BLUEMETAL
An Insight company

What Is a Microservice?

A Digital Transformation From Monolithic to Microservice

Contents

Insight

"Architecture is a result of a process of asking questions and testing them and re-interrogating and changing in a repetitive way." — Thom Mayne

Software as a Service (SaaS) has emerged as a model for modern software products that provides customers a great experience and the business a dynamic platform for frequent releases to support campaigns, marketing communication, and the ability to roll out new features to keep customers engaged.

In order to achieve this level of high-quality software development, a combination of lean engineering and DevOps-driven continuous delivery provide a methodology and process framework for high-velocity, high-quality software product development.

Cloud platforms such as Microsoft® Azure® provide an excellent foundation for SaaS solutions with their on-demand infrastructure and application services. In order to optimize the software development process and take advantage of the advanced Platform as a Service (PaaS) capabilities, microservices have emerged as the de facto application architecture.

BLUEMETAL
An Insight company

Microservices are …

The term "microservice" can be a bit misleading. The prefix "micro" implies that microservices are small. The "micro" in microservices is actually in reference to the scope of functionality that the service provides. A microservice provides a business or platform capability through a well-defined API, data contract and configuration. It provides this function and only this function. It does one thing, and it does it well. This simple concept provides the foundation for a framework that will guide the design, development and deployment of your microservices.

Within the context of doing one thing and doing it well, microservices also exhibit a number of other properties and behaviors; it is these elements that differentiate microservices from previous incarnations of service-oriented approaches. These elements affect every aspect of how we develop software today, from team structure, source code organization and control to continuous integration, packaging and deployment.

We will examine these properties and behaviors; we will also look at both the benefits and the challenges of this microservice approach.

Along with this examination, we will learn how to identify microservices and how to determine where the seams and boundaries are within the domains in which you are working.

Autonomous and Isolated

Autonomous: *Existing or capable of existing independently; responding, reacting or developing independently of the whole*

Isolated: *Separate from others, happening in different places and at different times*

Microservices are self-contained units of functionality with loosely coupled dependencies on other services and are designed, developed, tested and released independently.

Implications

For the past several years, we have been developing standards and practices for team development of large, complex systems using a layered, monolithic architecture. This is reflected in how we organize into teams, structure our solutions and source code control systems, and package and release our software.

Monolithic solutions are built, tested and deployed as one large body of code, typically across a set of server or Virtual Machine (VM) instances, in order to provide scale and performance. If a bug is fixed, a feature added or content updated, the entire solution is built, tested and deployed across the server farm as one large entity. The process of building, deploying and regression-testing the monolith is costly and time-consuming. Over time, these monoliths turn into large, complex, tightly coupled systems that are nearly impossible to maintain and evolve in new directions.

Insight

If you want to adopt a microservices architecture, your standards and practices will need to adapt to this new pattern. Teams will need to be organized in such a way as to support the development of microservices as distinct, independent products. The development, test and production environments will need to be organized to support these teams, developing and deploying their microservice products separate from one another.

When changes are made, only the microservice affected needs to go through the deployment pipeline, thus simplifying the process of updating the system and delivering new features and functions. As a result, the speed of development will increase and the cost of making changes will go down.

Elastic, resilient and responsive

Elastic: *Capable of returning to its original length, shape, etc., after being stretched, deformed, compressed or expanded*

Resilient: *Able to become strong, healthy or successful again after something bad happens*

Responsive: *Quick to respond or react*

Microservices are reused across many different solutions and therefore must be able to scale appropriately depending on the usage scenario. They must be fault-tolerant and provide a reasonable time frame for recovery if something goes awry. Finally, they need to be responsive, providing reasonable performance given the execution scenario.

Implications

The environment in which you deploy your microservices must provide dynamic scale and high-availability configurations for both stateful and stateless services. This is achieved by leveraging a modern cloud platform such as Microsoft Azure.

Cloud platforms provide all of the necessary capabilities to support elastic scale, fault tolerance and high availability, as well as configuration options that allow you the right size for performance.

Message-oriented and programmable

Message-oriented: *Software that connects separate systems in a network by carrying and distributing messages between them*

Programmable: *A plan of tasks that are done in order to achieve a specific result*

Microservices rely on APIs and data contracts to define how interaction with the service is achieved. The API defines a set of network-visible endpoints, and the data contract defines the structure of the message that is either sent or returned.

Insight

Implications

Defining service endpoints and data contracts is not new. Microservice architecture builds on the evolution of industry standards to define the interaction semantics. If these standards evolve or new ones are introduced, a microservice architecture will evolve to adopt these new standards.

