Kartik Padave

70362019039

A022

**Practical 7**

**Aim:** To do a Case study on Microservices and implementation of microservices in application

**Microservices**



Microservices architecture is a cloud native architectural approach in which a single application is composed of many loosely coupled and independently deployable smaller components, or services.

Microservices are not necessarily exclusively relevant to cloud computing but there are a few important reasons why they so frequently go together—reasons that go beyond microservices being a popular architectural style for new applications and the cloud being a popular hosting destination for new applications.

Among the primary benefits of microservices architecture are the utilization and cost benefits associated with deploying and scaling components individually. While these benefits would still be present to some extent with on-premises infrastructure, the combination of small, independently scalable components coupled with on-demand, pay-per-use infrastructure is where real cost optimizations can be found.

Secondly, and perhaps more importantly, another primary benefit of microservices is that each individual component can adopt the stack best suited to its specific job. Stack proliferation can lead to serious complexity and overhead when you manage it yourself but consuming the supporting stack as cloud services can dramatically minimize management challenges. Put another way, while it’s not impossible to roll your own microservices infrastructure, it’s not advisable, especially when just starting out.

The microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.

Enterprise Applications are often built in three main parts: a client-side user interface (consisting of HTML pages and JavaScript running in a browser on the user's machine) a database (consisting of many tables inserted into a common, and usually relational, database management system), and a server-side application. The server-side application will handle HTTP requests, execute domain logic, retrieve and update data from the database, and select and populate HTML views to be sent to the browser. This server-side application is a monolith - a single logical executable. Any changes to the system involve building and deploying a new version of the server-side application.

Such a monolithic server is a natural way to approach building such a system. All your logic for handling a request runs in a single process, allowing you to use the basic features of your language to divide up the application into classes, functions, and namespaces. With some care, you can run and test the application on a developer's laptop, and use a deployment pipeline to ensure that changes are properly tested and deployed into production. You can horizontally scale the monolith by running many instances behind a load-balancer.

These frustrations of monolithic applications have led to the microservice architectural style: building applications as suites of services. As well as the fact that services are independently deployable and scalable, each service also provides a firm module boundary, even allowing for different services to be written in different programming languages. They can also be managed by different teams .

Today microservices can be used with any language or tool. However, there are a few tools which have become pretty common in use of microservices both at a personal and at an enterprise level:

* **Containers, Docker, and Kubernetes**

One of the key elements of a microservice is that it’s generally pretty small. (There is no arbitrary amount of code that determines whether something is or isn’t a microservice, but “micro” is right there in the name.) When Docker ushered in the modern container era in 2013, it also introduced the compute model that would become most closely associated with microservices. Because individual containers don’t have the overhead of their own operating system, they are smaller and lighter weight than traditional virtual machines and can spin up and down more quickly, making them a perfect match for the smaller and lighter weight services found within microservices architectures. With the proliferation of services and containers, orchestrating and managing large groups of containers quickly became one of the critical challenges. Kubernetes, an open-source container orchestration platform, has emerged as one of the most popular orchestration solutions because it does that job so well.

* **API gateways**

Microservices often communicate via API, especially when first establishing state. While it’s true that clients and services can communicate with one another directly, API gateways are often a useful intermediary layer, especially as the number of services in an application grows over time. An API gateway acts as a reverse proxy for clients by routing requests, fanning out requests across multiple services, and providing additional security and authentication. There are multiple technologies that can be used to implement API gateways, including API management platforms, but if the microservices architecture is being implemented using containers and Kubernetes, the gateway is typically implemented using Ingress or, more recently, Istio.

* **Messaging and event streaming**

While best practice might be to design stateless services, state nonetheless exists and services need to be aware of it. And while an API call is often an effective way of initially establishing state for a given service, it’s not a particularly effective way of staying up to date. A constant polling, “are we there yet?” approach to keeping services current simply isn’t practical. Instead, it is necessary to couple state-establishing API calls with messaging or event streaming so that services can broadcast changes in state and other interested parties can listen for those changes and adjust accordingly. This job is likely best suited to a general-purpose message broker, but there are cases where an event streaming platform, such as Apache Kafka, might be a good fit. And by combining microservices with event driven architecture developers can build distributed, highly scalable, fault tolerant and extensible systems that can consume and process very large amounts of events or information in real-time.

* **Serverless**

Serverless architectures take some of the core cloud and microservices patterns to their logical conclusion. In the case of serverless, the unit of execution is not just a small service, but a function, which can often be just a few lines of code. The line separating a serverless function from a microservice is a blurry one, but functions are commonly understood to be even smaller than a microservice. Where serverless architectures and Functions-as-a-Service (FaaS) platforms share affinity with microservices is that they are both interested in creating smaller units of deployment and scaling precisely with demand.

