Why Microservices?

by Shamik Mitra MVB · Aug. 26, 16 · Integration Zone

Visually compose APIs with easy-to-use tooling. Learn how IBM API Connect provides near-universal access to data and services both on-premises and in the cloud, brought to you in partnership with IBM.

Companies like Netflix, Amazon, and others have adopted the concept of microservices in their products. Microservices are one of the hottest topics in the software industry, and many organizations want to adopt them. Especially helpful is the fact that DevOps can play very well with microservices.

But what is a microservice? Why should an organization adopt them?

To understand them, let's first take a look at monolithic software.

In monolithic software, we mainly use a three-tier architecture:

- Presentation layer
- Business layer
- Data access layer

Say a traditional web application client (a browser) posts a request. The business tier executes the business logic, the database collects/stores application specific persistence data, and the UI shows the data to the user.

However, there are several problems with this type of system. All code (**presentation, business layer, and data access layer**) is maintained within the same code base.

Although logically we divide the services like JMS Service and Data-Access Service, they are on the same code base and deployed as a single unit.

Even though you created a multi-module project, one module is dependent on another and, moreover, the module needs dependent modules in its class path. Although you use a distributed environment, it runs under single process context

So, in a single process, different services are communicating with each other. To achieve this, all artifacts and their required libraries (jars) are required in each application container.

Say a JMS service want to use the data access layer. The JMS container needs the data access layer jars and the jars upon which the data access layer is dependent (second level dependencies).

In this concept, there are lots of **pain points**, and the architecture is very rigid in nature.

Here are some of the problems you face with a monolith.

Problem 1

As there is one codebase, it grows gradually. Every programmer, whether it's a UI Developer or a business layer developer, commits in same code base, which becomes very inefficient to manage. Suppose one developer only works in the JMS module, but he has to pull the whole codebase to his local and configure the whole module in order to run it on a local server. Why? He should only concentrate on the JMS module, but the current scenario doesn't allow for that.

Problem 2

As there is one code base and modules are dependent on each other, minimal change in one module needs to generate all artifacts and needs to deploy in each server pool in a distributed environment.

Suppose in a multi-module project that the JMS module and business module are dependent on the data access module. A simple change in the data access module means we need to re-package the JMS module and business module and deploy them in their server pool.

module and deploy them in their server pool.

Problem 3

As monolithic software uses a three-tier architecture, three cross-functional teams are involved in developing a feature. Even though a three-tier architecture allows for separation of responsibility, in the long-run, the boundaries are crossed and the layers lose their fluidity and become rigid.

Suppose an inventory management feature has been developed. The UI, business layer, and data access layer have their own jobs. But everyone wants to take control of the main business part so that when defects come up, they can solve them and are not dependent on another layer's developer. Due to this competition, those boundaries end up being crossed, which results in inefficient architecture.

Problem 4

In many projects, I have seen that there is a developer team and another support team. The developer team only develops the project, and after it's released, they hand it over to the support team. I personally don't support this culture. Although some knowledge transfer happens during the handover, it doesn't solve the problem. For critical incidents, the support team has to get help from the developer team, which hurts their credibility.

Problem 5

As our system is monolithic, so is our team management. Often, we create teams base on the tier — UI developers, backend developers, database programmers, etc. They are experts in their domains, but they have little knowledge about other layers. So when there's a critical problem, it encompasses each layer, and the blame game starts. Not only that, but it takes additional time to decide which layer's problem it is and who needs to solve the issue

Netflix and Amazon address these problems with a solution called microservices.

Microservice architecture tells us to break a product or project into independent services so that it can be deployed and managed solely at that level and doesn't depend on other services.

Subscribe

After seeing this definition, an obvious question comes to

After seeing this definition, an obvious question comes to mind. On what basis do I break down my project into independent services?

Many people have the wrong idea about microservices. Microservices aren't telling you to break your project down based on the tier, such as JMS, UI, logging, etc.

