

**WA2894 Booz Allen Hamilton
Tech Excellence Modern
Software Development
Program - Phase 3**



Web Age Solutions
USA: 1-877-517-6540
Canada: 1-877-812-8887
Web: <http://www.webagesolutions.com>

The following terms are trademarks of other companies:

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

IBM, WebSphere, DB2 and Tivoli are trademarks of the International Business Machines Corporation in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.

For customizations of this book or other sales inquiries, please contact us at:

USA: 1-877-517-6540, email: getinfousa@webagesolutions.com

Canada: 1-877-812-8887 toll free, email: getinfo@webagesolutions.com

Copyright © 2023 Web Age Solutions

This publication is protected by the copyright laws of Canada, United States and any other country where this book is sold. Unauthorized use of this material, including but not limited to, reproduction of the whole or part of the content, re-sale or transmission through fax, photocopy or e-mail is prohibited. To obtain authorization for any such activities, please write to:

Web Age Solutions

220 Yonge Street, Suite 218B

Toronto, Ontario. M5B 2H1

Table of Contents

Chapter 1 - Docker Introduction.....	11
1.1 What is Docker.....	11
1.2 Where Can I Run Docker?.....	12
1.3 Installing Docker Container Engine.....	12
1.4 Docker Machine.....	12
1.5 Docker and Containerization on Linux.....	13
1.6 Linux Kernel Features: cgroups and namespaces.....	13
1.7 The Docker-Linux Kernel Interfaces.....	14
1.8 Docker Containers vs Traditional Virtualization.....	14
1.9 Docker Containers vs Traditional Virtualization.....	15
1.10 Docker Integration.....	15
1.11 Docker Services.....	16
1.12 Docker Application Container Public Repository.....	16
1.13 Competing Systems.....	17
1.14 Docker Command Line.....	17
1.15 Starting, Inspecting, and Stopping Docker Containers.....	17
1.16 Docker Volume.....	18
1.17 Dockerfile.....	18
1.18 Docker Compose.....	19
1.19 Using Docker Compose.....	19
1.20 Dissecting docker-compose.yml.....	20
1.21 Specifying services.....	20
1.22 Dependencies between containers.....	21
1.23 Injecting Environment Variables.....	21
1.24 runC Overview.....	21
1.25 runC Features.....	22
1.26 Using runC.....	22
1.27 Running a Container using runC.....	23
1.28 Summary.....	24
Chapter 2 - Introduction to Kubernetes.....	25
2.1 What is Kubernetes.....	25
2.2 What is a Container.....	25
2.3 Container – Uses.....	26
2.4 Container – Pros.....	27
2.5 Container – Cons.....	27
2.6 Composition of a Container.....	28
2.7 Control Groups.....	28
2.8 Namespaces.....	28
2.9 Union Filesystems.....	29
2.10 Popular Containerization Software.....	30
2.11 Microservices.....	31
2.12 Microservices and Containers / Clusters.....	31
2.13 Microservices and Orchestration.....	32
2.14 Microservices and Infrastructure-as-Code.....	32

