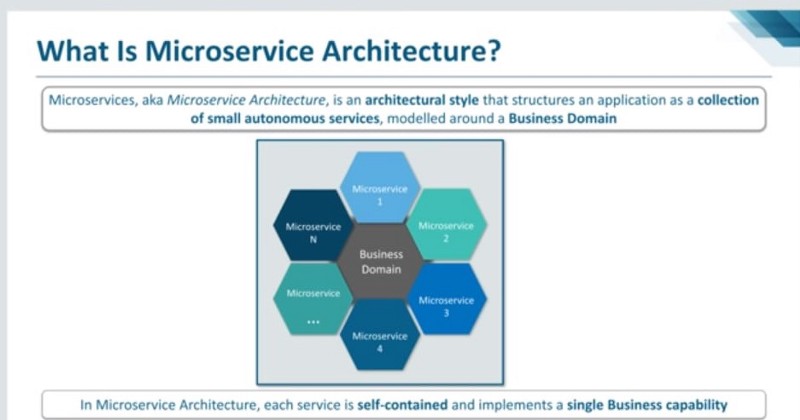
# What is Microservices ?

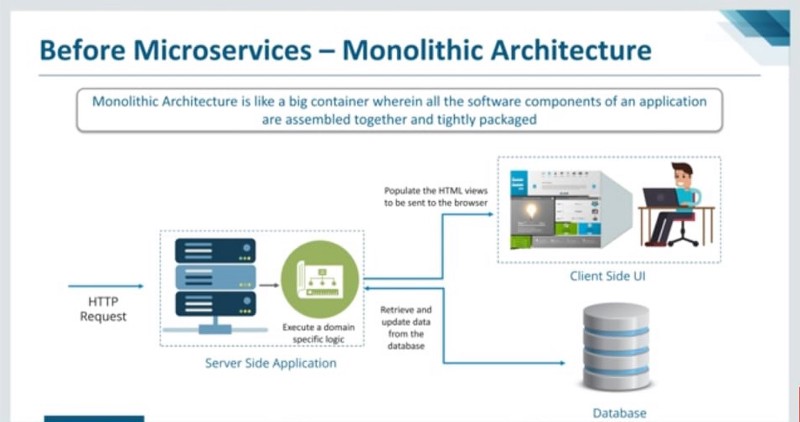


Microservices is a hot topic in software development circles these days. And for some very good reasons.

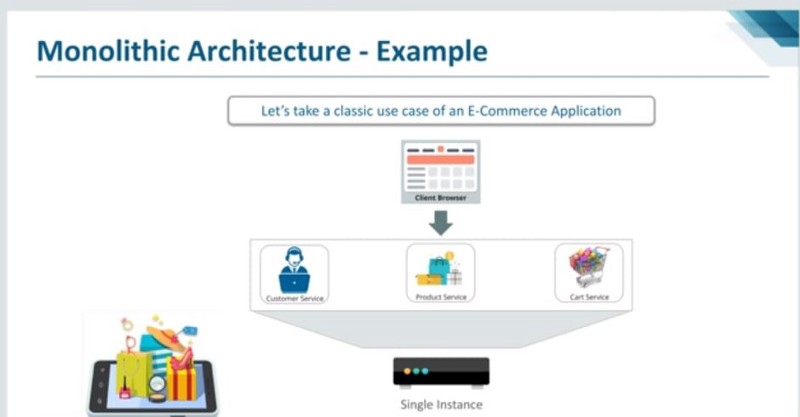
Put simply, the traditional way of building enterprise applications — using a [monolithic approach](https://en.wikipedia.org/wiki/Monolithic_application) — has become problematic as applications get larger and more complex. So developers are turning to a microservices software development architecture, in which applications are structured as collections of loosely coupled services. This makes them easier to build, and — more importantly — much easier to expand and scale.

Let’s take a closer look at how a microservices approach differs from a monolithic one, and examine their relative strengths and weaknesses. Before moving further first we understand the Monolithic architecture in detail, in order to understand the microservices better and later we will also differentiate among them so that you are about to be a pro in the topic.

# What is Monolithic Architecture??



A monolithic architecture is the traditional unified model for the design of a software program. Monolithic, in this context, means composed all in one piece. Monolithic software is designed to be self-contained; components of the program are interconnected and interdependent rather than loosely coupled as is the case with modular software programs. In a tightly-coupled architecture, each component and its associated components must be present in order for code to be executed or compiled.

