Introduction to microservices

15-19 minutes

Last reviewed 2021-06-24 UTC

This reference guide is the first in a four-part series about designing, building, and deploying microservices. This series describes the various elements of a microservices architecture. The series includes information about the benefits and drawbacks of the microservices architecture pattern, and how to apply it.

1. Introduction to microservices (this document)
2. Refactoring a monolith into microservices
3. Interservice communication in a microservices setup
4. Distributed tracing in a microservices application

This series is intended for application developers and architects who design and implement the migration to refactor a monolith application to a microservices application.

Monolithic applications

A monolithic application is a single-tiered software application in which different modules are combined into a single program. For example, if you're building an ecommerce application, the application is expected to have a modular architecture that is aligned with object-oriented programming (OOP) principles. The following diagram shows an example ecommerce application setup, in which the application consists of various modules. In a monolithic application, modules are defined using a combination of programming language constructs (such as Java packages) and build artifacts (such as Java JAR files).

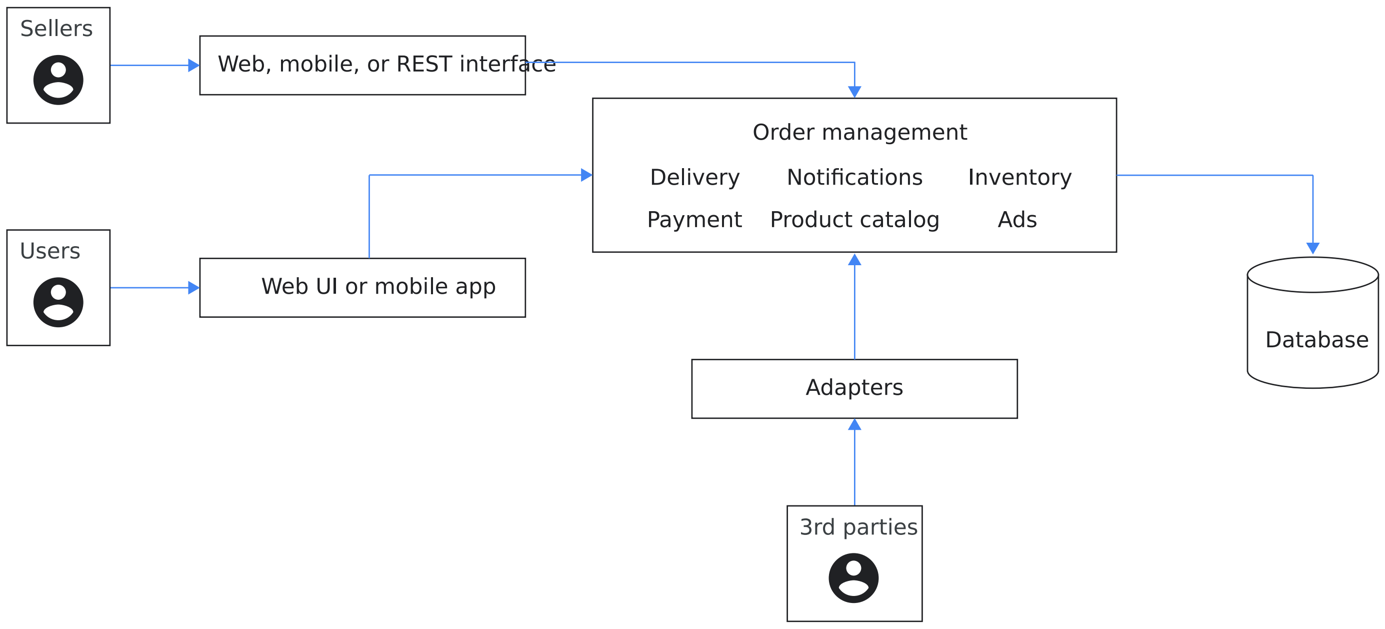

Figure 1. Diagram of a monolithic ecommerce application with several modules using a combination of programming language constructs.

In figure 1, different modules in the ecommerce application correspond to business logic for payment, delivery, and order management. All of these modules are packaged and deployed as a single logical executable. The actual format depends on the application's language and framework. For example, many Java applications are packaged as JAR files and deployed on application servers such as Tomcat or Jetty. Similarly, a Rails or Node.js application is packaged as a directory hierarchy.