While there are other options for defining microservice contracts, developers have, for the most part, settled on Representational State Transfer (REST) over HTTP for defining API endpoints and JavaScript® Object Notation (JSON) for the definition of message contracts. In addition, capabilities such as store and forward messaging and pub-sub broadcast patterns are used to provide loose coupling between components along with an asynchronous programming model.

APIs and data contracts are the outermost edge of a microservice and define how a client of the service can invoke a function. Behind this API may be a very sophisticated set of software components, storage media and multiple VM instances that provide the run-time function. To consumers of the service, this is all a black box, meaning they know the published inputs and outputs but nothing else about the inner workings. What is expected is that a message is constructed, the API is invoked and a response is returned. The interaction between the consumer and the service is finite and distinct.

Configurable

Configurable: *To design or adapt to form a specific configuration or for some specific purpose*

Microservices must provide more than just an API and a data contract. Each microservice will have different levels of configuration, and the act of configuring may take different forms. The key point, in order to be reusable and to address the needs of each system that chooses to employ its capabilities, is a microservice must provide a means by which it can be appropriately molded to the usage scenario.

Implications

As you begin the design process for a microservice, you will soon discover that multiple APIs will emerge. Along with the public-facing API you want to expose to the world, other endpoints will surface that are more of an administrative function and will be used to define how to bootstrap, monitor, manage, scale, configure and perform other perfunctory operations on the service.

Like any good software product, a microservice should provide an easy-to-use interface or console for administrative functions to configure and manage running instances. Behind the console is, of course, a set of private to semi-private APIs that provides access to the underlying data and configuration settings driving the service.

A microservice, then, is more than just its public-facing REST API and consists of multiple APIs with varying levels of access, supporting administrative consoles and a run-time infrastructure to maintain all of the above. It is a software product with all the trimmings.

Insight

Automated

Automated: *Having controls that allow something to work or happen without being directly controlled by a person*

The lifecycle of a microservice should be fully automated, from design through deployment.

Implications

As you ponder this new world of software product development made up of microservices, it may occur to you that this whole effort could be quite complex, with a proliferation of independent microservice products. You would not be wrong.

This approach should not be undertaken without first having complete automated control over the software development lifecycle. This is about having the right set of tools to support a fully automated development pipeline, but more importantly, it is about evolving to a DevOps culture.

A DevOps culture is one that promotes collaboration and integration of the development and operations teams. When you form a team that is responsible for the design, implementation and deployment of a microservice, that team should be cross functional, consisting of all the skills necessary to carry the process from design through deployment. This means that traditional development teams consisting of architects, developers and testers should be expanded to include operations.

While developers are traditionally responsible for the automation from build through test, called continuous integration, the operations group is traditionally responsible for deployment across test, staging and production. Combining these teams offers the opportunity to make automation of the entire product development pipeline — as well as the monitoring, diagnostics and recovery operations — a first-class activity of the product team.

The benefits of microservices

Now that you have a working definition of microservices, let us examine the benefits of this approach to distributed computing.

Evolutionary

The days of big bang software product development are over. It is no longer possible to go on a multimonth or multiyear development cycle before releasing a product because, by the time you release, the window of opportunity has passed and your competition has already been there, done that.

You may find yourself responsible for the ongoing maintenance and development of an existing, complex, monolithic system made up of millions of lines of code. There will be no stomach for the complete re-implementation of such a system; the business could not sustain it.

Insight

One of the benefits of microservice architecture is that you can evolve toward this approach one service at a time, identifying a business capability, implementing it as a microservice, and integrating using a loose coupling pattern, with the existing monolith providing a bridge to the new architecture

Over time, more and more capabilities can be migrated, shrinking the scope of the monolith until it is just a husk of its original form. The move to microservices will open the door to new user experiences and new business opportunities.

Open

Microservices are designed to expose their functionality through industry standards for network-addressable APIs and data contracts, and are hosted on highly scalable, elastic, resilient cloud platforms.

This allows a microservice team to choose the programming language, operating system and data store that best fits the needs of their service and their skill set without worrying about interoperability

It is possible to have some microservice teams developing in Node.JS, others in Java, and still others in C#. All those microservices can be used together in one solution because the composition is happening at the REST API level.

This can be a tremendous improvement for teams that are distributed around the world and currently trying to work together on a monolithic solution. By creating cross-functional teams in each geography and giving them complete responsibility over specific microservices, the need to coordinate around the clock goes away and is replaced with coordination at the API layer, which is not time-bound.

High-velocity

Individual microservices have a small surface area of functionality. With one team responsible for the development lifecycle and all the various components, technology and automation that make up its implementation, the velocity at which a microservice can be designed, developed, deployed and updated is magnitudes faster than trying to perform the same operations across a monolithic solution.

