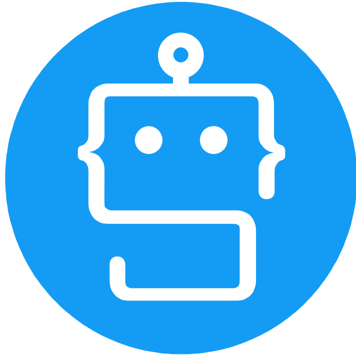
Microservices Guide

Stoplight — Read time: 12 minutes



Microservices Guide

Characteristics of microservices, best practices of microservice architecture, and tips for developers.

Over the last few decades, software development has grown more and more modular. One of the latest developments in this area has been the widespread adoption of microservices architectures.

Software was originally built in a monolithic fashion, but developers quickly discovered two key limitations to this approach. First, a single error can take down an entire monolithic application. Second, a monolithic approach lacks a straightforward approach for communication between different applications.

Tech giants including Amazon, Apple, Microsoft, and Netflix have embraced microservices. As such, most developers will find it in their best interest to be comfortable working with a microservice-based approach.

This guide will explain what microservices are and aren't, as well as offer some architectural advice and design-first principles that will lead to building better microservices.

What are microservices?

If you ask a group of developers what microservices are, you'll likely get many different answers. There's no industry-wide universal definition of microservices. But you'll find one of the most concise descriptions in Sam Newman's book, "Building Microservices."

Some companies define microservices as an architectural pattern used to build and structure applications as a collection of loosely coupled, small services. Each service is treated separately and can be reused within the company or externally by third-party developers.

A microservices architecture can consist of dozens, hundreds, or even thousands of individual services. And because each service is autonomous, you can add new features and fix problems without the risk of breaking the entire application.

What are the characteristics of microservices?

Microservices enable developers to build distributed applications that are reliable, scalable, and agile. To achieve this, microservices have the following key characteristics:

- Organized by business capabilities
- Loosely coupled
- Language agnostic

- Lightweight communication
- Decentralized

In the following sections, we'll go into detail of each of these characteristics, as well as explain how microservices differ from other approaches.

Services organized by business capabilities

In a microservices architecture, each service being invoked has its own autonomous code. Services are typically organized by business capabilities, with each independent service focusing on a specific business function. For example, if you were building a customer-facing application for a bank, you would add a microservice for each banking function:

- Open a checking account
- View account balances
- Download account transactions

If the bank is in Europe, it might provide Open Banking APIs that you could incorporate into an approved third-party financial services app.

Microservice architecture does not have to be limited to consumer-facing applications. The same approach could be used, for example, for tasks in a CI/CD pipeline:

- Source code control
- Integration testing
- Load testing
- Containerization

Each microservice represents a different business capability and each service is autonomous.

Loosely coupled, autonomous services

Netflix is well known for its use of microservices. And the company had a good reason for making the transition: When the Netflix website was a monolithic application, back in 2008, the entire site went down because of one missing semicolon.

Before the advent of microservices, applications were built as monoliths. When an application is a monolith, every service is tightly coupled together, and the entire app is treated as a single unit. One minor mistake anywhere in your tightly coupled monolithic application could break it. Incidents like this vividly illustrate the value of a more modular approach.

The basic idea of microservices is that the services are independent and autonomous—a problem with one service shouldn't impact other services or break your application. To realize the full value of autonomous services, it's important to minimize dependencies, and any dependencies your app does require should be as small as possible. There is nothing about a microservice-based approach that magically prevents dependencies, however, so it is important to be mindful of potential dependency issues from the earliest stages of the design process.

Language agnostic

Microservices don't require a specific programming language. This language independence lets you build each service with the best language, library, or framework for the job. It also allows you to experiment with cutting-edge technologies without the worry that you'll break the application.

Language flexibility is useful when you have multiple teams with different language specialties. It also means you aren't limited to one method of communication among services. But you should ensure that language independence is not taken to an extreme. If every service uses a different technology, maintaining the app can be unnecessarily challenging.

Lightweight communication

Every service in a microservices architecture must communicate with the others. Microservices usually rely on APIs for communication because APIs are language-independent and often lightweight. A microservice might use HTTP, REST, or WebSockets to communicate with other services within an app.

Decentralized data

When it comes to data management, each microservice generally handles its own database. Each service may use the same database technology but with different instances, or your app might take an approach called polyglot persistence, where each service adopts the database technology that best meets its needs.

