INTRODUCTION TO CLOUD SYSTEMS

Lecture 1 – Course introduction, Microservices, Docker

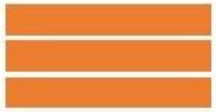
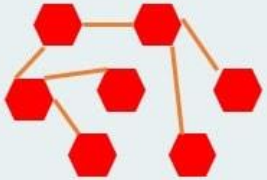
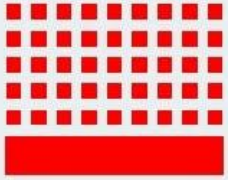
Agenda

- Course objectives
- Changes in business
- The monolith
- Problems of older applications
- Microservices
- Docker

Course objectives

- Microservices achitecture
- Docker and Kubernetes
- Alpine, UPI based images
- Kubernetes CNIs
- Service Mesh
- Server telemetry collecting and processing
- Dynamic scheduling
- Data processing acceleration
- Security Development Lifecycle
- Confidential Computing
- Out of band Management
- Smart Edge Open

Changes in business

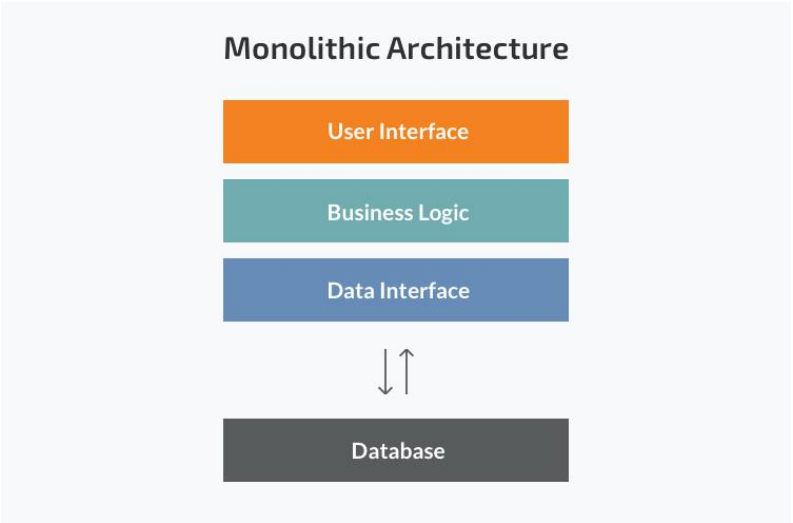
	Development Process	Application Architecture	Deployment and Packaging	Application Infrastructure
~ 1980 ~ 1990	Waterfall 	Monolithic 	Physical Server 	Datacenter 
~ 2000	Agile 	N-Tier 	Virtual Servers 	Hosted 
~ 2010 Now	DevOps 	Microservices 	Containers 	Cloud 

The monolith

Parts of Enterprise Applications:

- client-side user interface (rich desktop or web based)
- server-side application
- database (consisting of many tables inserted into a common, and usually relational, database management system).

The server-side application receives requests, executes domain logic, retrieves and/or updates data from the database, and responds back to the client. 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.



The monolith

Modularity within the application is typically based on features of the programming language (e.g., packages, modules). Over time the monolith grows larger as business needs change and as new functionality is added. It becomes increasingly difficult to keep a good modular structure, making it harder to keep changes that ought to only affect one module within that module.

In order to scale the application, we would simply create more instances of that process. It is not possible to scale the components independently.

Even a small change to the application requires that the entire monolith be rebuilt and deployed.

Problem of older applications

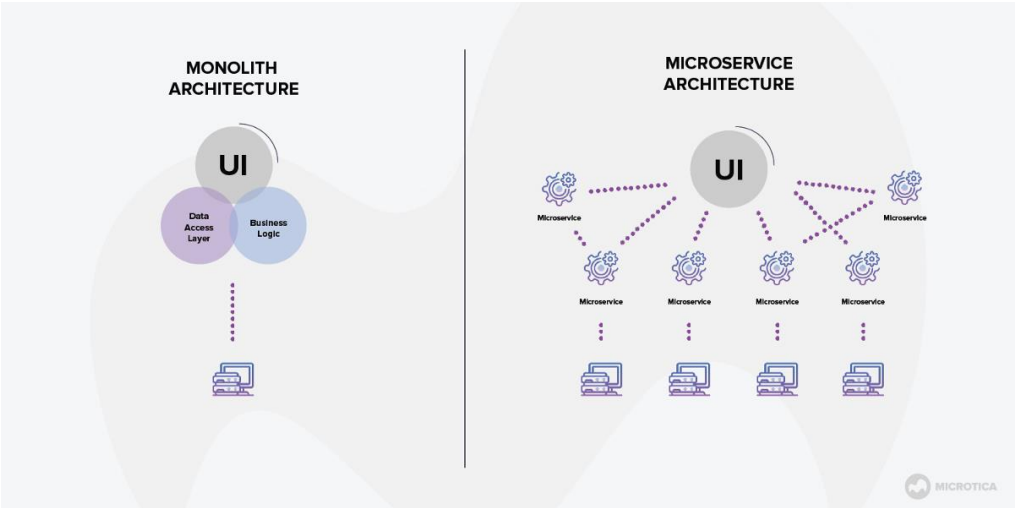
Many older cloud applications use a monolithic architecture.

Even though these legacy apps can serve multiple tenants, they're built as a large and cumbersome set of highly interdependent components.

- A failure in one component can have devastating impact on another component, resulting in service outages for many or all tenants.
- Updating these systems requires taking them offline, which limits user access during the upgrade process.
- The problems of monolithic services are exacerbated when they are deployed in proprietary data centers with limited hardware because the hardware constraints further limit the availability and scalability of the software resources.