Reusable and composable

Microservices by their very nature are reusable. They are independent entities providing a business or platform capability and exposing that functionality through open internet standards.

In order to create a useful solution for an end user, multiple microservices can be composed together. These user experiences can be implemented as web and mobile applications or targeting new devices such as wearables that may become popular in the future.

Insight

Flexible

By defining deployment and scale scenarios through automation tools, the microservice team can exert a great deal of control. There is a tremendous amount of flexibility in how the service moves through development, test, staging, and production and, when in production, how it can be modified to fit different usage scenarios through configuration.

Automation of deployment and scale configurations will provide the necessary control of defining the run-time environment from the container in which the service runs, the instances onto which those containers are deployed, the geographical regions into which those instances are instantiated, and the elastic configurations that define scale.

Versionable and replaceable

Since there is complete control over the deployment scenarios for a microservice, it becomes possible to have multiple versions of a service running side by side, providing backward compatibility and easy migration.

Versioning is typically handled at the API level where version numbers are integrated into the URL. A new version of the microservice API can be released without impacting clients that are using previous versions of the API. It is also possible to provide ongoing updates and enhancements to existing services while in production.

Using this approach, services can be fully replaced while maintaining the current API or new implementations can be released under a new version.

Owned by one team

As mentioned, a microservice architecture approach requires organizing cross-functional teams for the purpose of owning the microservice product lifecycle from design through deployment. If you are down the path of adopting agile principles and the Scrum process, you are well-suited to adopt this architecture pattern.

The challenges of microservices

A move to microservices will not be without its challenges. Go in with both your eyes and mind wide open.

[Re]organization

Organizing to support a microservice architecture approach is one of the most difficult challenges you will have. If you are part of a command and control organization using a waterfall software project management approach, you will struggle because you are not oriented to high-velocity product development.

Insight

If you lack a DevOps culture and there is no collaboration between development and operations to automate the deployment pipeline, you will struggle.

If you are looking for an opportunity to adopt this approach, it is recommended you not try to make large, sweeping changes to your organization. Instead, look for an opportunity within the context of a business initiative to test out this new formula and then follow these steps:

- Form a small cross-functional team.
- Provide training and guidance on adopting agile, Scrum, Azure and microservice architecture.
- Provide a separate physical location for this team to work so they are not adversely effected by internal politics and old habits.
- Take a minimal-viable-product approach and begin to deliver small incremental releases of one microservice all the way through the lifecycle.
- Integrate this service with the existing systems using a loosely coupled approach.
- Go through the lifecycle on this microservice several times until you feel comfortable with the process.
- Put the core team into leadership positions as you form new cross-functional teams to disseminate the knowledge.

Platform

Creating the run-time environment for microservices requires a significant investment in dynamic infrastructure across regionally dispersed data centers. If your current on-premise application platform does not support automation, dynamic infrastructure, elastic scale and high availability, then it makes sense to consider a cloud platform.

Identification

Domain-driven design was introduced by Eric Evans in his book by the same title. Evans outlined an approach to describing domain models made up of entities; their attributes, roles and relationships; and the bounded contexts, the areas of business capability where these models are applied.

From a well-articulated, domain-driven design, you can formulate a layered architecture that will provide the framework for a monolithic solution (see Figure 1).

