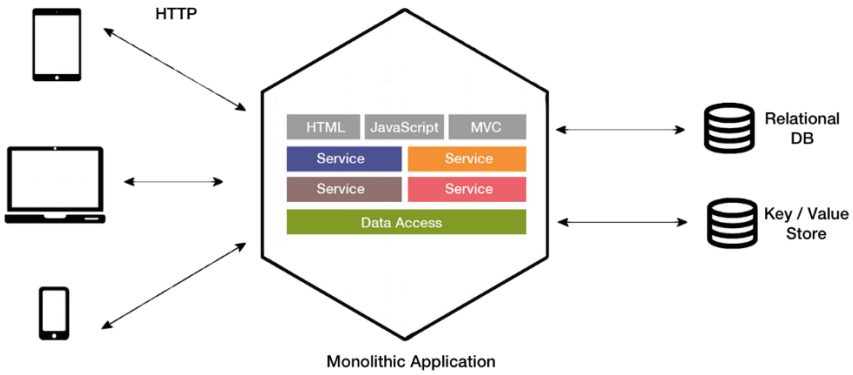
Agenda: Microservices and Kubernetes

- What is Monolithic Application
- What are Microservices
- Kubernetes Architecture
- Azure Kubernetes Service (AKS)
- Example

What are Monolithic Applications?

*In **software engineering**, a monolithic application describes a single-tiered **software application** in which the **user interface** and data access code are combined into a single program from a single **platform**. A monolithic application is self-contained, and independent from other computing applications.*



Benefits of Monolithic Architecture

- Simple to develop.
- Simple to test. 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.