Microservices

Microservices is an architecture that promotes breaking down a big monolithic application into smaller and simpler services (organized around business capabilities) to be built and deployed independently, communicating with lightweight mechanisms (often REST based).



The “micro” in microservices refers to the scope of the service’s functionality, not the number of Lines of Code.

Microservices

Key characteristics of microservices are:

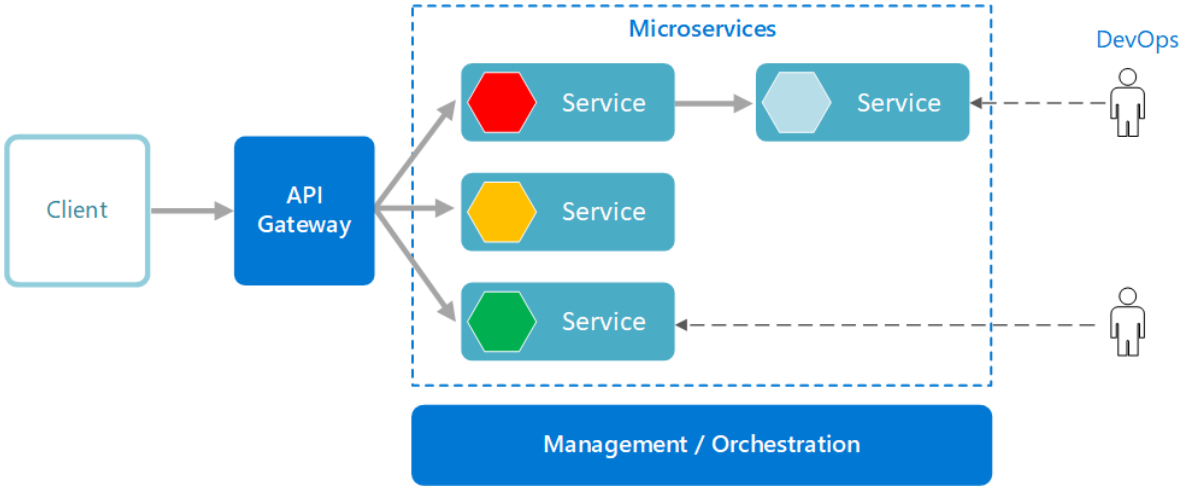
- **Domain-Driven Design**. Functional decomposition can be easily achieved using Eric Evans's DDD approach.
- **Single Responsibility Principle**. Each service is responsible for a single part of the functionality and does it well.
- **Explicitly Published Interface**. A producer service publishes an interface that is used by a consumer service.
- **Independent DURS** (Deploy, Update, Replace, Scale). Each service can be independently deployed, updated, replaced, and scaled.
- **Smart Endpoints and Dumb Pipes**. Each microservice owns its domain logic and communicates with others through simple protocols such as REST over HTTP

Microservices

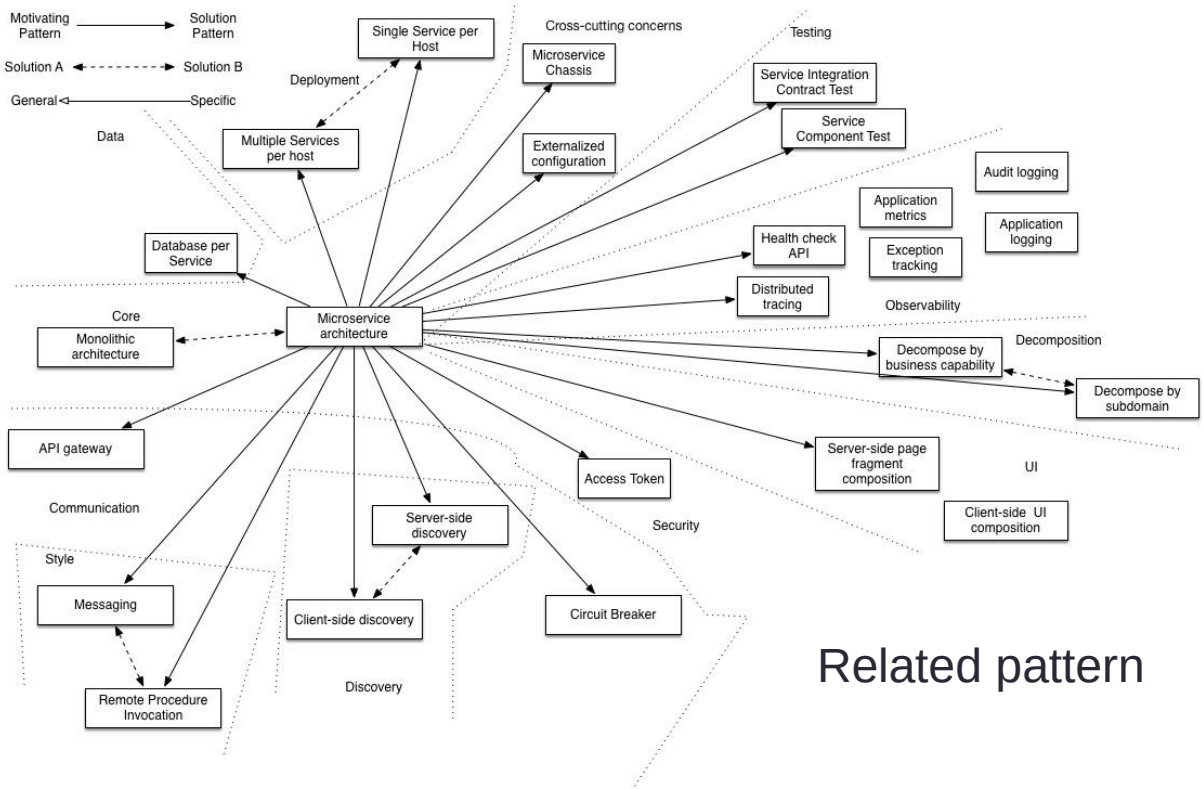
- **Agility.** Microservices foster an organization of small, independent teams that take ownership of their services. Teams act within a small and well understood context, and are empowered to work more independently and more quickly. This shortens development cycle times. You benefit significantly from the aggregate throughput of the organization.
- **Flexible Scaling.** Microservices allow each service to be independently scaled to meet demand for the application feature it supports. This enables teams to right-size infrastructure needs, accurately measure the cost of a feature, and maintain availability if a service experiences a spike in demand.
- **Easy Deployment** Microservices enable continuous integration and continuous delivery, making it easy to try out new ideas and to roll back if something doesn't work. The low cost of failure enables experimentation, makes it easier to update code, and accelerates time-to-market for new features.
- **Technological Freedom.** Microservices architectures don't follow a "one size fits all" approach. Teams have the freedom to choose the best tool to solve their specific problems. As a consequence, teams building microservices can choose the best tool for each job.
- **Reusable Code.** Dividing software into small, well-defined modules enables teams to use functions for multiple purposes. A service written for a certain function can be used as a building block for another feature. This allows an application to bootstrap off itself, as developers can create new capabilities without writing code from scratch.
- **Resilience Service.** independence increases an application's resistance to failure. In a monolithic architecture, if a single component fails, it can cause the entire application to fail. With microservices, applications handle total service failure by degrading functionality and not crashing the entire application.