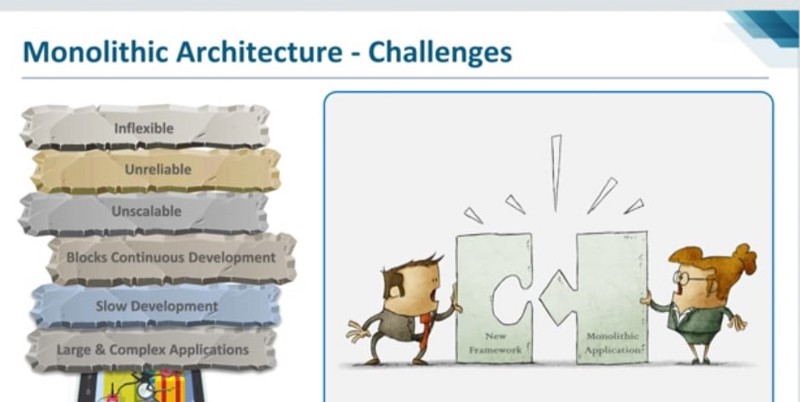


In our example of E-commerce application, let’s figure out the Monolithic Architecture is , so as u see that in basic e-commerce application, we have common options of costumer service,product service and cart service which a costumer can access through their browser, and when you launch the application it is deployed as single monolithic application. There is only one single instance, so we have costumer service,product service and cart service and when you deploy all these services, it will be basically a single monolithic application.Now what you can do is, in order to scale it you can run multiple instances of this application behind a load balancer.

# Advantages of Monolithic Architecture:

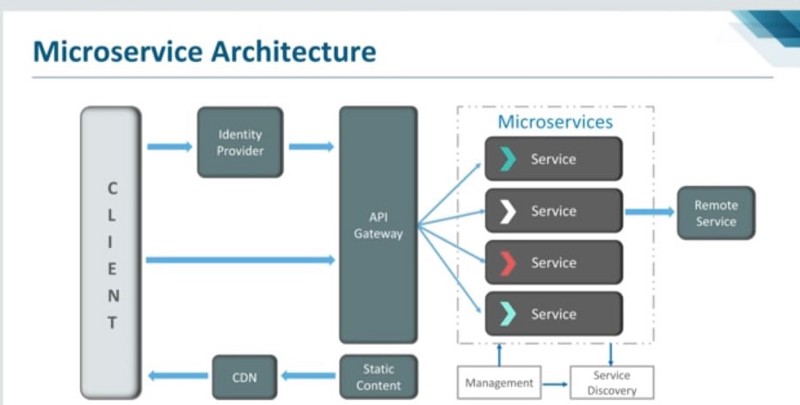
* Simple to develop.
* Simple to test. For example you can implement end-to-end testing by simply launching the application and testing the UI with Selenium.
* Simple to deploy. You just have to copy the packaged application to a server.
* Simple to scale horizontally by running multiple copies behind a load balancer.

# Challenges of Monolithic Architecture:



* This simple approach has a limitation in size and complexity.
* Application is too large and complex to fully understand and made changes fast and correctly.
* The size of the application can slow down the start-up time.
* You must redeploy the entire application on each update.
* Impact of a change is usually not very well understood which leads to do extensive manual testing.
* Continuous deployment is difficult.
* Monolithic applications can also be difficult to scale when different modules have conflicting resource requirements.
* Another problem with monolithic applications is reliability. Bug in any module (e.g. memory leak) can potentially bring down the entire process. Moreover, since all instances of the application are identical, that bug will impact the availability of the entire application.
* Monolithic applications has a barrier to adopting new technologies. Since changes in frameworks or languages will affect an entire application it is extremely expensive in both time and cost.

# What is Microservice Architecture ?



The idea is to split your application into a set of smaller, interconnected services instead of building a single monolithic application. Each microservice is a small application that has its own hexagonal architecture consisting of business logic along with various adapters. Some microservices would expose a REST, RPC or message-based API and most services consume APIs provided by other services. Other microservices might implement a web UI.

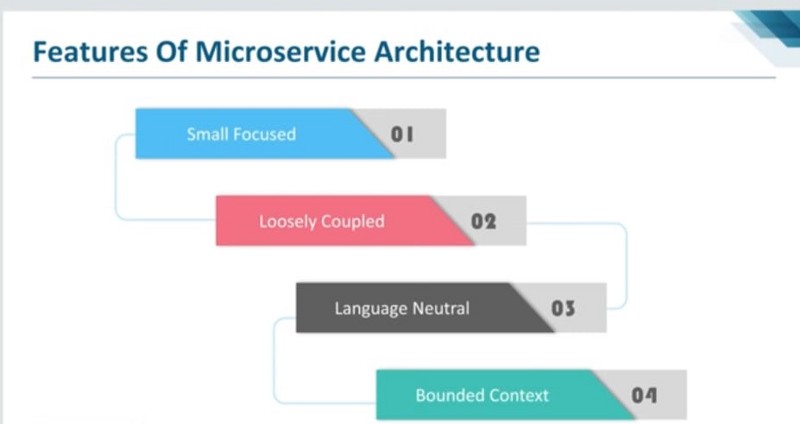
The Microservice architecture pattern significantly impacts the relationship between the application and the database. Instead of sharing a single database schema with other services, each service has its own database schema. On the one hand, this approach is at odds with the idea of an enterprise-wide data model. Also, it often results in duplication of some data. However, having a database schema per service is essential if you want to benefit from microservices, because it ensures loose coupling. Each of the services has its own database. Moreover, a service can use a type of database that is best suited to its needs, the so-called [polyglot persistence](http://www.infoq.com/presentations/The-Evolving-Panorama-of-Data) architecture.