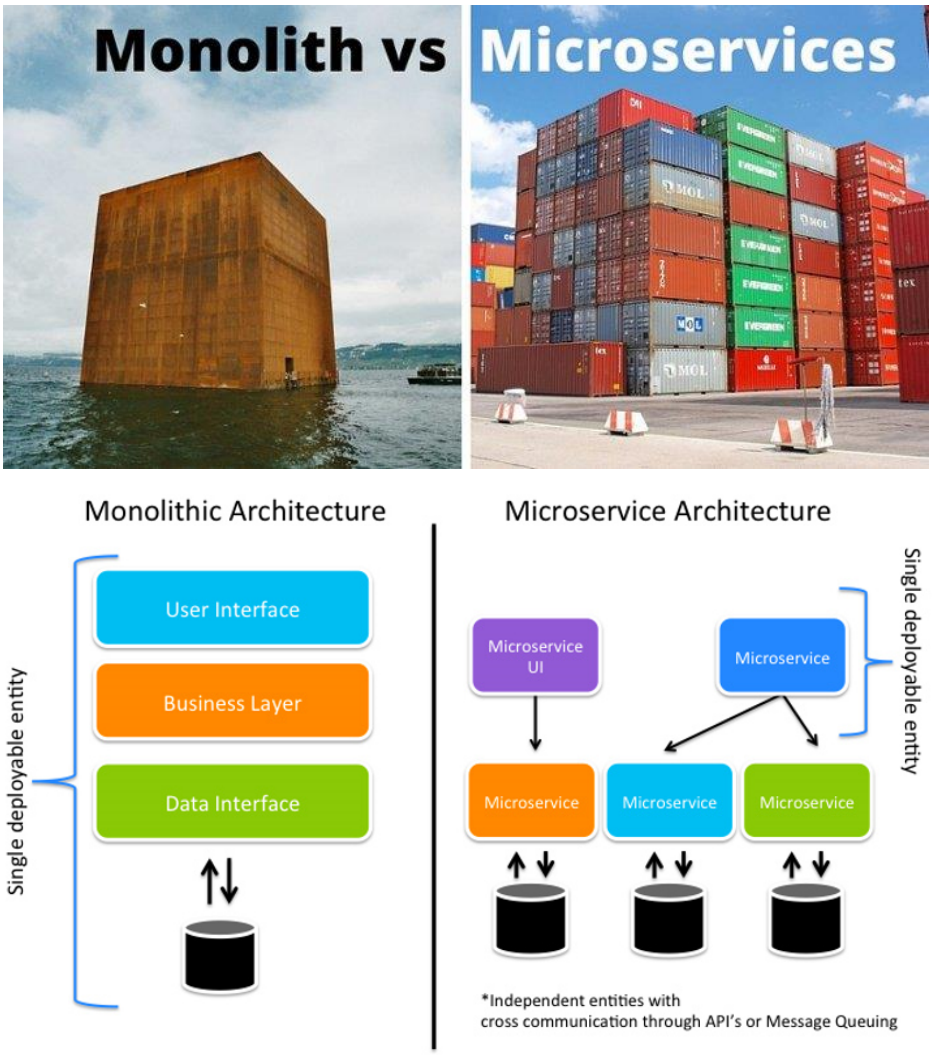
Drawbacks of Monolithic Architecture

- 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**.
- **Difficult to scale** when different modules have conflicting resource requirements.

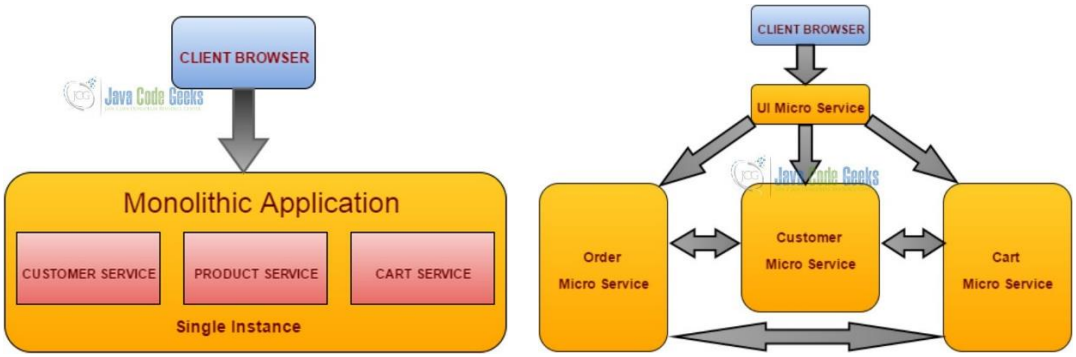
- **Bug** in any module (e.g. memory leak) can potentially bring **down** the **entire process**. Moreover, since all instances of the application are identical, that bug will **impact** the **availability** of the entire application.
- Monolithic applications have a **barrier to adopting new technologies**. Since changes in frameworks or languages will affect an entire application it is extremely expensive in both time and cost.

What are Microservices

- ***Microservice applications are composed of small, independently versioned, and scalable customer-focused services that communicate with each other over standard protocols with well-defined interfaces.***
- The idea is to split your application into a set of smaller, interconnected services.
- 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.



- Also, monolithic pattern conflicts with the container principle "***a container does one thing, and does it in one process***".
- The microservices approach allows agile changes and rapid iteration of each microservice, because you can change specific, small areas of complex, large, and scalable applications.



Drawback of Microservices

- ✓ Deployment is complex.
- ✓ Debugging and Testing is difficult. For a similar test for a service you would need to launch that service and any services that it depends upon it.
- ✓ Monitoring/Logging is difficult
- ✓ New service versions must support old/new API contracts. In a Microservice architecture you need to carefully plan and coordinate the rollout of changes to each of the services.
- ✓ Distributed databases make transactions hard.
- ✓ Cluster and orchestration tools overhead.
- ✓ Distributed services adds more network communication
    - ✓ Increased network hops.
    - ✓ Requires failure/recovery code
    - ✓ Need service discovery solution
- ✓ Advanced DevOps capability will be required.

What is Kubernetes

Containerization has brought a lot of flexibility for developers in terms of managing the deployment of the applications. However, the more granular the application is, the more components it consists of and hence requires some sort of management for those.

One still needs to take care of scheduling the deployment of a certain number of containers to a specific node, managing networking between the containers, following the resource allocation, moving them around as they grow and much more.

Nearly all applications nowadays need to have answers for things like

- Replication of components
- Auto-scaling
- Load balancing
- Rolling updates
- Logging across components
- Monitoring and health checking
- Service discovery
- Authentication

The process of organizing multiple **containers and managing them as needed** is known as **container orchestration**.

Kubernetes, a container orchestration technology used to orchestrate the deployment and management of hundreds and thousands of containers in clustered environment.