Microservices

A microservices architecture consists of a collection of small, autonomous services. Each service is self-contained and should implement a single business capability within a bounded context. A bounded context is a natural division within a business and provides an explicit boundary within which a domain model exists.



Microservices

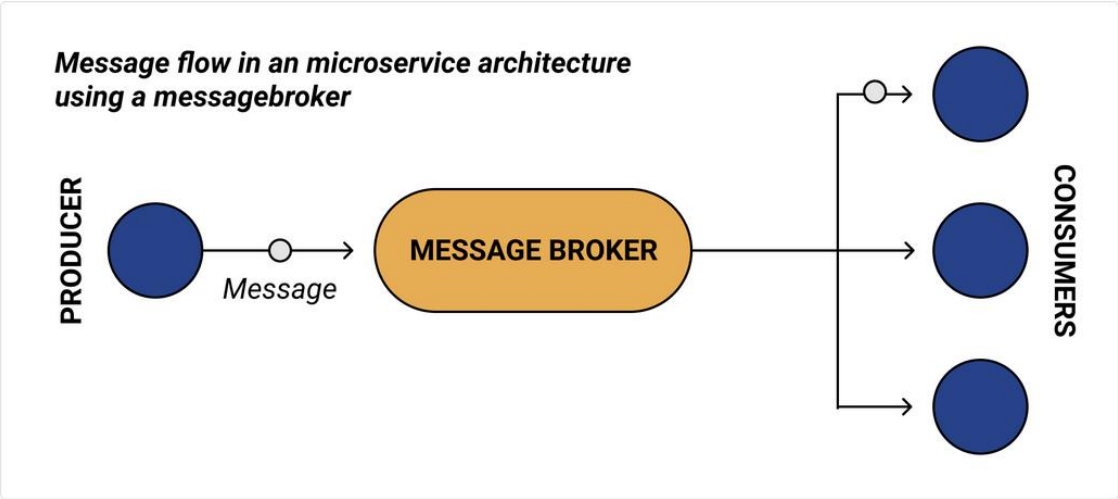


Related pattern

Microservices

Message queueing

The use of Message Queues provides a way for parts of the application to push messages to a queue asynchronously and ensure they are delivered to the correct destination. To implement message queuing, a message broker like RabbitMQ is a good option. The message broker provides temporary message storage when the receiving service is busy or disconnected.



Microservices

Asynchronous Messaging

Supports multiple messaging protocols, message queuing, delivery acknowledgement, flexible routing to queues, multiple exchange type.

Developer Experience

Deploy with BOSH, Chef, Docker and Puppet. Develop cross-language messaging with favorite programming languages such as: Java, .NET, PHP, Python, JavaScript, Ruby, Go, and many others.

Distributed Deployment

Deploy as clusters for high availability and throughput; federate across multiple availability zones and regions.

RabbitMQ

Enterprise & Cloud Ready

Pluggable authentication, authorisation, supports TLS and LDAP. Lightweight and easy to deploy in public and private clouds.

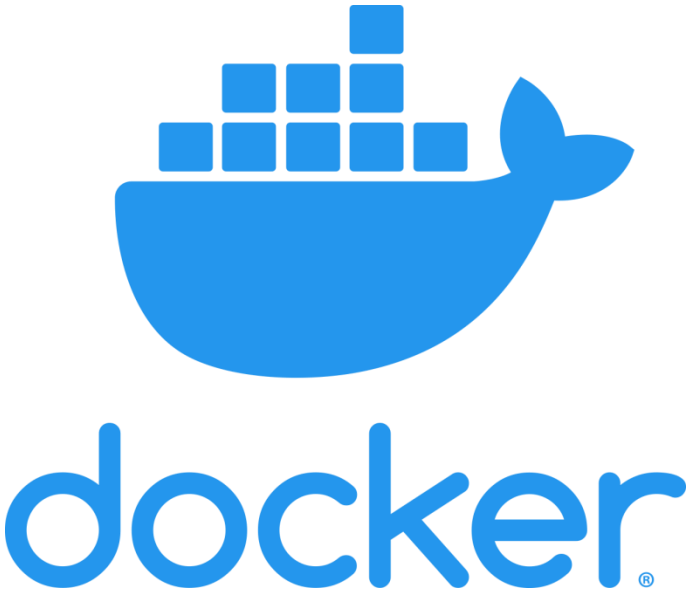
Tools & Plugins

Diverse array of tools and plugins supporting continuous integration, operational metrics, and integration to other enterprise systems. Flexible plug-in approach for extending RabbitMQ functionality.

