world is asked "how do I decide on service boundaries?", its answer is "along bounded contexts."

Database per Service

This is perhaps the most frequently violated rule in microservice architecture: **each service is the sole owner of its own data, and never reaches directly into another service's database.** `order-service` cannot directly query `inventory-service`'s tables -- if it needs inventory information, it calls `inventory-service`'s API instead (we'll see this in the Inter-Service Communication lesson).

The reasoning connects directly back to "Defining Service Boundaries": if two services share the same database schema, the boundary between them only exists at the code level -- a schema change still affects both services, which is exactly the tight coupling we were trying to avoid in "Why Do They Exist? (The Limits of the Monolith)," just moved across two

separate processes instead of staying in one place. This situation is sometimes called a **"distributed monolith"** -- it carries all of the operational complexity of microservices without any of the monolith's advantages (a single, simple deployment unit), the worst of both worlds.

The database-per-service principle comes with its own cost, too: you can no longer join `order-service`'s and `inventory-service`'s data in a single SQL query, and there's no longer an instant, guaranteed consistency between the two services' data the way there was in a single, shared database -- instead, you work with **eventual consistency**. We'll dig deeper into this trade-off in "The New Challenges of Distributed Systems" and "A Quick Look at the CAP Theorem".

> "Database per service" doesn't have to mean one database server per service -- having each service own its own separate schema/database on the same PostgreSQL server is enough. What matters isn't the number of physical servers, it's **logical isolation**: no service ever reaching directly into another service's tables.

The New Challenges of Distributed Systems

So far we've looked at the problems microservices solve -- now let's look at the other side of the coin. The four characteristics we listed in "Characteristics of Monolithic Architecture" (a single codebase, a single deployment, direct method calls, a shared database) were also guarantees the monolith gave you **for free** -- none of them come for free once you move to microservices; you have to solve each one yourself, separately:

- **The network can be unreliable:** A method call within the same process almost never fails; but an HTTP call between services can time out, the network can drop, or the other service simply might not be up at that moment. Writing code that's resilient to this kind of failure is exactly why libraries like Resilience4j exist -- something we're considering covering later in this course.
- **Partial failure:** In a monolith, "the application either fully works or fully crashes." In a distributed system, some services can be up while others are down. You need to decide upfront what the rest of the system does in that situation -- return an error, or carry on with incomplete data.
- **Eventual consistency:** As we touched on in "Database per Service", data between services is no longer instantly consistent -- a moment after an order is created, the inventory service might not have updated yet. How a system manages this brief window of inconsistency is the focus of a possible later topic in this course, Distributed Transactions (through approaches like the Saga pattern).
- **Observability:** In a monolith, one log file is often enough; tracing a request that's spread across ten services requires centralized logging and distributed tracing.

- **Testing:** Testing a single service in isolation isn't any different from testing a module in a monolith, but testing the integration between services requires the other services to actually be running (or realistic mocks of them).

None of these challenges mean "don't use microservices" -- they just mean "know that new responsibilities come along with microservices." In "Monolith or Microservices? Decision Criteria" we'll look at how to weigh this trade-off.

A Quick Look at the CAP Theorem

Behind the idea of eventual consistency mentioned in "The New Challenges of Distributed Systems" is a result from distributed systems theory: the **CAP theorem**. It states that a distributed system can guarantee **at most two of** the following three properties at the same time (when a network partition happens):

- **Consistency:** Every read sees the most recent write (or an error) -- it never returns stale or inconsistent data.
- **Availability:** Every request gets a response, whether it succeeds or not -- the system is never left unresponsive.
- **Partition Tolerance:** The system keeps working even if the network connection between services is cut (even if parts of it can't see each other).

In the real world, network partitions (what Partition Tolerance addresses) can always happen -- so in practice, the choice is really between Consistency and Availability: when the network partitions, the system either sacrifices consistency (returning errors or waiting on some requests until it's sure the data is consistent) or sacrifices availability (responding right away, but possibly without being fully sure the data is current). Decisions like this, about what happens when the connection between `order-service` and `inventory-service` is cut, are an unavoidable part of microservice architecture -- a monolith never has to ask this question, because there's no such thing as a "network partition" within a single process.