- Kubernetes is an open source project was started by Google in the year 2014. **Redhat** is the second major contributor along with Microsoft, HP, VMWare etc…
- The name **Kubernetes** originates from Greek, meaning *helmsman (who steers ship or boat)* or *pilot*, and is the root of *governor* and cybernetic (theory or study of communication and control). **K8s** is an abbreviation derived by replacing the 8 letters "ubernete" with "8".
- It is a **platform** designed to completely manage the life cycle of containerized applications and services using methods that provide **predictability, scalability, and high availability**.
- It orchestrates **computing**, **networking**, and **storage** infrastructure on behalf of user workloads.
- It is easier to **deploy**, **scale**, and **manage** applications.
- Kubernetes aims to support an extremely diverse variety of workloads, including **stateless**, **stateful**, and **data-processing** workloads. If an application can **run in a container**, it should run great on Kubernetes.

Kubernetes is:

- a container platform.
- a microservices platform
- a portable cloud platform.

Following are some of the important features of Kubernetes.

- Simplifies Application Deployment
- Continues development, integration and deployment
- Application-centric management and better hardware utilization.
- Modern tooling, have CLI and management REST API support.

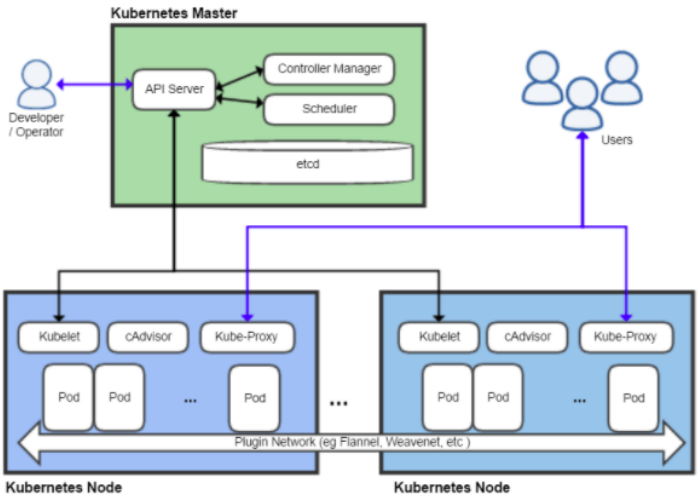
- Highly Scalable infrastructure with Autoscaling support
- Kubernetes Monitors Health of each node and does self-healing of docker containers.
- Rolling updates

Kubernetes Architecture

A cluster is a group of nodes. They can be physical servers or virtual machines that have the Kubernetes platform installed.

Components in Kubernetes are divided into

1. Master Components
2. Worker / Node Components



Kubernetes Master:

A single master host will manage the cluster and run several core Kubernetes services.

They are responsible for providing global decisions about the cluster like **scheduling** and **detecting** and **responding** to cluster events eg: starting up a new pod when a replication controllers replica field is unsatisfied.

Following are the components of Master Node:

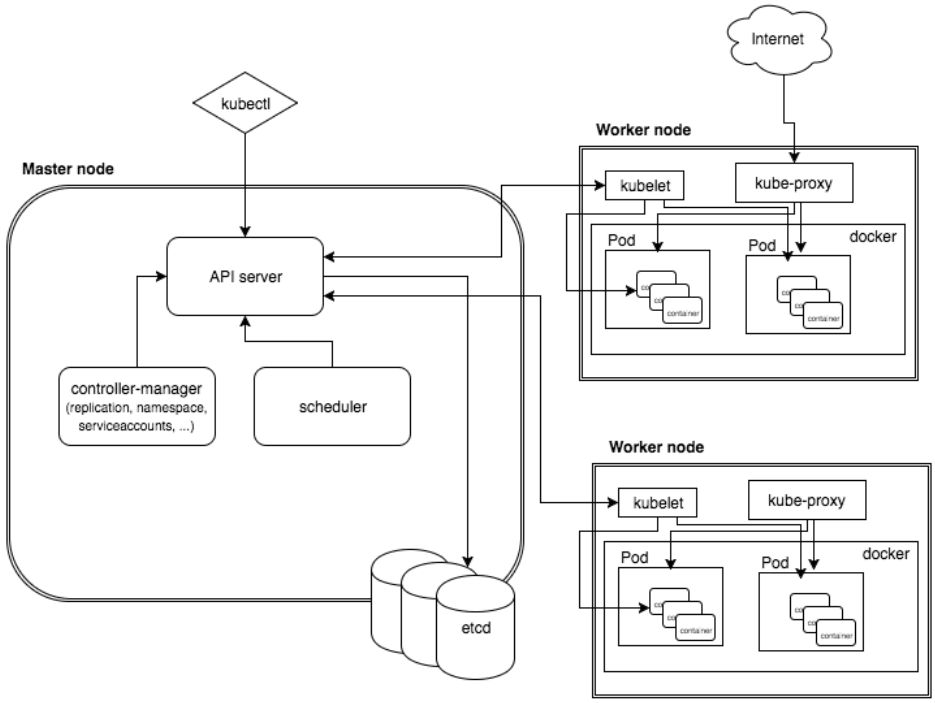
1. **API Server:** It's a frontend used by users, clients, tools and libraries with Kubernetes. It exposes REST API for health checking of other servers, allocating pods to available nodes, and orchestrating communication between other components.
2. **Etcd:** It is a high available **key-value store** that can be distributed among multiple nodes. It stores information about itself, pods, services etc.
3. **Scheduler**: It finds a suitable worker node for a newly created pod and services. The scheduler has the information regarding resources available on the members of the cluster, as well as the ones required for the