Insight

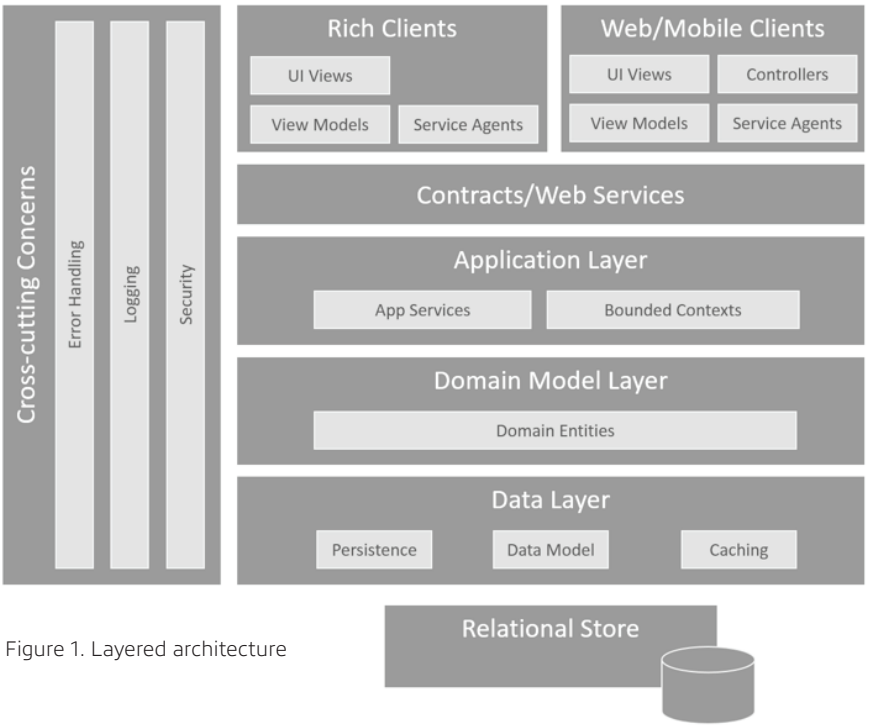

Figure 1. Layered architecture

Domain-driven design has served us well and survived the test of time. Domain modeling is still a relevant technique we can use in the age of microservices. Instead of mapping our models and bounded contexts to a layered architecture, we can instead find the seams and separate each bounded context along with its model and use that as the starting point for a microservice architecture.

If you are currently working with a complex layered architecture and have a reasonable domain model defined, the domain model will provide a road map to an evolutionary approach to migrating to a microservice architecture.

If a domain model does not exist, you can apply domain-driven design in reverse to identify the bounded contexts, the capabilities within the system. Look for areas of the system where the language changes; this is a design seam. These seams define the boundaries of possible candidates for microservices.

You can also look for areas of the system that are changing rapidly, have a lot of refactoring going on, or where the rate of change is very slow and system components are solid. These areas may make good candidates for microservices.

Finally, areas of the system that are causing the most pain may be good candidates as well as you cleave the affected appendages and replace them with bright, shiny, new microservices.

Insight

Testing

Microservices do not alter much about the way we write and test code. Test-driven development, mocking and unit, functional and regression testing are all in play. We are doing object-oriented development of service-oriented components, and all of the best practices, techniques and tools we used in the past still apply. In addition to these traditional testing mechanisms, we need to test the microservice as it moves through the deployment pipeline.

- **Internals:** Test the internal functions of the service, including use of data access, caching and other cross-cutting concerns.
- **Service:** Test the service implementation of the API. This is a private internal implementation of the API and its associated models.
- **Protocol:** Test the service at the protocol level, calling the API over the specified wire protocol, usually HTTP.
- **Composition:** Test the service in collaboration with other services within the context of a solution.
- **Scalability/throughput:** Test the scalability and elasticity of the deployed microservice.
- **Failover/fault tolerance:** Test the ability of the microservice to recover after a failure.
- **Penetration:** Work with a third-party software security firm to perform penetration testing. Note: This requires cooperation with Microsoft if you are pen-testing microservices deployed to Azure.

Discoverability

Locating services in a distributed environment can be handled in three ways:

- **Hard-code the locations of the microservices** and deal with the issues that arise when services move or fail with or without notice. This is akin to hard-coding a database connection string or user ID and password. It is not a good idea.
- **Leverage file-based or run-time environment-based configuration mechanisms** to store and retrieve the microservice locations. This is a good choice if combined with an automated process to update when locations change.
- **Provide a dynamic location microservice for microservices** so that applications and services can look up the current location at run time. This lookup service may also provide health checks and notices if services are failing or performing poorly.

In order to provide discoverability as a service, it may require either acquiring a third-party product, integrating an open-source solution or building it yourself.

Insight

BLUEMETAL
An Insight company

Summary

Microservices do one thing, and they do it well. They represent business capabilities defined using domain-driven design, implemented using object-oriented best practices, tested at each step in the deployment pipeline and deployed through automation as autonomous, isolated, highly scalable, resilient services in a distributed cloud infrastructure. They are owned by a single team that approaches the development of a microservice as a product, delivering high-quality software in an iterative, high-velocity process with customer involvement and satisfaction as the key metrics of success.

Microservice architecture — along with a lean engineering methodology, DevOps drive, continuous delivery and an advanced cloud platform that provides both Infrastructure as a Service (IaaS) and PaaS — supply the process and tools necessary to achieve successful SaaS development. The only additional need is a team that understands the process and how to use the tools to be successful.

About the author

BlueMetal, an Insight company, is an interactive design and technology architecture consulting firm that solves the most challenging business and technical problems facing our clients. We leverage lean engineering methodology, DevOps process, microservice architecture and cloud platforms to deliver mission-critical and business-transformative SaaS solutions for our clients.

As the national practice director at BlueMetal, Bob Familiar leads a team of seasoned principal architects who are responsible for technical pre-sales, industry research and development, rapid prototyping and outreach to the technology community. He holds a patent in object-relational architecture and technology and is author of the book “Microservices, IoT and Azure.”

Insight