2.15 Kubernetes Container Networking.....	33
2.16 Kubernetes Networking Options.....	33
2.17 Kubernetes Networking – Balanced Design.....	34
2.18 Summary.....	35
Chapter 3 - Kubernetes – From the Firehose.....	37
3.1 What is Kubernetes?.....	37
3.2 Container Orchestration.....	38
3.3 Kubernetes Basic Architecture.....	38
3.4 Kubernetes Detailed Architecture.....	39
3.5 Kubernetes Concepts.....	39
3.6 Kubernetes Concepts (contd.).....	40
3.7 Cluster and Namespace.....	40
3.8 Node.....	41
3.9 Master.....	41
3.10 Pod.....	42
3.11 Label.....	42
3.12 Annotation.....	43
3.13 Label Selector.....	43
3.14 Replication Controller and Replica Set.....	44
3.15 Service.....	44
3.16 Persistent Volumes & Storage Class.....	45
3.17 Persistent Volumes & Storage Class.....	45
3.18 Secret.....	45
3.19 ConfigMaps.....	46
3.20 Custom Resource Definitions (CRDs).....	46
3.21 Operators.....	46
3.22 Resource Quota.....	47
3.23 Authentication and Authorization.....	48
3.24 Routing.....	48
3.25 Registry.....	49
3.26 Using Docker Registry.....	50
3.27 Summary.....	50
Chapter 4 - Kubernetes and the Twelve Factors.....	51
4.1 Kubernetes and the Twelve Factors - 1 Codebase.....	51
4.2 Kubernetes and the Twelve Factors - 2 Dependencies.....	51
4.3 Kubernetes and the Twelve Factors - 3 Config.....	51
4.4 Kubernetes and the Twelve Factors - 4 Backing Services.....	52
4.5 Kubernetes and the Twelve Factors - 5 Build, Release, Run.....	52
4.6 Kubernetes and the Twelve Factors - 6 Processes.....	52
4.7 Kubernetes and the Twelve Factors - 7 Port Binding.....	53
4.8 Kubernetes and the Twelve Factors - 8 Concurrency.....	53
4.9 Kubernetes and the Twelve Factors - 9 Disposability.....	54
4.10 Kubernetes and the Twelve Factors - 10 Dev/Prod Parity.....	54
4.11 Kubernetes and the Twelve Factors - 11 Logs.....	54
4.12 Kubernetes and the Twelve Factors - 12 Admin Processes.....	55
Chapter 5 - Leading Practices for Microservice Logging.....	57

5.1 Logging Challenges.....	57
5.2 Leading Practices.....	57
5.3 Correlate Requests with a Unique ID.....	58
5.4 Include a Unique ID in the Response.....	58
5.5 Send Logs to a Central Location.....	58
5.6 Structure Your Log Data.....	59
5.7 Add Context to Every Record.....	59
5.8 Examples of Content.....	60
5.9 Write Logs to Local Storage.....	61
5.10 Collecting Logs with Fluentd.....	61
5.11 Leading Practices for Microservice Logging Summary.....	62
5.12 Metrics Using Prometheus.....	62
5.13 Overview.....	63
5.14 Prometheus.....	63
5.15 Prometheus.....	63
5.16 Prometheus.....	64
5.17 Prometheus Architecture.....	65
5.18 Service Discovery.....	65
5.19 File-based Service Discovery.....	66
5.20 Istio and Prometheus.....	67
5.21 Exposing Metrics in Services.....	68
5.22 Exposing Metrics in Services.....	69
5.23 Exposing Metrics in Services.....	70
5.24 Exposing Metrics in Services.....	70
5.25 Exposing Metrics in Services.....	71
5.26 Exposing Metrics in Services.....	73
5.27 Exposing Metrics in Services.....	74
5.28 Querying in Prometheus.....	74
5.29 Querying in Prometheus.....	75
5.30 Querying in Prometheus.....	75
5.31 Grafana.....	76
5.32 Grafana.....	77
5.33 Grafana.....	77
5.34 Grafana.....	78
5.35 Grafana.....	79
5.36 Grafana.....	80
5.37 Business Metrics.....	80
5.38 Metrics Using Prometheus Summary.....	81
5.39 Tracing Using Jaeger.....	81
5.40 OpenTracing.....	82
5.41 OpenTracing.....	82
5.42 OpenTracing.....	83
5.43 OpenTracing.....	83
5.44 OpenTracing.....	84
5.45 Jaeger.....	85
5.46 Jaeger.....	85