configured service to run and hence is able to decide where to deploy a specific service. Impacts the availability, performance and capacity of the system.

4. **Controller Manager:** It is a control loop that watches the shared state of the cluster through the API server and makes changes attempting to move the current state towards the **desired state**.
   It is responsible for some of the following:
   - Create one or more copies of pods using Pod templates
   - Autoscale pods up or down as appropriate
   - Can be used for rolling deployments

Kubernetes (Worker) Nodes Components

Worker node are the servers that perform work by running containers. Following are the key components of the Node server which are necessary to communicate with master components, configuring the container networking, and running the actual workloads assigned to them.



1. **Container Runtime:** The container runtime on each node is the component that runs the containers defined in the workloads submitted to the cluster. This can be docker or anything equivalent to it.
2. **Kubelet:** This is the main contact point for each node and is responsible for **communication with the master** component to receive commands and work. The Kubelet takes a set of PodSpecs (YAML or JSON object that describes the Pod) that are provided through apiserver and ensures that the containers described in those PodSpecs are running and healthy.

3. **Kubernetes Proxy Service:** This is a proxy service which runs on each node and helps in making services available to the end users/clients. It helps in forwarding the request to correct containers and is capable of performing primitive load balancing.
4. **cAdvisor**: Container advisor is a resource monitoring agent and provides information on resource usage and performance characteristics of the running containers. It collects, aggregates, processes, and exports information about running containers.

Kubernetes Objects and Workloads

Kubectl: Is the CLI: command line interface for running commands against the Kubernetes cluster

```
Kubectrl <operation> <object> <resource name> <optional flags>
```

Kubernetes Pods

- A *Pod* is the basic building block of Kubernetes – the smallest and simplest unit in the Kubernetes object model that you create or deploy. A Pod represents a running process on your cluster.
- Each Pod has one or more application containers (such as Docker).
- Querying a pod returns a data structure that contains information about containers and its metadata.
- Each Pod is meant to run a single instance of a given application. If you want to scale your application horizontally (e.g., run multiple instances), you should use multiple Pods, one for each instance. Replicated Pods are usually created and managed as a group by an abstraction called a **Replication Controller**.
- Pods aren't intended to be treated as durable entities. They won't survive scheduling failures, node failures, or other evictions, such as due to lack of resources, or in the case of node maintenance.

Deployments

- The Deployment instructs Kubernetes on how to create **and update instances** of your application. Once you've created a Deployment, the Kubernetes master **schedules** mentioned application instances onto individual Nodes in the cluster.
- Once the application instances are created, a Kubernetes Deployment Controller continuously monitors those instances. If the Node hosting an instance goes down or is deleted, the Deployment controller replaces it. This provides a self-healing mechanism to address machine failure or maintenance.
- It is possible to create an individual pod without using a deployment controller, but the pod will lack the mechanism needed to recreate itself without any manual intervention. That makes the process unsustainable.

Service

- Services enable communication between various components within and outside of the application.

- Although each Pod has a unique IP address, those IPs are not exposed outside the cluster without a Service. Services allow your applications to receive traffic.