Conway's Law

An observation put forward by computer scientist Melvin Conway in 1967 comes up constantly in microservices discussions: **"Organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations."** In simpler terms: software architecture tends to mirror the way the teams building it are organized.

The practical consequence for microservices is this: if a company has separate, independently operating teams -- an "Order Team," an "Inventory Team," and a "Payments Team" -- the architecture those teams produce naturally tends toward separate services like `order-service`, `inventory-service`, and `payment-service`, because each team wants to be

able to deploy its own service independently. The reverse is also true: a single, large, tightly coordinated team naturally tends to produce a single monolith, because they're already in constant, synchronous communication with each other.

Some organizations flip this relationship deliberately and use it on purpose -- this is called the **"Inverse Conway Maneuver"**: reorganizing the team structure first (into small, autonomous, end-to-end teams) in order to arrive at a desired architecture (say, independent microservices). The "team autonomy" point in "Core Characteristics of Microservice Architecture" is a direct extension of this idea.

"Modular Monolith": A Middle Ground

In "Characteristics of Monolithic Architecture" we saw that a monolith's simplicity is an advantage, and in "The New Challenges of Distributed Systems" we saw that microservices aren't free. Between the two, there's a middle ground that's become increasingly popular in recent years: the **modular monolith**.

The idea is this: keep the application as a single deployment unit, running in a single process -- but organize it internally, with the same discipline we learned in "Defining Service Boundaries", into modules with clear boundaries that never reach directly into each other's internals (only through defined interfaces). Each module lives in its own package, "owns" its own data (even if it's technically the same database, only its own tables), and calls between modules deliberately go through a narrow interface.

The appeal of this approach is that you never experience any of the network unreliability, partial failure, or eventual consistency problems microservices bring (you're still a single process, still using direct method calls) -- but if it later turns out you genuinely need to (a module's traffic or team grows), extracting that module into its own service is much easier than starting with microservices from day one, precisely because the boundaries are already clear. Many experienced architects prefer the order "start with a modular monolith, split into microservices when you actually need to" over "start with microservices from the beginning" -- we'll look at how to weigh that choice in the next section.

Monolith or Microservices? Decision Criteria

Putting the previous sections together, this isn't a question with one right answer -- it's a trade-off to weigh. Signals that lean toward microservices:

- The application, and the team building it, are genuinely large (dozens or hundreds of developers), and the deploy-coupling and merge-conflict problems from "Why Do They Exist? (The Limits of the Monolith)" are actually happening.
- Different modules have clearly different traffic/scaling needs (say, one module handles thousands of requests per second while another handles a few hundred a day).
- Different teams want to deploy at their own pace, without blocking each other (see "Conway's Law").

- Service boundaries are already clear -- either because they're being carved out of a modular monolith (see "Modular Monolith": A Middle Ground) or because the domain is already well understood.

Signals that lean toward a monolith (or a modular monolith):

- A small team (a handful of developers) -- there isn't the capacity to carry the operational overhead from "The New Challenges of Distributed Systems" (monitoring, deploying, and debugging a dozen separate services).
- The domain/business rules aren't settled yet, and it's not clear where the service boundaries would even go -- without a clear answer like the one in "Defining Service Boundaries", a microservice architecture split along the wrong lines can turn into a hard-to-reverse mistake, as we'll see in "Common Mistakes".
- The application is still small or medium-sized, and there's no real scaling or team-coordination problem yet.

In practice, many experienced architects recommend not starting with microservices from day one, but starting with the approach in "'Modular Monolith': A Middle Ground" and moving to microservices gradually, once a real need (a concrete pain point in scale, team size, or deployment frequency) actually shows up.

Best Practices

- **Draw service boundaries around business capabilities, not technical layers** -- see "Defining Service Boundaries". A split like "all controllers in one service" delivers none of the benefits of microservices.
- **Make sure each service is the sole owner of its own data** -- never reach directly into another service's database (see "Database per Service"); always ask for data you need through that service's API.
- **Don't adopt microservices just because it's trendy -- wait for a real signal.** Wait until the signals in "Monolith or Microservices? Decision Criteria" (team size, differing scaling needs, deployment conflicts) are actually happening.
- **If you're not sure, start with a modular monolith.** As we saw in "'Modular Monolith': A Middle Ground," moving from a monolith organized with clear boundaries to microservices is much easier than fixing service boundaries that were drawn wrong from the start.
- **Accept the new responsibilities distributed systems bring (network unreliability, eventual consistency, observability) up front** -- these aren't details you can put off for "later," they're part of the architecture itself (see "The New Challenges of Distributed Systems").
- **Keep your team structure aligned with the architecture you want.** As we saw in "Conway's Law", architecture and organizational structure already tend to mirror each other -- use that deliberately instead of ignoring it.