The key to microservices is that the services don't share the same database or one persistent store. Monolithic database design has the same problem that monolithic code has: if any portion is unavailable, the whole thing is unavailable. As well, a decentralized data structure can improve your app's security and privacy by keeping unrelated data fully separate—a hotel's guests' personal data is fully segregated from its online advertising campaigns.

What microservices aren't

Now that we've established what microservices are, let's clarify what they're not.

First of all, microservices aren't always small. A microservice focuses on one task, and its size depends on what it does. "Micro" refers to the scope of the service, not its number of endpoints or lines of code.

Microservices also aren't two things they're commonly confused with: APIs or SOAs.

APIs vs. microservices

Microservices and APIs are related concepts, and may work together, but they are not the same thing. An API is a protocol for communication—a way of making a service available to use, or to make it available for consumption. The term "API" is also used, however, to mean an implementation of an API, the code and infrastructure used to deploy it. It is increasingly likely—but by no means required—that that infrastructure will include one or more microservices.

Additionally, microservices often use APIs to communicate with one another within an app. See why this can be confusing?

APIs and microservices may overlap, but an API is not necessarily implemented as a microservice (it does not need to be restricted to a narrow function), and a microservice is not explicitly concerned with a communication format (it may be served by an API, or by some other method).

SOAs vs. microservices

Microservices is not another name for service-oriented architecture (SOA). They are different things that share a few of the same characteristics. New approaches to software development often build on what has come before, and this is definitely true of microservices, which are in many ways a descendant of service-oriented architecture (SOA).

As with microservices, if you ask different developers what SOA is, you'll get different answers. In general, though, SOA focuses on reconsidering monolithic applications as separate services, just as microservices do. SOA also enables the reuse of individual components and organizes applications based on business functions.

The key differences between SOA and microservices are scope and communication. SOA is an enterprise-wide approach to building applications, where developers can reuse functions from one app to another across the enterprise, while microservices enable an application to be broken into individual business functions that can be scaled, reused, and administered independently. When it comes to communication, every microservice has its own protocol. However, each service of an SOA application shares an ESB (enterprise service bus) for communication. If the ESB fails, then the services can't communicate with each other, and the application breaks.

Key decisions for microservices architectures

Understanding the basic principles of microservices is a great first step toward using them successfully in your application. But before you start using them to build applications, you need to make some decisions regarding application architecture.

Monolith or microservices first?

If your team isn't experienced with building microservices architectures, sticking with the monolith approach might seem like a convenient option. If you are confident that a particular API will stay within a limited scope, a monolithic application may serve your needs effectively for the long term. The components of a monolith will eventually become tightly coupled together, however, meaning that breaking a monolith into services later will be very challenging.

If you're looking to add cutting-edge features to your application, or anticipate that the scale of the application may grow over time, then starting with a microservices architecture may be the best choice, as it will help you remain nimble even as complexity grows. In addition, microservices are language agnostic, so you can use the best language, framework, and library for each feature you want to add to your app, including incorporating new technologies as they emerge. By contrast, monoliths tend to use only a few languages, and migrating to new languages and frameworks is a major undertaking.

The choice between monolith and microservices is an essential design question, as it requires a clear picture of your long-term goals for your API programs. A design-first approach can provide a sense of security when venturing into a new architectural style—as well as prevent you from making choices you'll come to regret later.

Microservice interfaces

In general, most of the services in a microservices architecture are private and not made available for public consumption. However, you may want to expose some of the services as public or partner APIs. You'll need to decide on an API strategy to accompany your microservices, one that provides developers a good experience when consuming your APIs, and also the necessary security.

While each microservice focuses on one specific task, your public API might provide multiple capabilities: numerous microservices exposed as one mega public API. This a great example of how APIs and microservices are often related but not synonymous—and in fact, are often used together.

One popular approach to hiding a group of microservices behind a single entry point is an API gateway. For example, you might provide an API gateway for all the information an application's user interface might need. One of the key benefits of an API gateway is security: it provides advanced security features such as API user authentication, authorization, and data encryption. For prototype products, it might make sense to use something simpler, such as an API proxy, but for production-level applications, the increased overhead of an API gateway is usually worth it for the increased security.

Microservices design best practices

Once you've made the high-level architectural decisions, you can start designing your microservices and building your apps. Here are four microservices best practices to observe as you do.

