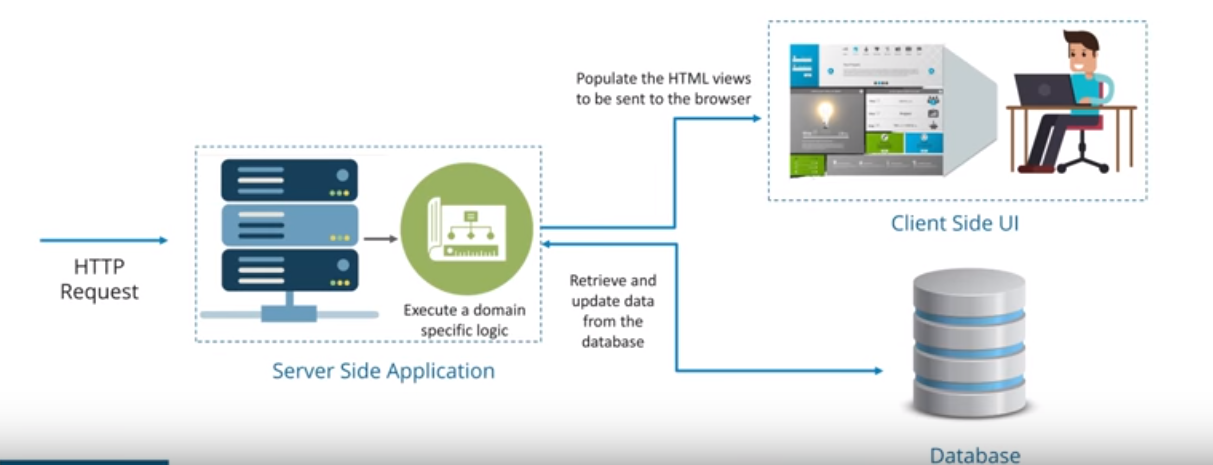
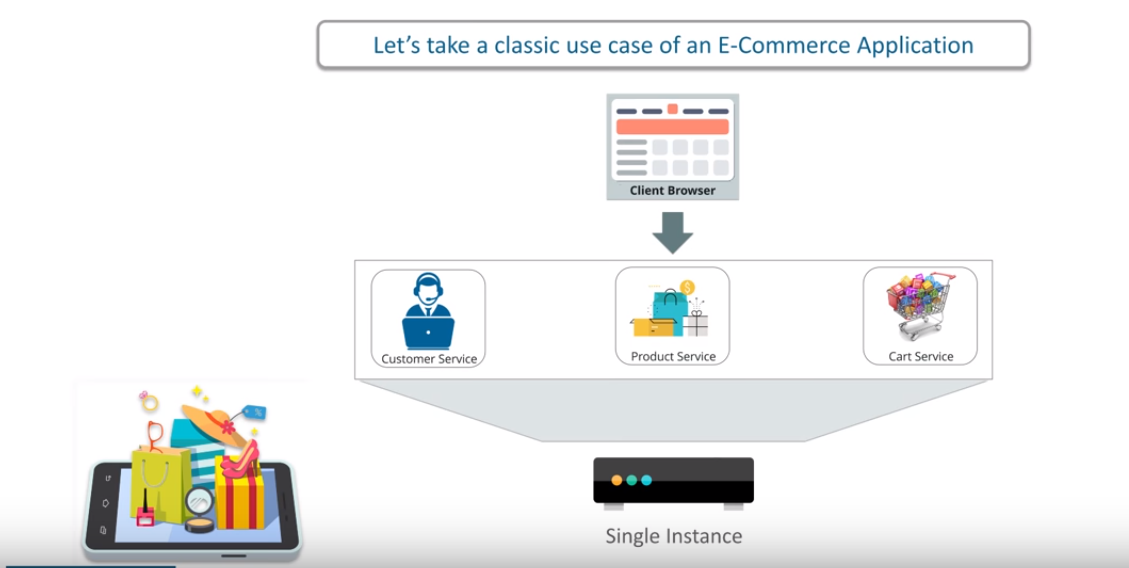
* **Why Microservices?**
* **What is Microservices?**
* **Features of Microservices Architecture**
* **Adv. Of Microservices**

**Why Microservices ?**

* Monolithic Architecture is like big container wherein all software components of an application are assembled together and tightly packaged.