Steps Required for Running an Application in Kubernetes

1. Package your Application in to one or more containers
2. Push those images to an image registry like Docker Hub or ACR
3. Post App Descriptor to the Kubernetes API Server
    a. Scheduler schedules containers on available Worker Nodes
    b. Kubelet instructs Nodes to download Container Images
    c. Kubelet instructs Nodes to run the Containers

Azure Kubernetes Service (AKS)

- AKS makes it quick and easy to deploy and manage containerized applications without container orchestration expertise.
- AKS reduces the complexity and operational overhead of managing Kubernetes by offloading much of that responsibility to Azure.
- As a hosted Kubernetes service, Azure handles critical tasks like provisioning, upgrading, scaling, health monitoring and maintenance for you.
- In addition, the service is free, you only pay for the agent nodes within your clusters, not for the masters.

Walkthrough

Step 1: Develop and Test the application locally

Step2: Deploy the docker images in Docker Hub

Step3: Create Kubernetes Cluster

Step4: Run the application developed in step 1

Step 1: Develop and Test the application locally

1. Execute the following commands to create an ASP.NET Core MVC project.

    D:\>md HelloWebApp

    D:\>cd HelloWebApp

    D:\HelloWebApp:\> dotnet new mvc

    D:\HelloWebApp:\> Code .

2. Add the following docker file to the project

```
FROM microsoft/dotnet:2.1-sdk
WORKDIR /app
COPY *.csproj .
```

```
RUN dotnet restore
COPY . .
RUN dotnet publish -c Release -o /app
ENTRYPOINT ["dotnet", "HelloWebApp.dll"]
```

3. Build the docker image

   D:\HelloWebApp>**docker build** -t sandeepsoni/hellowebapp .

4. Test the image locally

   D:\HelloWebApp>**docker run** -p "8080:80" sandeepsoni/hellowebapp

   Visit: http://localhost:8080

Step2: Deploy the docker images in docker hub

5. Login to docker hub

   D:\HelloWebApp>**docker login**

6. Push the image to Docker Registry (ensure that the image is public)

   D:\HelloWebApp>**docker push** sandeepsoni/hellowebapp

Step3: Create Kubernetes Cluster

7. Create a resource → Containers → Kubernetes Service
8. Basic tab
   a. Resource Group name = "DemoKRG"
   b. Kubernetes cluster name = "dssdemoapp"
   c. DNS name prefix = "dssdemoapp"
   d. Node count = 2
   e. Click Next: Authentication
9. Authentication Tab
   a. Service Principal = new default service principal
   b. Enable RBAC = yes
10. Click Review + create button

Login to Azure Subscription

11. D:\HelloWebApp>**az login**
12. D:\HelloWebApp>**powershell**

Configure kubectl to connect to your Kubernetes cluster.

13. D:\HelloWebApp>az **aks get-credentials** --resource-group DemoKRG --name dssdemoapp

14. Open Dashboard

    D:\HelloWebApp>**az aks browse** --resource-group DemoKRG --name dssdemoapp

You will notice that t**he ServiceAccount used by the dashboard does not have enough rights to access all resources.** The best solution is to give service account **kubernetes-dashboard,** rights to access the dashboard.

Create a ClusterRoleBinding which gives the role **dashboard-admin** to the ServiceAccount **kubernetes-dashboard**.

15. **kubectl create clusterrolebinding** kubernetes-dashboard -n kube-system --clusterrole=**cluster-admin** --serviceaccount=kube-system:**kubernetes-dashboard**

Step 4: Run the application in Kubernetes

Kubernetes manifest files define a desired state for a cluster, including what container images should be running.

16. Create a file DeploymentAndService.yaml

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: hellowebapp-deployment
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: hellowebapp
    spec:
      containers:
      - name: hellowebapp
        image: sandeepsoni/hellowebapp
        imagePullPolicy: Always
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: hellowebapp
spec:
  type: LoadBalancer
```

```
ports:
- port: 80
selector:
  app: hellowebapp
```

17. Install kubectl locally

    `az aks install-cli`

18. Use the kubectl create command to run the application.

    kubectl create -f deployment.yaml

Test the application

19. To monitor progress, use the the below command.

    kubectl get service hellowebapp –watch

    Once the *EXTERNAL-IP* address has changed from *pending* to an *IP address*, use `CTRL-C` to stop the kubectl watch process.

20. Open a web browser to the external IP address of your service