No this is absolutely not. We need to break it down by function. A complete function and its functionality may consist of UI, business, logging, JMS, data access, JNDI lookup service, etc.

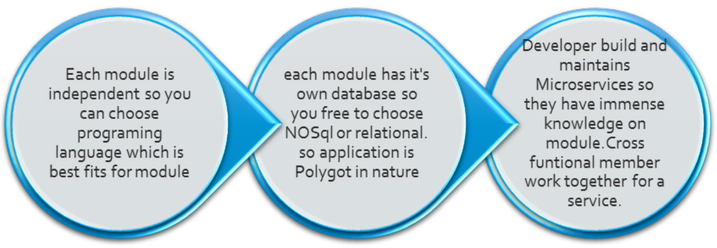
The function should not be divisible and not dependent on other functions.

So If the project has Inventory, Order, Billing, Shipping, and UI shopping cart modules, we can break each service down as an independently deployable module. Each has its own maintenance, monitoring, application servers, and database. So with microservices, there is no centralized database — each module has its own database.

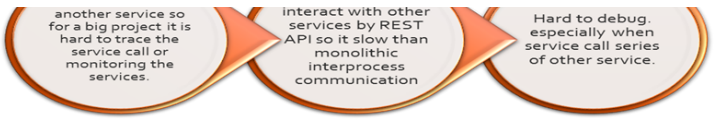
And it could be a relational or a NoSQL database. The choice is yours based on the module. It creates a ***polyglot persistence.***

The most important aspect of microservice culture is that whoever develops the service, it is that team's responsibility to manage it. This avoids the handover concept and the problems associated with it.

Microservice Benefits and Shortcomings


Each module is independent so you can choose programing language which is best fits for module
each module has it's own database so you free to choose NOSql or relational. so application is Polygot in nature
Developer build and maintains Microservices so they have immense knowledge on module.Cross funtional member work together for a service.

Subscribe


another service so for a big project it is hard to trace the service call or monitoring the services.
interact with other services by REST API so it slow than monolithic interprocess communication
Hard to debug. especially when service call series of other service.

Benefit 1

As in monolithic software, you only develop in one language, say Java, as the code base. But with microservices, as each service is independent and each service is a new project, each service can be developed in any language that is best fits for the requirement.

Benefit 2

The developer is only concentrated on a particular service, so the code base will be very small, and the developer will know the code very well.

Benefit 3

When one service needs to talk with another service, they can talk via API, specifically by a REST service. A REST service is the medium to communicate through, so there is little transformation. Unlike SOA, a microservice message bus is much thinner than an ESB, which does lots of transformation, categorization, and routing.

Benefit 4

There is no centralized database. Each module has its own, so there's data decentralization. You can use NoSQL or a relational database depending on the module, which introduces that polyglot persistence I mentioned before.

A lot of people think SOA and microservices are the same thing. By definition, they look the same, but SOA is used for communicating different systems over an ESB, where the ESB takes a lot of responsibility to manage data, do categorization, etc.

But microservices use a dumb message bus which just transfers the input from one service to another, but its endpoint is smart enough to do the aforementioned tasks**.** It has a dumb message bus, but smart endpoints.

As microservices communicate through REST, the transformation scope is very small — only one service is dependent on another service via API call.

Subscribe

But Microservices Have Shortcomings, Too

As every functional aspect is an individual service, so in a big project, there are many services. Monitoring these services adds to the overhead.

Not only that, but when there's a service failure, tracking it down can be a painstaking job.

Service calls to one another, so tracing the path and debugging can be difficult, too.

Each service generates a log, so there is no central log monitoring. That's painful stuff, and we need a very good log management system for it.

With microservices, each service communicares through API/remote calls, which have more overhead than with monolothic software's interprocess communication calls.

But in spite of all of those detriments, microservices do real separation of responsibilities.

Topics: MICROSERVICE ARCHITECTURE, SEPARATION OF CONCERNS, SOA