**Monolithic Archicture - Example**



**Advantages of Monolithic Architecture.**

* Easy to Develop
* Can have a large number of cross cutting concerns such as logging, rate limiting, security features such as audit trails
* Share memory Access
* Simplified Repository Administration
* Easy branching and tagging

**Demerits of Monolithic Architecture.**

* Large and complex application
* Overloaded IDE
* Overloaded Web container
* Continuous deployment is difficult
* Scaling is difficult
* Long term commitment to Tech Stack

<http://microservices.io/patterns/monolithic.html>

The large monolithic code base intimidates developers, especially ones who are new to the team. The application can be difficult to understand and modify. As a result, development typically slows down. Also, because there are not hard module boundaries, modularity breaks down over time. Moreover, because it can be difficult to understand how to correctly implement a change the quality of the code declines over time. It’s a downwards spiral.

Overloaded IDE - the larger the code base the slower the IDE and the less productive developers are.

Overloaded web container - the larger the application the longer it takes to start up. This had have a huge impact on developer productivity because of time wasted waiting for the container to start. It also impacts deployment too.

Continuous deployment is difficult - a large monolithic application is also an obstacle to frequent deployments. In order to update one component you have to redeploy the entire application. This will interrupt background tasks (e.g. Quartz jobs in a Java application), regardless of whether they are impacted by the change, and possibly cause problems. There is also the chance that components that haven’t been updated will fail to start correctly. As a result, the risk associated with redeployment increases, which discourages frequent updates. This is especially a problem for user interface developers, since they usually need to iterative rapidly and redeploy frequently.

Scaling the application can be difficult - a monolithic architecture is that it can only scale in one dimension. On the one hand, it can scale with an increasing transaction volume by running more copies of the application. Some clouds can even adjust the number of instances dynamically based on load. But on the other hand, this architecture can’t scale with an increasing data volume. Each copy of application instance will access all of the data, which makes caching less effective and increases memory consumption and I/O traffic. Also, different application components have different resource requirements - one might be CPU intensive while another might memory intensive. With a monolithic architecture we cannot scale each component independently

Obstacle to scaling development - A monolithic application is also an obstacle to scaling development. Once the application gets to a certain size its useful to divide up the engineering organization into teams that focus on specific functional areas. For example, we might want to have the UI team, accounting team, inventory team, etc. The trouble with a monolithic application is that it prevents the teams from working independently. The teams must coordinate their development efforts and redeployments. It is much more difficult for a team to make a change and update production.

Requires a long-term commitment to a technology stack - a monolithic architecture forces you to be married to the technology stack (and in some cases, to a particular version of that technology) you chose at the start of development. With a monolithic application, can be difficult to incrementally adopt a newer technology. For example, let’s imagine that you chose the JVM. You have some language choices since as well as Java you can use other JVM languages that inter-operate nicely with Java such as Groovy and Scala. But components written in non-JVM languages do not have a place within your monolithic architecture. Also, if your application uses a platform framework that subsequently becomes obsolete then it can be challenging to incrementally migrate the application to a newer and better framework. It’s possible that in order to adopt a newer platform framework you have to rewrite the entire application, which is a risky undertaking.

**What is Microservices?**

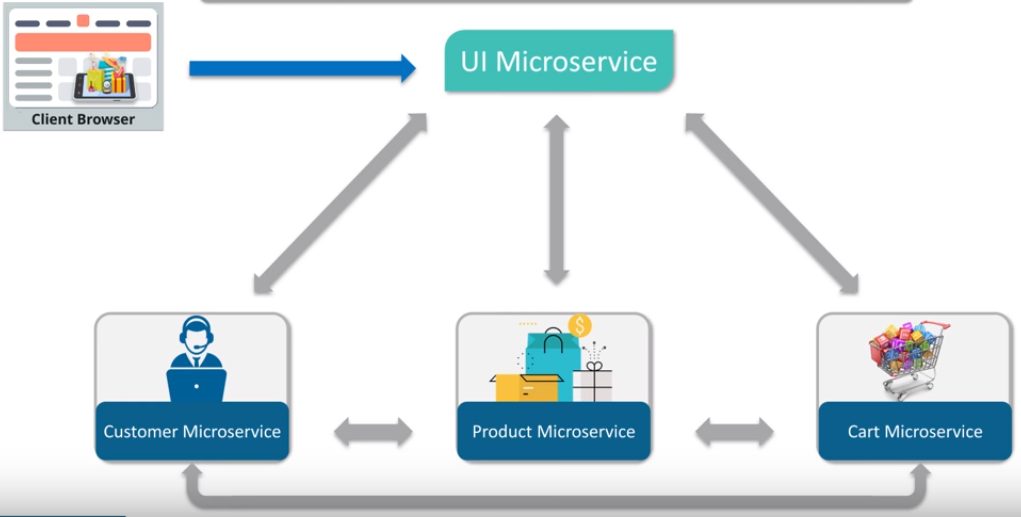
**What is Microservice Architecture?**

* Microservice, Microservice Architecture, is an **Architectural style** that structure an application as a collection of small autonomous services modelled around a **Business Domain.**

