raygun.com

What are microservices? The pros, cons, and how they work

Microservices are a popular software design architecture that breaks apart monolithic systems. Applications are built as collections of loosely coupled services. Each microservice is responsible for a single feature. They interact with each other through communication protocols such as HTTP and TCP.

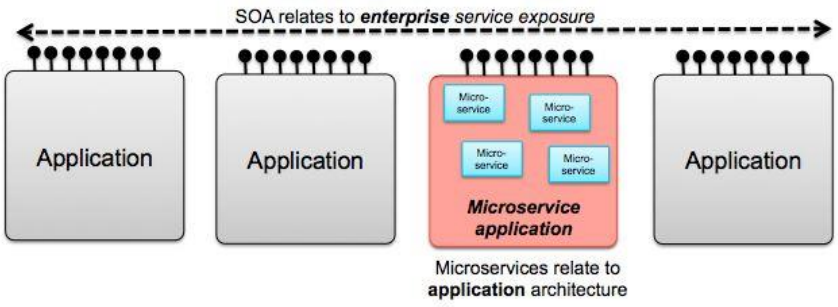
Microservices follow the principles of service-oriented architecture (SOA) design. Although SOA has no official standards, principles defined in Thomas Erl's book entitled "SOA: Principles of Service Design" are often used as rules of thumb.

They are as follows:

1. Standardized service contract (services follow a standardized description)
2. Loose coupling (minimal dependencies)
3. Service abstraction (services hide their internal logic)
4. Service reusability (service structure is planned according to the DRY principle)
5. Service autonomy (services internally control their own logic)
6. Service statelessness (services don't persist state from former requests)
7. Service discoverability (services come with discoverable metadata and/or a service registry)
8. Service composability (services can be used together)

Although SOA and microservices follow similar principles, their relationship is often debated. Some developers emphasize their similarity and consider microservices as a subtype of SOA. Others rather stress their differences and claim that each solves a different set of problems.

According to the latter view, SOA has an enterprise scope while microservices have an application scope. Within the enterprise scope, apps communicate with each other.



You can read more about the SOA vs microservices debate in this article by IBM DeveloperWorks. The most interesting question, however, is how microservices compare to monolithic applications.

Pros of microservices

Microservices have become hugely popular in recent years. Mainly, because they come with a couple of benefits that are super useful in the era of containerization and cloud computing. You can develop

and deploy each microservice on a different platform, using different programming languages and developer tools. Microservices use APIs and communication protocols to interact with each other, but they don't rely on each other otherwise.

The biggest pro of microservices architecture is that teams can develop, maintain, and deploy each microservice independently. This kind of single-responsibility leads to other benefits as well. Applications composed of microservices scale better, as you can scale them separately, whenever it's necessary. Microservices also reduce the time to market and speed up your CI/CD pipeline. This means more agility, too. Besides, isolated services have a better failure tolerance. It's easier to maintain and debug a lightweight microservice than a complex application, after all.

Cons of microservices As microservices heavily rely on messaging, they can face certain problems. Communication can be hard without using automation and advanced methodologies such as Agile. You need to introduce DevOps tools such as CI/CD servers, configuration management platforms, and APM tools to manage the network. This is great for companies who already use these methods. However, the adoption of these extra requirements can be a challenge for smaller companies.

Having to maintain a network lead to other kinds of issues, too. What we gain on the simplicity of single-responsibility microservices, lose on the complexity of the network. Or, at least a part of it. For instance, while independent microservices have better fault tolerance than monolithic applications, the network has worse.

Communication between microservices can mean poorer performance, as sending messages back and forth comes with a certain overhead. And, while teams can choose which programming language and platform they want to use, they also need to collaborate much better. After all, they need to manage the whole lifecycle of the microservice, from start to end.

Pros	Cons
Greater agility	Needs more collaboration (each team has to cover the whole microservice lifecycle)
Faster time to market	Harder to test and monitor because of the complexity of the architecture
Better scalability	Poorer performance, as microservices need to communicate (network latency, message processing, etc.)
Faster development cycles (easier deployment and debugging)	Harder to maintain the network (has less fault tolerance, needs more load balancing, etc.)
Easier to create a CI/CD pipeline for single-responsibility services	Doesn't work without the proper corporate culture (DevOps culture, automation practices, etc.)
Isolated services have better fault tolerance	Security issues (harder to maintain transaction safety, distributed communication goes wrong more likely, etc.)
Platform- and language agnostic services	

Pros	Cons
Cloud-readiness	

Examples of Microservices

Microservices in Java

Java is one of the best languages to develop microservices. There are a couple of microservice frameworks for the Java platform you can use, such as:

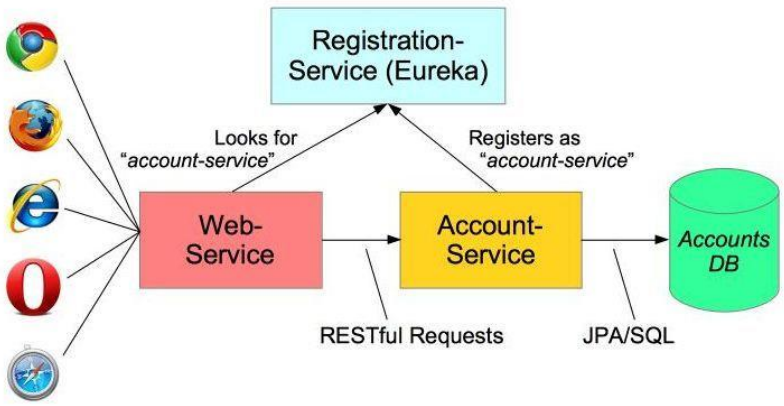
- Dropwizard
- JHipster
- Spark framework
- Spring framework
- Swagger
- Play framework
- Vert.x

Using Spring Boot is the most popular way to build microservices in Java. Spring Boot is a utility built on top of the Spring platform. It makes it possible to set up stand-alone Spring apps with minimal configuration. It can save a lot of time by automatically configuring Spring and third-party libraries.