Some microservices characteristics are:

1. **Componentization via Services**

For as long as we've been involved in the software industry, there's been a desire to build systems by plugging together components, much in the way we see things are made in the physical world. During the last couple of decades, we've seen considerable progress with large compendiums of common libraries that are part of most language platforms. When talking about components we run into the difficult definition of what makes a component. Our definition is that a component is a unit of software that is independently replaceable and upgradeable. Microservice architectures will use libraries, but their primary way of componentizing their own software is by breaking down into services. We define libraries as components that are linked into a program and called using in-memory function calls, while services are out-of-process components who communicate with a mechanism such as a web service request, or remote procedure call. (This is a different concept to that of a service object in many OO programs). One main reason for using services as components (rather than libraries) is that services are independently deployable. If you have an application that consists of multiple libraries in a single process, a change to any single component results in having to redeploy the entire application. But if that application is decomposed into multiple services, you can expect many single service changes to only require that service to be redeployed. That's not an absolute, some changes will change service interfaces resulting in some coordination, but the aim of a good microservice architecture is to minimize these through cohesive service boundaries and evolution mechanisms in the service contracts. Another consequence of using services as components is a more explicit component interface. Most languages do not have a good mechanism for defining an explicit Published Interface. Often it's only documentation and discipline that prevents clients breaking a component's encapsulation, leading to overly-tight coupling between components. Services make it easier to avoid this by using explicit remote call mechanisms. Using services like this does have downsides. Remote calls are more expensive than in-process calls, and thus remote APIs need to be coarser-grained, which is often more awkward to use. If you need to change the allocation of responsibilities between components, such movements of behaviour are harder to do when you're crossing process boundaries. At a first approximation, we can observe that services map to runtime processes, but that is only a first approximation. A service may consist of multiple processes that will always be developed and deployed together, such as an application process and a database that's only used by that service.

1. **Organized around Business Capabilities**

When looking to split a large application into parts, often management focuses on the technology layer, leading to UI teams, server-side logic teams, and database teams. When teams are separated along these lines, even simple changes can lead to a cross-team project taking time and budgetary approval. A smart team will optimise around this and plump for the lesser of two evils - just force the logic into whichever application they have access to. Logic everywhere in other words. This is an example of Conway's Law in action.

Enterprise benefits of microservices are:

* **Independently deployable**

Perhaps the single most important characteristic of microservices is that because the services are smaller and independently deployable, it no longer requires an act of Congress in order to change a line of code or add a new feature in application. Microservices promise organizations an antidote to the visceral frustrations associated with small changes taking huge amounts of time. It doesn’t require a Ph.D. in computer science to see or understand the value of an approach that better facilitates speed and agility. But speed isn’t the only value of designing services this way. A common emerging organizational model is to bring together cross-functional teams around a business problem, service, or product. The microservices model fits neatly with this trend because it enables an organization to create small, cross-functional teams around one service or a collection of services and have them operate in an agile fashion. Microservices' loose coupling also builds a degree of fault isolation and better resilience into applications. And the small size of the services, combined with their clear boundaries and communication patterns, makes it easier for new team members to understand the code base and contribute to it quickly—a clear benefit in terms of both speed and employee morale.

* **Right tool for the job**

In traditional n-tier architecture patterns, an application typically shares a common stack, with a large, relational database supporting the entire application. This approach has several obvious drawbacks—the most significant of which is that every component of an application must share a common stack, data model and database even if there is a clear, better tool for the job for certain elements. It makes for bad architecture, and it’s frustrating for developers who are constantly aware that a better, more efficient way to build these components is available. By contrast, in a microservices model, components are deployed independently and communicate over some combination of REST, event streaming and message brokers—so it’s possible for the stack of every individual service to be optimized for that service. Technology changes all the time, and an application composed of multiple, smaller services is much easier and less expensive to evolve with more desirable technology as it becomes available.

* **Precise scaling**

With microservices, individual services can be individually deployed—but they can be individually scaled, as well. The resulting benefit is obvious: Done correctly, microservices require less infrastructure than monolithic applications because they enable precise scaling of only the components that require it, instead of the entire application in the case of monolithic applications.

The enterprises also face a number of challenges while adopting and using microservices:

* **Complexity**

A microservices application has more moving parts than the equivalent monolithic application. Each service is simpler, but the entire system as a whole is more complex.

* **Development and testing**

Writing a small service that relies on other dependent services requires a different approach than a writing a traditional monolithic or layered application. Existing tools are not always designed to work with service dependencies. Refactoring across service boundaries can be difficult. It is also challenging to test service dependencies, especially when the application is evolving quickly.

* **Lack of governance**

The decentralized approach to building microservices has advantages, but it can also lead to problems. You may end up with so many different languages and frameworks that the application becomes hard to maintain. It may be useful to put some project-wide standards in place, without overly restricting teams' flexibility. This especially applies to cross-cutting functionality such as logging.

* **Network congestion and latency**

The use of many small, granular services can result in more interservice communication. Also, if the chain of service dependencies gets too long (service A calls B, which calls C...), the additional latency can become a problem. You will need to design APIs carefully. Avoid overly chatty APIs, think about serialization formats, and look for places to use asynchronous communication patterns like queue-based load levelling.

* **Data integrity**

With each microservice responsible for its own data persistence. As a result, data consistency can be a challenge. Embrace eventual consistency where possible.

* **Management**

To be successful with microservices requires a mature DevOps culture. Correlated logging across services can be challenging. Typically, logging must correlate multiple service calls for a single user operation.

* **Versioning**

Updates to a service must not break services that depend on it. Multiple services could be updated at any given time, so without careful design, you might have problems with backward or forward compatibility.

* **Skill set**

Microservices are highly distributed systems. Carefully evaluate whether the team has the skills and experience to be successful.