Management & Monitoring

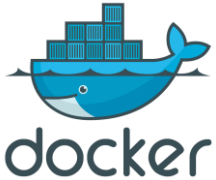
HTTP-API, command line tool, and UI for managing and monitoring RabbitMQ.

Docker



Docker is a set of platform as a service (PaaS) products that use OS-level virtualization to deliver software in packages called containers.

Docker
It is not…



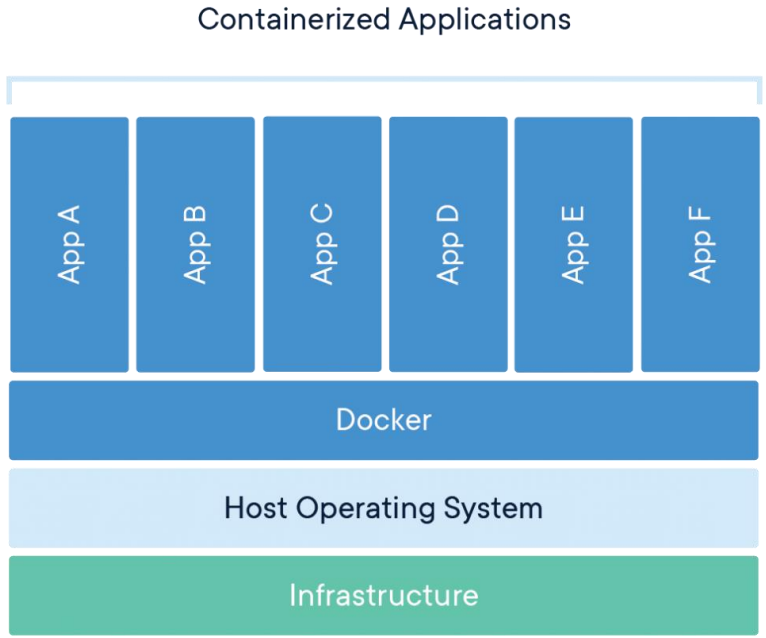
- **…a virtual machine, or virtualization framework.**
  - Like Vagrant, VirtualBox, VMWare
  - (Runs in a VM on OSX and Windows hosts.)

- **…SaaS, or other business service.**
  - Like Elastic Container Service (ECS), Docker Cloud, etc.

- **...a deployment or orchestration framework itself.**
  - Like Marathon, Consul, Puppet, or Chef
  - Docker Swarm provides something similar to this

Docker

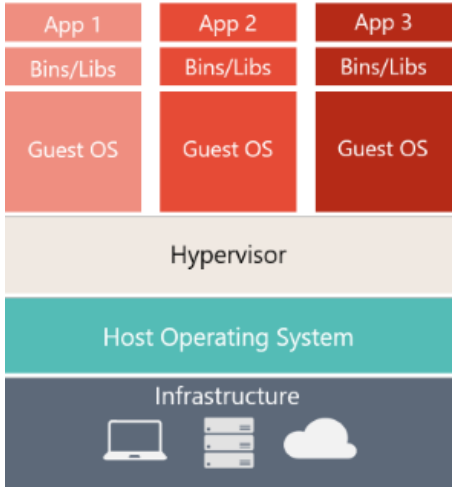
Docker platform enables building advanced environments based on OS-level virtualization technologies. There is no need to install components in the system such as SQL database, Wordpress, Postgresql, etc. You can download a ready-made software package with the required minimum to run. Application images prepared to run on Docker contain the runtime environment, most often it is the minimum version of Linux.



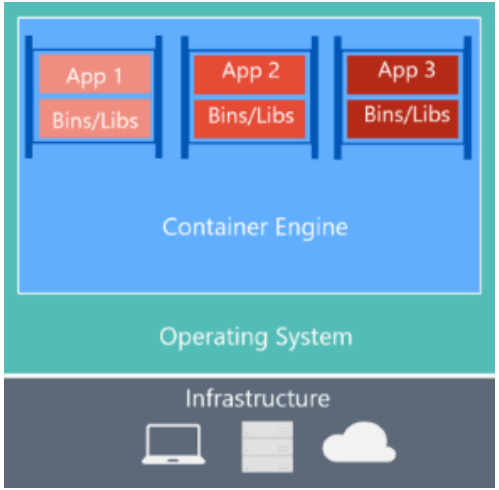
Docker

Virtual machines include the application, the required libraries or binaries, and a full guest operating system. Full virtualization requires more resources than containerization.