Common Mistakes

1. Building a "distributed monolith" -- splitting services apart but still sharing a database. As we saw in "Database per Service", this means taking on the operational cost of microservices (network, deployment complexity) without any of their benefits.

2. Drawing service boundaries along technical layers (say, "all data-access code in one service, all business logic in another"). This is the exact opposite of the business-capability-focused approach in "Defining Service Boundaries" -- nearly every request ends up bouncing back and forth between two services.

3. Starting with dozens of microservices for a small team, before there's a real need. Without the signals from "Monolith or Microservices? Decision Criteria" (real team /scale/deployment pressure), this just means paying the cost from "The New Challenges of Distributed Systems" with none of the benefit.

4. Ignoring the new challenges distributed systems bring (network errors, partial failure, eventual consistency), and assuming everything will always be instantly consistent and available the way it was in a monolith. As we saw in "A Quick Look at the CAP Theorem" and "The New Challenges of Distributed Systems," these assumptions don't hold in a distributed system.

5. Drawing service boundaries by hand-waving, after the application has already grown, without any real design process. Boundaries drawn without a framework like the one in "A Quick Look at Domain-Driven Design: Bounded Context" tend to keep shifting over time and end up constantly stepping on each other.

Summary and Glossary

Microservice architecture is an approach to splitting an application into independently deployable services that own their own data and have narrow responsibilities -- it aims to solve the deploy-coupling, merge-conflict, and uniform-scaling problems a monolith runs into as it grows, but in exchange, it brings new responsibilities like network unreliability, partial failure, and eventual consistency. Key points:

- A monolith is **not** a bad thing -- for many small and medium-sized applications it's still the right starting point; the problems only show up as the application and team grow
- Service boundaries are drawn around **business capabilities**, not technical layers -- Domain-Driven Design's **bounded context** is a commonly used framework for this
- **Database per service**: each service is the sole owner of its own data -- violating this leads to a "distributed monolith"
- Distributed systems bring new problems that never exist in a monolith: network unreliability, partial failure, and **eventual consistency** -- the theoretical basis for this is the **CAP theorem**
- **Conway's Law** says architecture tends to mirror organizational structure

- A **modular monolith** keeps a monolith's simplicity while organizing it internally with clear boundaries, making it easier to split into microservices later if needed

Decision checklist (see "Monolith or Microservices? Decision Criteria"):

- Is there a real team-size/deployment-conflict problem, or is it hypothetical?
- Do the modules genuinely have different scaling needs?
- Are the service boundaries clear, or is the domain still unsettled?
- Is there enough capacity to handle the new operational overhead (monitoring, deployment, debugging) a distributed system brings?

Glossary

Microservices (Microservice Architecture) — An approach to building an application out of independently deployable services, each owning its own data and responsible for a narrow piece of functionality.

Monolith — An architecture where all of an application's functionality lives in a single codebase and a single deployment unit.

Modular Monolith — A type of monolith that remains a single deployment unit but is organized internally with clear, disciplined module boundaries.

Business Capability — A meaningful responsibility an organization carries out on its own (e.g., order management, inventory tracking); the basic unit used to define service boundaries.

Bounded Context — In Domain-Driven Design, a sub-part of a domain within which a particular language and model stays consistent, with clear boundaries.

Database per Service — The principle that each microservice is the sole owner of its own data and never reaches directly into another service's database.

Distributed Monolith — An anti-pattern where an application has been split into services that still share a database or remain tightly coupled, carrying the cost of microservices without their benefits.

Eventual Consistency — The property of a distributed system where data becomes consistent a short time later, rather than instantly.

CAP Theorem — A theorem stating that a distributed system can guarantee at most two of Consistency, Availability, and Partition Tolerance at the same time during a network partition.

Conway's Law — The observation that the systems an organization designs tend to mirror that organization's communication structure.