Monolith benefits

Monolithic architecture is a conventional solution for building applications. The following are some advantages of adopting a monolithic design for your application:

- You can implement end-to-end testing of a monolithic application by using tools like Selenium.
- To deploy a monolithic application, you can simply copy the packaged application to a server.
- All modules in a monolithic application share memory, space, and resources, so you can use a single solution to address cross-cutting concerns such as logging, caching, and security.
- The monolithic approach can provide performance advantages, because modules can call each other directly. By contrast, microservices typically require a network call to communicate with each other.

Monolith challenges

Complex monoliths often become progressively harder to build, debug, and reason about. At some point, the problems outweigh the benefits.

- Applications typically grow over time. It can become complicated to implement changes in a large and complex application that has tightly coupled modules. Because any code change affects the whole system, you have to thoroughly coordinate changes. Coordinating changes makes the overall development and testing process much longer compared to microservice applications.
- It can be complicated to achieve continuous integration and deployment (CI/CD) with a large monolith. This complexity is because you must redeploy the entire application in order to update any one part of it. Also, it's likely that you have to do extensive manual testing of the entire application to check for regressions.
- Monolithic applications can be difficult to scale when different modules have conflicting resource requirements. For example, one module might implement CPU-intensive image-processing logic. Another module might be an in-memory database. Because these modules are deployed together, you have to compromise on the choice of hardware.
- Because all modules run within the same process, a bug in any module, such as a memory leak, can potentially bring down the entire system.
- Monolithic applications add complexity when you want to adopt new frameworks and languages. For example, it is expensive (in both time and money) to rewrite an entire application to use a new framework, even if that framework is considerably better.

Microservices-based applications

A microservice typically implements a set of distinct features or functionality. Each microservice is a mini-application that has its own architecture and business logic. For example, some microservices expose an API that's consumed by other microservices or by the application's clients, such as third-party integrations with payment gateways and logistics.

Figure 1 showed a monolithic ecommerce application with several modules. The following diagram shows a possible decomposition of the ecommerce application into microservices:

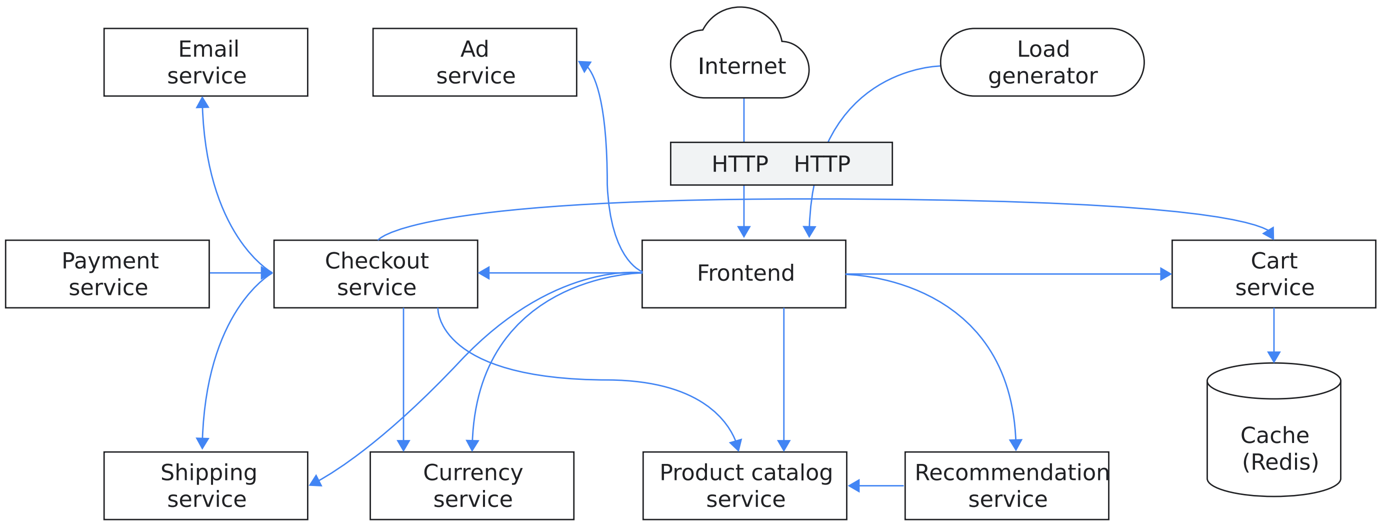

Figure 2. Diagram of an ecommerce application with functional areas implemented by microservices.