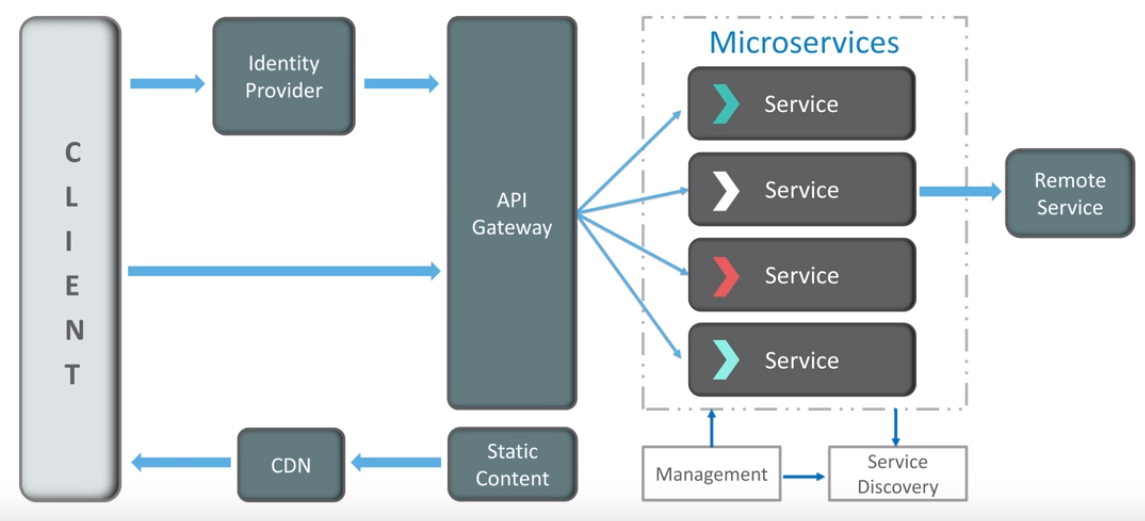


**In Microservice architecture, each service is self-contained and implement a simple Business capability.**

**Microservice Architecture**



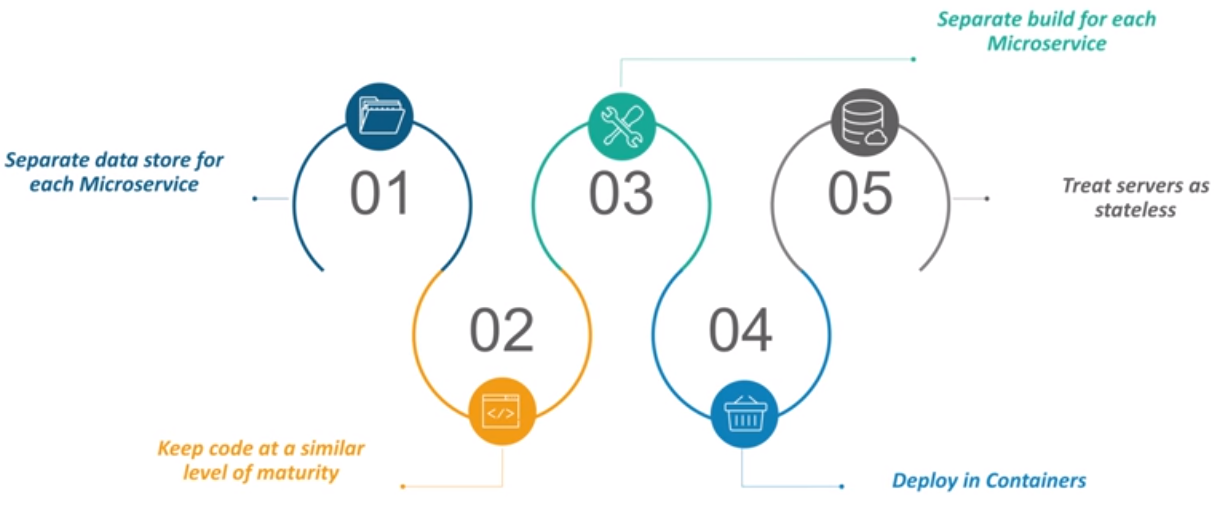
**Microservice Architecture – Deep Dive**

---

**Features of Microservices**

* Small Focused
* Loosely Coupled
* Language Neutral
* Bounded Context

**Best Practices**



**Advantages of Microservices**

* Independent development
* Development by a small team
* Services can be written using different languages
* Fault isolation
* Mixed technology stack
* Granular Scaling
* 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 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

**Challenges with Microservices**

* Due to distributed deployment, testing can become complicated and tedious
* 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
* 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
* Increased memory consumption