Some APIs are also exposed to the mobile, desktop, web apps. The apps don’t, however, have direct access to the back-end services. Instead, communication is mediated by an intermediary known as an [API Gateway](https://www.linkedin.com/pulse/api-gateway-pattern-ronen-hamias). The API Gateway is responsible for tasks such as load balancing, caching, access control, API metering, and monitoring.

# Features of Microservices

There are several main characteristics/features of Microservices:

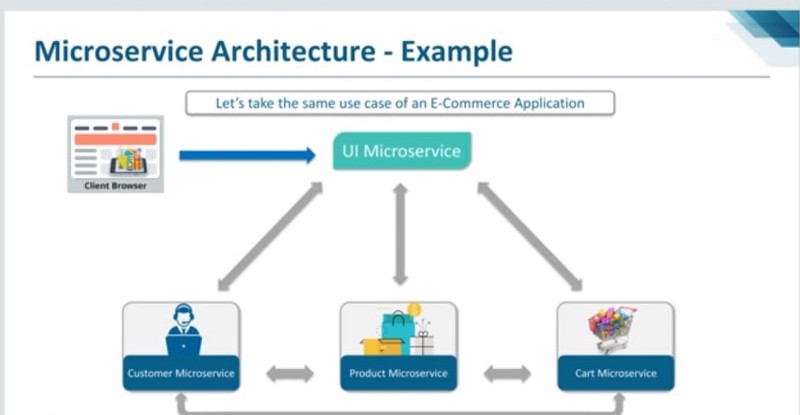
* Small Focused
* Loosely coupled
* Language Flexible
* Bounded Context



Used effectively, microservice architectures allow you to scale your application as the number of developers working on your application increases. The key is to build applications without creating a complex, unwieldy beast at the macro level. That means keep tracking each time a new service is added to your system or a new connection between microservices is made.

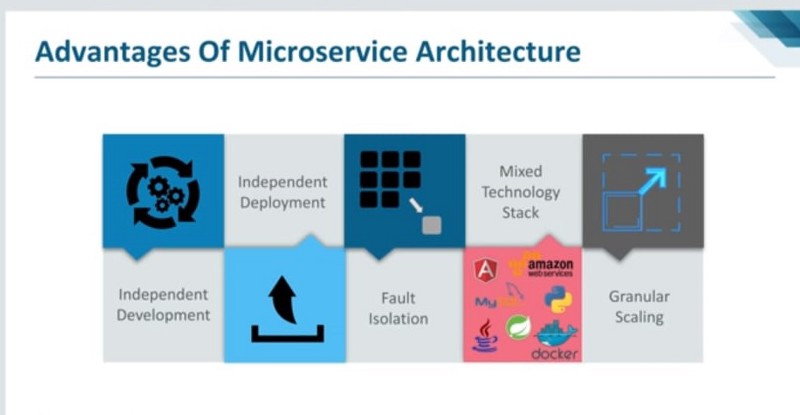
It also means examining the complexity increase and making sure it is warranted and well understood. Regularly examining the entire application system is critical to keep an interconnected set of microservices working effectively and reliably.

lets again take the same E-commerce website example to understand microservice architecture.



In our application the Client side is not at all disturbed by any other feature addition and updates. All the three services are containerized in separate microservices.

# Advantages of Microservices Architecture



* Microservice architecture gives developers the freedom to independently develop and deploy services
* A microservice can be developed by a fairly small team
* Code for different services can be written in different languages (though many practitioners discourage it)
* Easy integration and automatic deployment (using open-source continuous integration tools such as Jenkins, Hudson, etc.)
* Easy to understand and modify for developers, thus can help a new team member become productive quickly
* The developers can make use of the latest technologies
* The code is organized around business capabilities
* Starts the web container more quickly, so the deployment is also faster
* When change is required in a certain part of the application, only the related service can be modified and redeployed — no need to modify and redeploy the entire application
* Better fault isolation: if one microservice fails, the other will continue to work (although one problematic area of a monolith application can jeopardize the entire system)
* Easy to scale and integrate with third-party services
* No long-term commitment to technology stack

# Drawbacks of Microservices Architecture:

* Due to distributed deployment, testing can become complicated and tedious
* Increasing number of services can result in information barriers
* The architecture brings additional complexity as the developers have to mitigate fault tolerance, network latency, and deal with a variety of message formats as well as load balancing
* Being a distributed system, it can result in duplication of effort
* When number of services increases, integration and managing whole products can become complicated
* In addition to several complexities of monolithic architecture, the developers have to deal with the additional complexity of a distributed system
* Developers have to put additional effort into implementing the mechanism of communication between the services
* Handling use cases that span more than one service without using distributed transactions is not only tough but also requires communication and cooperation between different teams