In figure 2, a dedicated microserve implements each functional area of the ecommerce application. Each backend service might expose an API, and services consume APIs provided by other services. For example, to render web pages, the UI services invoke the checkout service and other services. Services might also use asynchronous, message-based communication. For more information about how services communicate with each other, see the third document in this series, Interservice communication in a microservices setup.

The microservices architecture pattern significantly changes the relationship between the application and the database. Instead of sharing a single database with other services, we recommend that each service have its own database that best fits its requirements. When you have one database for each service, you ensure loose coupling between services because all requests for data go through the service API and not through the shared database directly. The following diagram shows a microservices architecture pattern in which each service has its own database:

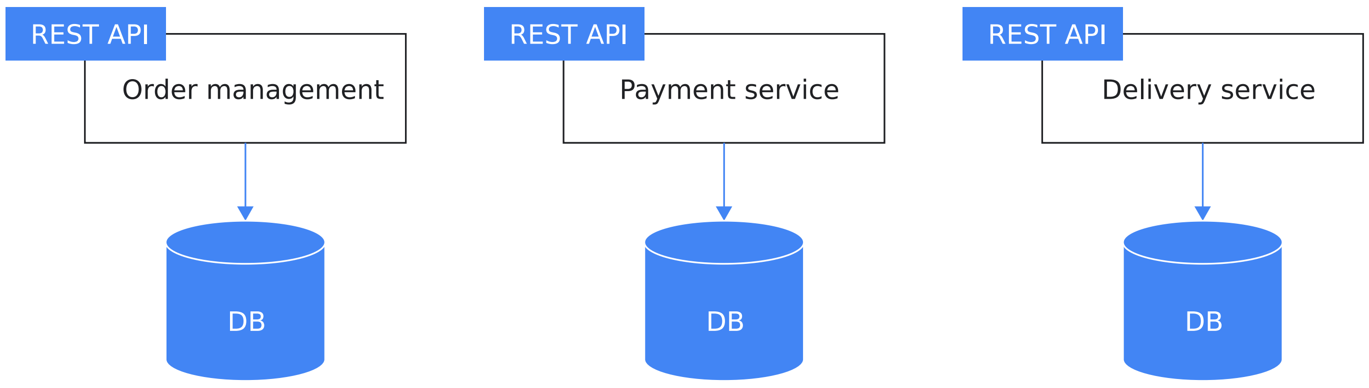

Figure 3. Each service in a microservice architecture has its own database.

In figure 3, the order service in the ecommerce application functions well using a document-oriented database that has real-time search capabilities. The payment and delivery services rely on the strong atomicity, consistency, isolation, durability (ACID) guarantees of a relational database.

Microservices benefits

The microservices architecture pattern addresses the problem of complexity described in the preceding Monolith challenges section. A microservices architecture provides the following benefits:

- Although the total functionality is unchanged, you use microservices to separate the application into manageable chunks or services. Each service has a well-defined boundary in the form of an RPC or message-driven API. Therefore, individual services can be faster to develop, and easier to understand and maintain.
- Autonomous teams can independently develop individual services. You can organize microservices around business boundaries, not the technical capabilities of a product. You organize your teams for a single, independent responsibility for the entire lifecycle of their assigned piece of software from development to testing to deployment to maintenance and monitoring.
- Independent microservice development process also lets your developers write each microservice in a different programming language, creating a polyglot application. When you use the most effective language for each microservice, you can develop an application more quickly and optimize your application to reduce code complexity and to increase performance and functionality.
- When you decouple capabilities out of a monolith, you can have the independent teams release their microservice independently. Independent release cycles can help improve your teams' velocity and product time to market.
- Microservices architecture also lets you scale each service independently. You can deploy the number of instances of each service that satisfy its capacity and availability constraints. You can also use the hardware that best matches a service's resource requirements. When you scale services independently, you help increase the availability and the reliability of the entire system.

The following are some specific instances in which it can be beneficial to migrate from a monolith to a microservice architecture:

- Implementing improvements in scalability, manageability, agility, or speed of delivery.
- Incrementally rewriting a large legacy application to a modern language and technology stack to meet new business demands.
- Extracting cross-cutting business applications or cross-cutting services so that you can reuse them across multiple channels. Examples of services you might want to reuse include payment services, login services, encryption services, flight search services, customer profile services, and notification services.
- Adopting a purpose-built language or framework for a specific functionality of an existing monolith.