Virtual machines



Docker containers

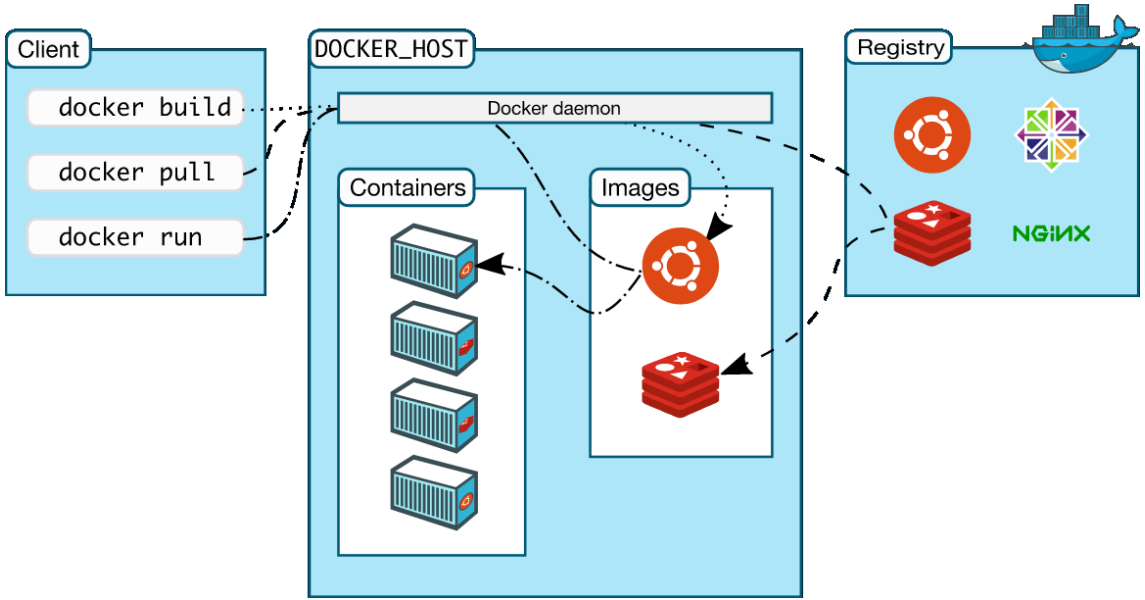


Containers include the application and all its dependencies. However, they share the OS kernel with other containers, running as isolated processes in user space on the host operating system. (Except in Hyper-V containers, where each container runs inside of a special virtual machine per container.)

Docker

Architecture

Docker uses a client-server architecture. The Docker *client* talks to the Docker *daemon*, which does the heavy lifting of building, running, and distributing your Docker containers. The Docker client and daemon *can* run on the same system, or you can connect a Docker client to a remote Docker daemon. The Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface. Another Docker client is Docker Compose, that lets you work with applications consisting of a set of containers.



Docker

Architecture

- The **Docker daemon** (dockerd) listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. A daemon can also communicate with other daemons to manage Docker services.
- The **Docker client** (docker) is the primary way that many Docker users interact with Docker. When you use commands such as docker run, the client sends these commands to dockerd, which carries them out. The docker command uses the Docker API. The Docker client can communicate with more than one daemon.
- **Docker Desktop** is an easy-to-install application for your Mac or Windows environment that enables you to build and share containerized applications and microservices. Docker Desktop includes the Docker daemon (dockerd), the Docker client (docker), Docker Compose, Docker Content Trust, Kubernetes, and Credential Helper. For more information, see Docker Desktop.
- A **Docker registry** stores Docker images. Docker Hub is a public registry that anyone can use, and Docker is configured to look for images on Docker Hub by default. You can even run your own private registry. When you use the docker pull or docker run commands, the required images are pulled from your configured registry. When you use the docker push command, your image is pushed to your configured registry.

Docker

- **Images** are “snapshots” of a file system
  - They are binary files.
  - They contain files, binaries and libraries added at “build time.”
  - Images that are just an operating system are **base images**

Base Images

fedora

centos

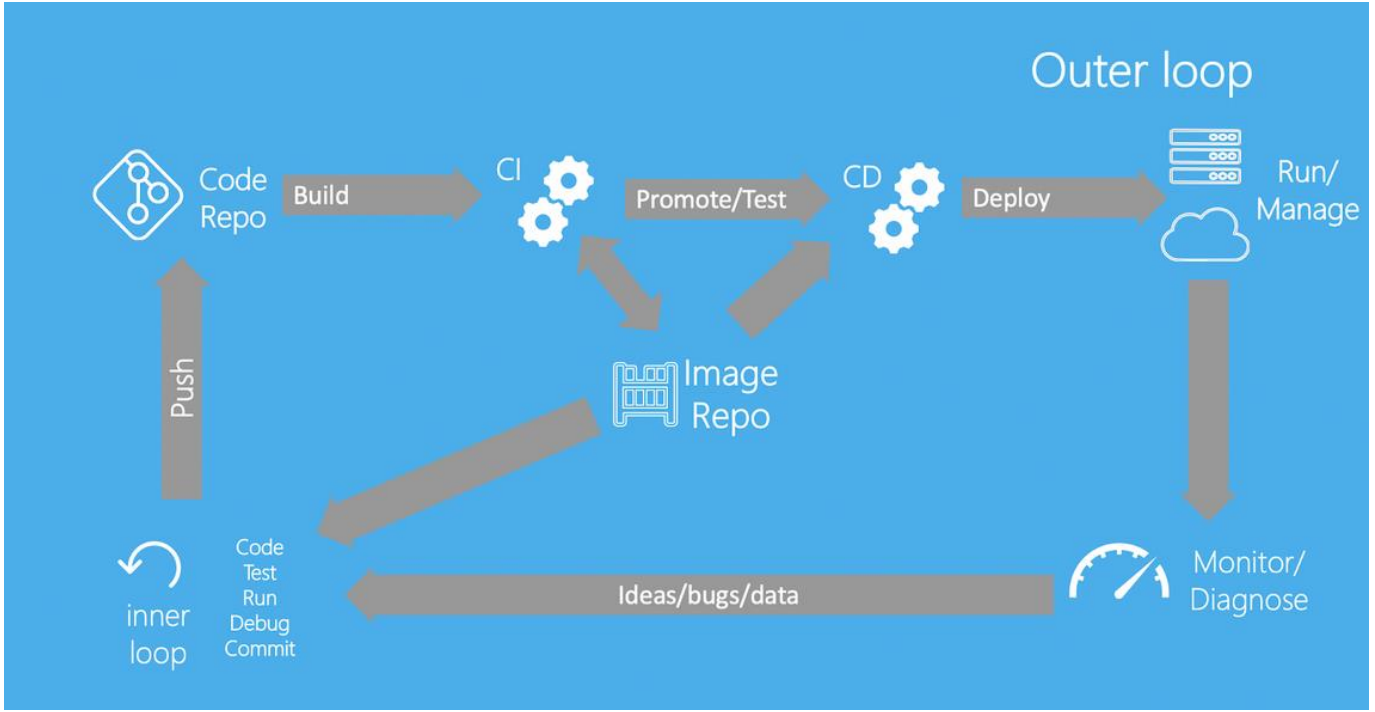
ubuntu

debian

busybox

Docker

CI/CD


Outer loop
Code Repo
Build
CI
Promote/Test
CD
Deploy
Run/ Manage
Push
Image Repo
inner loop
Code
Test
Run
Debug
Commit
Ideas/bugs/data
Monitor/ Diagnose