For instance, here's a very simple microservices example by Paul Chapman, published in detail in the official Spring blog. It uses Spring, Spring Boot, and Spring Cloud to build the application. I won't include code examples here, as you can find them in the original article. We'll only briefly take a look at the application structure. The steps are as follows:

1. Create the service registration (so that microservices can find each other), using the Eureka registration server (incorporated in Spring Cloud)
2. Create an account management microservice called "Account Service" with Spring Boot
3. Create a web service to access the microservice, using Spring's RestTemplate class

Here's an illustration of the app's structure (also from the Spring blog):



Of course, this is a very simple example. In real-life apps, the web service makes requests to more than one microservices.

Spring Boot is a great tool to build microservices in Java, however, you can use other frameworks as well. Here are some other great articles (with example code), too:

- Currency conversion microservice with Spring Boot, by "Spring Boot Tutorial"
- Reactive microservices with Vert.x, by Clement Escoffier from RedHat
- TaskList microservice with Dropwizard, by Bartosz Jedrzejewski
- Blog microservice with JHipster, by Matt Raible

Microservices in Docker

Containerization is one of the biggest trends in the dev world right now. Docker, being the most popular containerization platform, is an excellent tool to build microservices. You have different options to structure microservices in Docker.

You can deploy each microservice in its own Docker container,read more on how Docker works You can also break down a microservice into various processes and run each in a separate container.

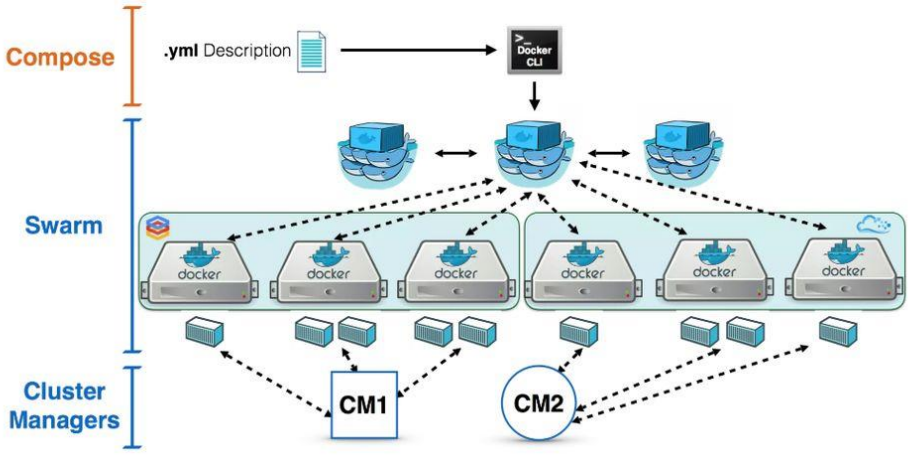
You can also use Docker Compose to run multi-container applications. It can save you a lot of time, as you don't have to create and start each container separately.

With Docker Compose, you can configure your app's microservices using a YAML file. If you are interested here's a useful article by Linode on how to deploy microservices with Docker and Docker Compose.

If you have a large-scale application with several containers you can also make use of a container orchestration platform. The two most popular tools are Docker Swarm and Kubernetes. Both allow you to deploy containers to a cluster of computers instead of just one machine.

Docker Swarm is embedded in the Docker Engine; it's Docker's native orchestration tool. Kubernetes was created by Google and, it's the most popular orchestration platform at the moment. While Swarm fits well into the Docker ecosystem and it's easy to set up, Kubernetes is more customizable and has higher fault tolerance.

Below, you can see an illustration from the Docker blog about how to use Docker Swarm and Compose together to manage container clusters:



You won't need a container orchestration tool in the case of a smaller app. However, you might want to automate container management when you deal with several microservices.

Microservices in the Cloud Microservices are frequently run as cloud applications, as they are lightweight and easy to scale and deploy. Popular cloud platforms come with several microservice-friendly features, such as:

- On-demand resources
- Pay as you go pricing
- Infrastructure as code
- Continuous Deployment and Delivery
- Managed services (e.g. dealing with scaling, software configuration and optimization, automatic software updates, etc.)
- Large choice of programming languages, operating system, database technologies
- Built-in tools such as Docker and Kubernetes

Microservices in the cloud are usually deployed in containers, as that's how you can make the most out of the infrastructure. Besides, containers are isolated, run anywhere, and create a predictable environment. However, it's also possible to deploy microservices in the cloud without using containers. Although the latter solution is less common, sometimes it's the better choice.

Conclusion

Microservices are the most suitable for large-scale applications. Smaller apps are usually better off with a monolithic code base, though.

While it's easier to develop and maintain independent microservices, network management requires additional efforts. Container platforms, DevOps practices, and cloud computing can help a lot in adopting the microservices architecture.

Source Raygun.com