Microservices challenges

Microservices have some challenges when compared to monoliths, including the following:

- A major challenge of microservices is the complexity that's caused because the application is a distributed system. Developers need to choose and implement an inter-services communication mechanism. The services must also handle partial failures and unavailability of upstream services.

- Another challenge with microservices is that you need to manage transactions across different microservices (also referred to as a *distributed transaction*). Business operations that update multiple business entities are fairly common, and they are usually applied in an atomic manner in which either all operations are applied or everything fails. When you wrap multiple operations in a single database transaction, you ensure atomicity.

  In a microservices-based application, business operations might be spread across different microservices, so you need to update multiple databases that different services own. If there is a failure, it's non-trivial to track the failure or success of calls to the different microservices and roll back state. The worst case scenario can result in inconsistent data between services when the rollback of state due to failures didn't happen correctly. For information about the various methodologies to set up distributed transactions between services, see the third document in this series, Interservice communication in a microservices setup.

- Comprehensive testing of microservices-based applications is more complex than testing a monolithic application. For example, to test the functionality of processing an order in a monolithic ecommerce service, you select items, add them to a cart, and then check out. To test the same flow in a microservices-based architecture, multiple services - such as frontend, order, and payment - call each other to complete the test run.

- Deploying a microservices-based application is more complex than deploying a monolithic application. A microservice application typically consists of many services, each of which has multiple runtime instances. You also need to implement a service discovery mechanism that enables a service to discover the locations of any other services it needs to communicate with.

- A microservices architecture adds operations overhead because there are more services to monitor and alert on. Microservice architecture also has more points of failure due to the increased points of service-to-service communication. A monolithic application might be deployed to a small application server cluster. A microservices-based application might have tens of separate services to build, test, deploy and run, potentially in multiple languages and environments. All of these services need to be clustered for failover and resilience. Productionizing a microservices application requires high-quality monitoring and operations infrastructure.

- The division of services in a microservice architecture allows the application to perform more functions at the same time. However, because the modules run as isolated services, latency is introduced in the response time due to network calls between services.

- Not all applications are large enough to break down into microservices. Also, some applications require tight integration between components—for example, applications that must process rapid streams of real-time data. Any added layers of communication between services may slow real-time processing down. Thinking about the communication between services beforehand can provide helpful insights in clearly marking the service boundaries.

When deciding whether microservice architecture is best for your application, consider the following points:

- Microservice best practices require per-service databases. When you do data modeling for your application, notice whether per-service databases fit your application.
- When you implement a microservice architecture, you must instrument and monitor the environment so that you can identify bottlenecks, detect and prevent failures, and support diagnostics.
- In a microservice architecture, each service has separate access controls. To help ensure security, you need to secure access to each service both within the environment and from external applications that consume its APIs.
- Synchronous interservice communication typically reduces the availability of an application. For example, if the order service in an ecommerce application synchronously invokes other services upstream, and if those services are unavailable, it can't create an order. Therefore, we recommend that you implement asynchronous, message-based communication.

When to migrate a monolithic application to microservices

If you're already successfully running a monolith, adopting microservices is a significant investment cost for your team. Different teams implement the principles of microservices in different ways. Each engineering team has unique outcomes for how small their microservices are, or how many microservices they need.

To determine if microservices are the best approach for your application, first identify the key business goals or pain points you want to address. There might be simpler ways to achieve your goals or address the issues that you identify. For example, if you want to scale your application up faster, you might find that autoscaling is a more efficient solution. If you're finding bugs in production, you can start by implementing unit tests and continuous integration (CI).

If you believe that a microservice approach is the best way to achieve your goals, start by extracting one service from the monolith and develop, test, and deploy it in production. For more information, see the next document in this series, Refactoring a monolith into microservices. After you have successfully extracted one service and have it running in production, start extraction of the next service and continue learning from each cycle.

The microservice architecture pattern decomposes a system into a set of independently deployable services. When you develop a monolithic application, you have to coordinate large teams, which can cause slow software development. When you implement a microservices architecture, you enable small, autonomous teams to work in parallel, which can accelerate your development.

In the next document in this series, Refactoring a monolith into microservices, you learn about various strategies for refactoring a monolithic application into microservices.

What's next

- Read the next document in this series to learn about application refactoring strategies to decompose microservices.
- Read the third document in this series to learn about interservice communication in a microservices setup.
- Read the fourth, final document in this series to learn about distributed tracing of requests between microservices.