Docker

Commands

Docker Cheat Sheet

docker

Build

Build an image from the Dockerfile in the current directory and tag the image
`docker build -t myimage:1.0 .`

List all images that are locally stored with the Docker Engine
`docker image ls`

Delete an image from the local image store
`docker image rm alpine:3.4`

Share

Pull an image from a registry
`docker pull myimage:1.0`

Retag a local image with a new image name and tag
`docker tag myimage:1.0 myrepo/myimage:2.0`

Push an image to a registry
`docker push myrepo/myimage:2.0`

Run

Run a container from the Alpine version 3.9 image, name the running container "web" and expose port 5000 externally, mapped to port 80 inside the container.
`docker container run --name web -p 5000:80 alpine:3.9`

Stop a running container through SIGTERM
`docker container stop web`

Stop a running container through SIGKILL
`docker container kill web`

List the networks
`docker network ls`

List the running containers (add `--all` to include stopped containers)
`docker container ls`

Delete all running and stopped containers
`docker container rm -f $(docker ps -aq)`

Print the last 100 lines of a container's logs
`docker container logs --tail 100 web`

Docker Management

All commands below are called as options to the base `docker` command. `Run docker <command> --help` for more information on a particular command.

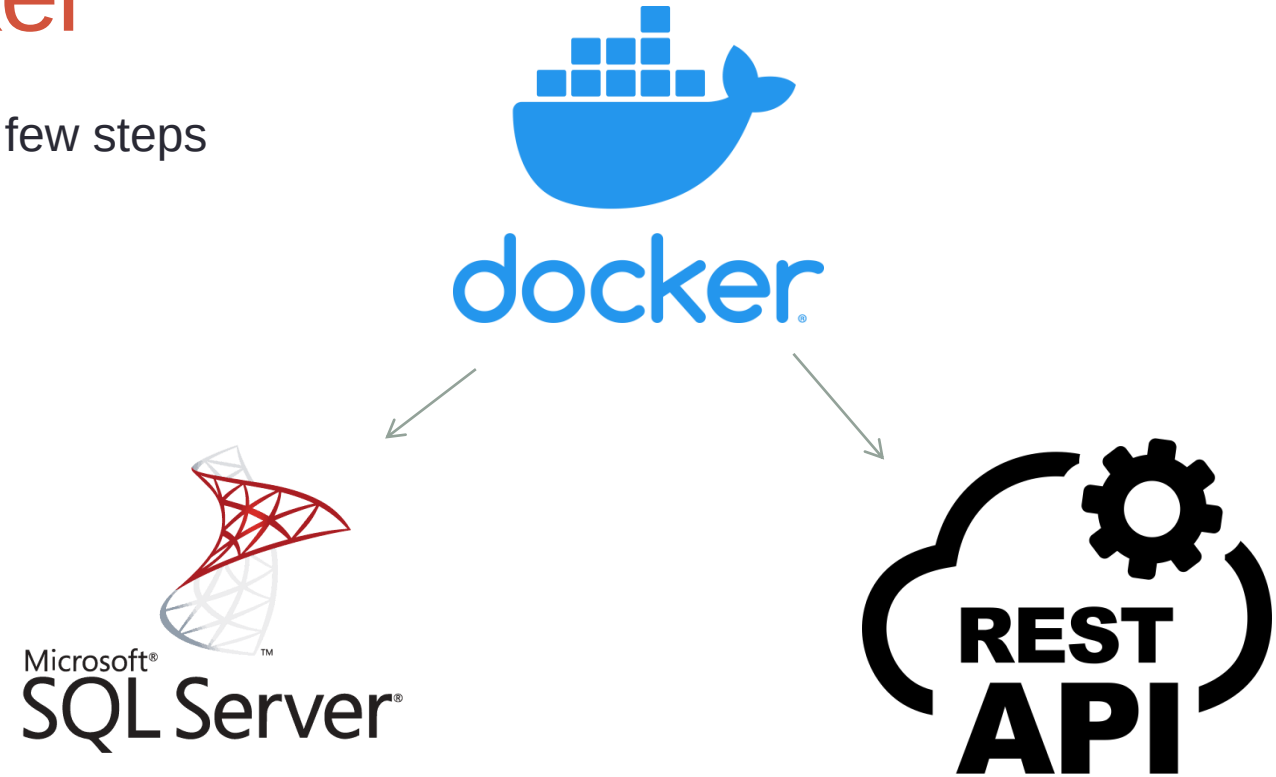
`app*`	*Docker Application*
`assemble*`	*Framework-aware builds (Docker Enterprise)*
`builder`	*Manage builds*
`cluster`	*Manage Docker clusters (Docker Enterprise)*
`config`	*Manage Docker configs*
`context`	*Manage contexts*
`engine`	*Manage the docker Engine*
`image`	*Manage images*
`network`	*Manage networks*
`node`	*Manage Swarm nodes*
`plugin`	*Manage plugins*
`registry*`	*Manage Docker registries*
`secret`	*Manage Docker secrets*
`service`	*Manage services*
`stack`	*Manage Docker stacks*
`swarm`	*Manage swarm*
`system`	*Manage Docker*
`template*`	*Quickly scaffold services (Docker Enterprise)*
`trust`	*Manage trust on Docker images*
`volume`	*Manage volumes*