1. Follow design-first principles

Building a microservices architecture means that you need to design and deploy dozens, hundreds, or perhaps even thousands of APIs. As such, you should follow design-first principles to ensure the success of all the APIs in your architecture. This means writing an API specification that all stakeholders review and offer feedback on.

If you use OpenAPI to create your API specification, you can iron out all the details of your APIs before writing a single line of code. And with Stoplight, you

can do nifty things with your OpenAPI files like generate mock servers to test API responses, create style guides, and generate documentation.

2. Maintain consistency across microservices

Microservices and the APIs associated with them are independent, but you should encourage consistency across your application. To do that, you need a consistent design style, one that every development team follows.

Creating a style guide is one of the best ways to ensure that microservices and APIs remain consistent across your entire application. You can use OpenAPI to create a style guide to suggest how endpoints, requests, responses, and fields should be implemented. And you can design contracts for your APIs that explain how the APIs will behave. You could also use a linting tool like Spectral to enforce the style guide by linting your OpenAPI documents.

3. Reuse common components

You can save yourself a lot of time by reusing components across APIs, and to some extent, microservices. Microservices are autonomous and loosely coupled, but they sometimes share some functionality. With OpenAPI, you can define reusable components in one place and reuse object definitions across your APIs. The approach works similarly to the libraries of functions and object definitions provided in SDKs and languages. A typical example would be a single definition of a 'User' that holds information about an employee’s access privileges, location, or organizational role. All microservices would refer to that single source of truth for the User, reducing complexity when employees’ roles change.

4. Automate processes

Today, you can find many tools, including free open source tools, that allow you to automate much of the API and microservices design process. Automating design processes not only saves you time but also improves the accuracy and quality of your API and microservices designs, so it's worth finding the right tools for your organization and project. Stoplight's open source tools are free, and can scale for enterprise use and facilitate collaboration more effectively as your needs grow.

Next steps with microservices

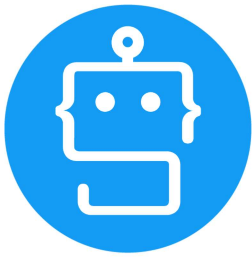
You have a lot to consider if you're thinking about building an application with microservices. And the best practices we've highlighted are only the beginning.

Here are some additional microservices best practices you might tackle as a next step:

- Maintaining a central repository for your APIs and microservices.
- Practicing continuous integration and delivery (CI/CD).
- Monitoring and visualizing your architecture.
- Implementing automated service discovery.

Ready to get started with microservices? Then sign up for a free account to see how Stoplight can help you follow best practices and build consistent, high-quality APIs and microservices.
Develop microservices up to 10x faster.

By clicking "Accept All Cookies", you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.



Stoplight

Privacy Preference Center

When you visit any website, it may store or retrieve information on your browser, mostly in the form of cookies. This information might be about you, your preferences or your device and is mostly used to make the site work as you expect it to. The information does not usually directly identify you, but it can give you a more personalized web experience. Because we respect your right to privacy, you can choose not to allow some types of cookies. Click on the different category headings to find out more and change our default settings. However, blocking some types of cookies may impact your experience of the site and the services we are able to offer.

Manage Consent Preferences

Strictly Necessary Cookies

These cookies are necessary for the website to function and cannot be switched off in our systems. They are usually only set in response to actions made by you which amount to a request for services, such as setting your privacy preferences, logging in or filling in forms. You can set your browser to block or alert you about these cookies, but some parts of the site will not then work. These cookies do not store any personally identifiable information.

Performance Cookies

These cookies allow us to count visits and traffic sources so we can measure and improve the performance of our site. They help us to know which pages are the most and least popular and see how visitors move around the site. All information these cookies collect is aggregated and therefore anonymous. If you do not allow these cookies we will not know when you have visited our site, and will not be able to monitor its performance.

Functional Cookies

These cookies enable the website to provide enhanced functionality and personalisation. They may be set by us or by third party providers whose services we have added to our pages. If you do not allow these cookies then some or all of these services may not function properly.

Targeting Cookies

These cookies may be set through our site by our advertising partners. They may be used by those companies to build a profile of your interests and show you relevant adverts on other sites. They do not store directly personal information, but are based on uniquely identifying your browser and internet device. If you do not allow these cookies, you will experience less targeted advertising.

Cookie List

Your Privacy [`dialog closed`]