*Experimental in Docker Enterprise 3.0.

www.docker.com

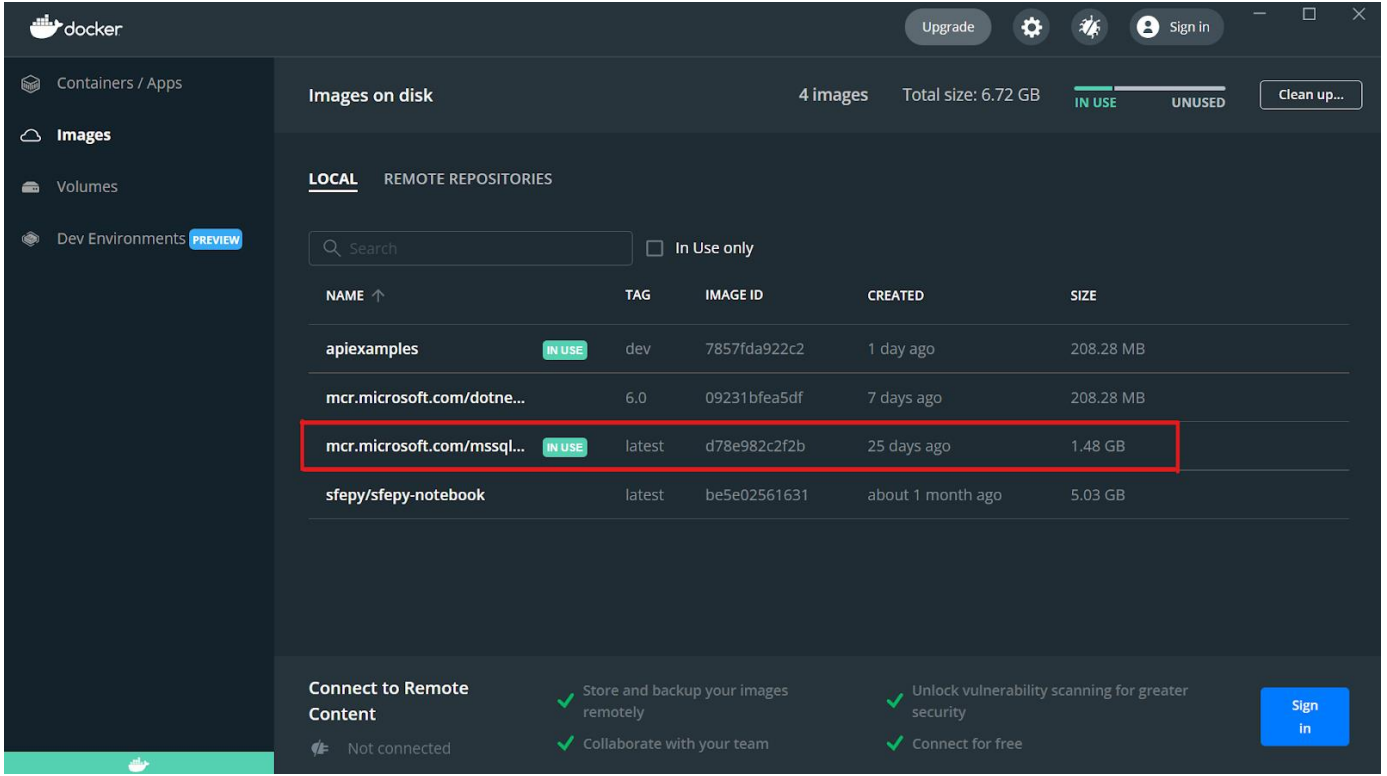
Docker

Example in few steps



Docker

1. Download MSSQL

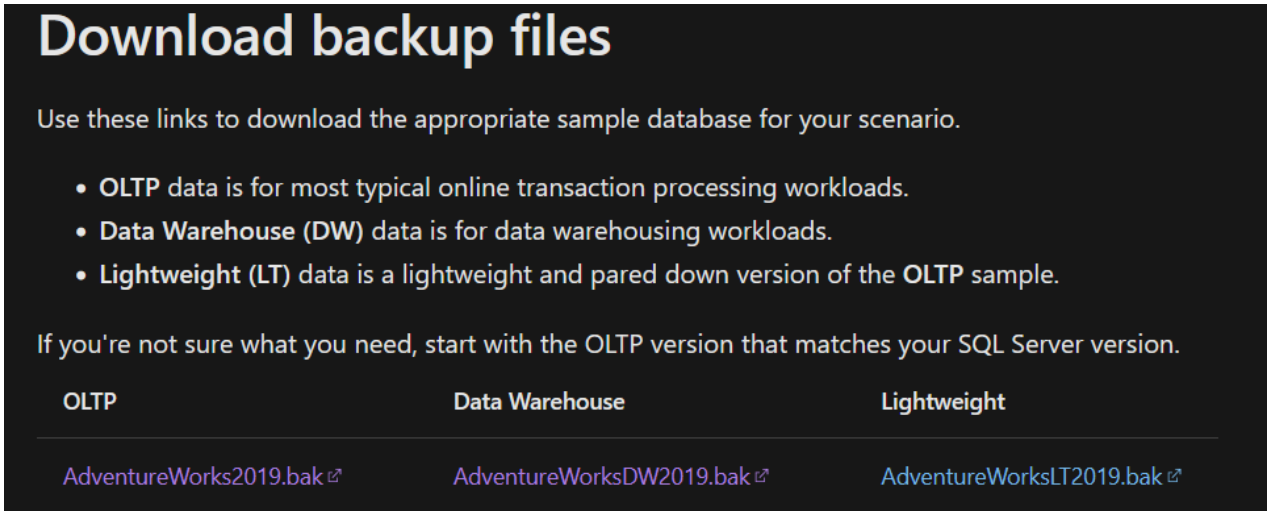


Docker

2. Run MSSQL

```
docker run -e 'ACCEPT_EULA=Y' -e
'SA_PASSWORD=***' -p 1433:1433 -d
mcr.microsoft.com/mssql/server
```

3. Download example data



Download backup files

Use these links to download the appropriate sample database for your scenario.

- **OLTP** data is for most typical online transaction processing workloads.
- **Data Warehouse (DW)** data is for data warehousing workloads.
- **Lightweight (LT)** data is a lightweight and pared down version of the **OLTP** sample.

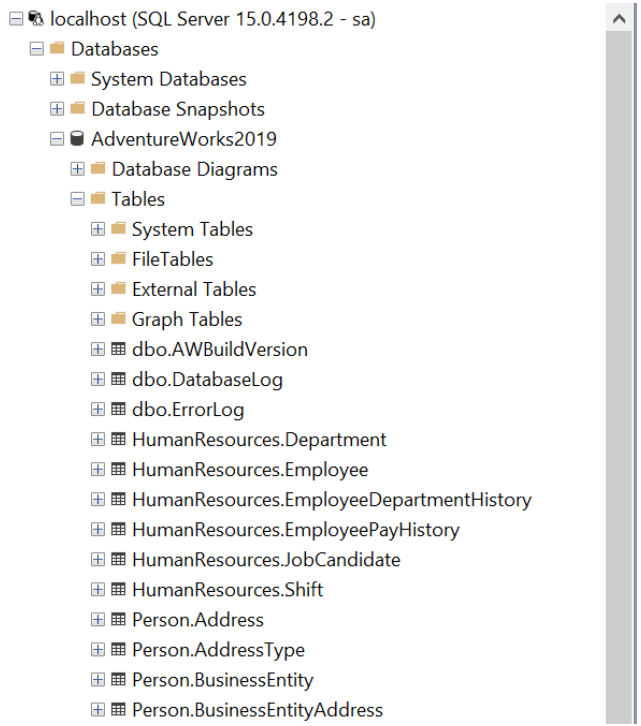
If you're not sure what you need, start with the OLTP version that matches your SQL Server version.

OLTP	Data Warehouse	Lightweight
AdventureWorks2019.bak	AdventureWorksDW2019.bak	AdventureWorksLT2019.bak

Docker

4. Backup data

Create backup folder in Docker container:
docker exec -it container_name mkdir /var/opt/mssql/backup
Copy database backup from Windows into container with Linux:
docker cp AdventureWorks2019.bak container_name:/var/opt/mssql/backup
From SQL Server Management Studio - restore backup



Docker

5. Create simple client

- Create new API project with Entity Framework - ApiExample
- Scaffold database with Docker SQL connection
- Add simple API endpoint - GetSalesOrderHeaderById(int id)

```
[HttpGet("GetSalesOrderHeaderById{id}")]
0 references
public IActionResult GetSalesHeader(int id)
{
    var result = _dataContext.SalesOrderHeader.Where(x => x.SalesOrderId == id);

    if (!result.Any())
    {
        return NotFound();
    }

    return Ok(result);
}
```

Docker

6. Create configuration file

Docker compose is configuration file for project. You can create relations between separate container like .NET API application hosted in Docker and container with database run in Docker too.

```
services:
  db:
    container_name: "mssql_docker"
    image: "mcr.microsoft.com/mssql/server"
    ports:
      - 1433:1433
    environment:
      - ACCEPT_EULA=Y
      - SA_PASSWORD=***

  apiexamples:
    container_name: "ApiExample"
    image: ${DOCKER_REGISTRY-}apiexamples
    build:
      context: .
      dockerfile: ApiExamples/Dockerfile
    depends_on:
      - db
```

Docker

Container use cases

- **“Lift and shift” existing applications into modern cloud architectures** - Some organizations use containers to migrate existing applications into more modern environments. While this practice delivers some of the basic benefits of operating system virtualization, it does not offer the full benefits of a modular, container-based application architecture.
- **Refactor existing applications for containers** - Although refactoring is much more intensive than lift-and-shift migration, it enables the full benefits of a container environment.
- **Develop new container-native applications** - Much like refactoring, this approach unlocks the full benefits of containers.
- **Provide better support for microservices architectures** - Distributed applications and microservices can be more easily isolated, deployed, and scaled using individual container building blocks.
- **Provide DevOps support for continuous integration and deployment (CI/CD)** - Container technology supports streamlined build, test, and deployment from the same container images.
- **Provide easier deployment of repetitive jobs and tasks** - Containers are being deployed to support one or more similar processes, which often run in the background, such as ETL functions or batch jobs.

Docker

Benefits of containers

- The primary advantage of containers, especially compared to a VM, is providing a level of abstraction that makes them lightweight and portable.
- Lightweight: Containers share the machine OS kernel, eliminating the need for a full OS instance per application and making container files small and easy on resources. Their smaller size, especially compared to virtual machines, means they can spin up quickly and better support cloud-native applications that scale horizontally.
- Portable and platform independent: Containers carry all their dependencies with them, meaning that software can be written once and then run without needing to be re-configured across laptops, cloud, and on-premises computing environments.
- Supports modern development and architecture: Due to a combination of their deployment portability/consistency across platforms and their small size, containers are an ideal fit for modern development and application patterns—such as DevOps, serverless, and microservices—that are built are regular code deployments in small increments.
- Improves utilization: Like VMs before them, containers enable developers and operators to improve CPU and memory utilization of physical machines. Where containers go even further is that because they also enable microservice architectures, application components can be deployed and scaled more granularly, an attractive alternative to having to scale up an entire monolithic application because a single component is struggling with load.