5.47 Jaeger Architecture Diagram.....	86
5.48 Jaeger Client Libraries.....	86
5.49 Jaeger Sampling.....	87
5.50 Jaeger Agent.....	87
5.51 Jaeger Collector.....	88
5.52 Query and Ingester Services.....	88
5.53 Jaeger UI Example.....	89
5.54 Jaeger and Prometheus.....	89
5.55 Jaeger and Istio.....	90
5.56 Tracing Using Jaeger Summary.....	90
5.57 Summary.....	90
Chapter 6 - Shared Libraries.....	93
6.1 Extending with Shared Libraries.....	93
6.2 Directory Structure.....	93
6.3 Sample Groovy Code.....	94
6.4 Defining Shared Libraries.....	94
6.5 Using Shared Libraries.....	95
6.6 Sample Shared Library Usage Code.....	95
6.7 Defining Global Variables.....	96
6.8 Solutions Delivery Platform.....	96
6.9 SDP - Components.....	97
6.10 Jenkins Templating Plugin.....	97
6.11 Why Templating?.....	98
6.12 Why Templating? (Contd.).....	98
6.13 Pipeline Templating - Templates.....	99
6.14 Pipeline Templating - Steps.....	100
6.15 Pipeline Templating - Primitives.....	100
6.16 Pipeline Templating - Configuration Files.....	102
6.17 Pipeline Libraries.....	103
6.18 SDP Pipeline Libraries - Examples.....	103
6.19 Summary.....	103
Chapter 7 - Best Practices for Jenkins [OPTIONAL].....	105
7.1 Best Practices - Secure Jenkins.....	105
7.2 Best Practices - Users.....	105
7.3 Best Practices - Backups.....	106
7.4 Best Practices - Reproducible Builds.....	106
7.5 Best Practices - Testing and Reports.....	107
7.6 Best Practices - Large Systems.....	107
7.7 Best Practices - Distributed Jenkins.....	108
7.8 Best Practices - Summary.....	108
Chapter 8 - CI/CD with Docker, Kubernetes, Jenkins, and Blue Ocean [OPTIONAL]..	111
8.1 Jenkins Continuous Integration.....	111
8.2 Jenkins Features.....	111
8.3 Running Jenkins.....	112
8.4 Downloading and Installing Jenkins.....	112
8.5 Running Jenkins as a Stand-Alone Application.....	112

8.6 Running Jenkins on an Application Server.....	113
8.7 Installing Jenkins as a Windows Service.....	114
8.8 Different types of Jenkins job.....	115
8.9 Configuring Source Code Management(SCM).....	116
8.10 Working with Subversion.....	116
8.11 Working with Subversion (cont'd).....	117
8.12 Working with Git.....	117
8.13 Build Triggers.....	118
8.14 Schedule Build Jobs.....	119
8.15 Polling the SCM.....	120
8.16 Maven Build Steps.....	120
8.17 Jenkins / Kubernetes Pipeline.....	121
8.18 Jenkins / Kubernetes Pipeline Output.....	122
8.19 The Blue Ocean Plugin.....	122
8.20 Blue Ocean Plugin Features.....	122
8.21 Installing Jenkins Plugins.....	122
8.22 New modern user experience.....	123
8.23 Advanced Pipeline visualizations with built-in failure diagnosis.....	123
8.24 Branch and Pull Request awareness.....	124
8.25 Personalized View.....	124
8.26 OpenShift Pipeline Output.....	125
8.27 Creating OpenShift Blue Ocean Pipeline.....	125
8.28 Summary.....	126
Appendix 1 - [Appendix] Operational Readiness.....	127
1.1 What is Operational Readiness.....	127
1.2 Telemetry.....	127
1.3 End-to-end Requirements Traceability.....	128
1.4 Log Strategy.....	129
1.5 Monitoring Strategy.....	129
1.6 Runbooks.....	130
1.7 Operational Readiness Checklist.....	130
1.8 Strategy.....	131
1.9 Planning.....	131
1.10 Architecture.....	131
1.11 Design.....	131
1.12 Development (Acquire/Build).....	132
1.13 Transition.....	132
1.14 Acceptance.....	133
1.15 Evaluation.....	133
1.16 Summary.....	133
Appendix 2 - [Appendix] Reusable Design Patterns from BAH.....	135
2.1 Code Reviews for Every Pull Request.....	135
2.2 Code Reviews for Every Pull Request (Contd.).....	136
2.3 Code Reviews for Every Pull Request - Implementation Overview.....	137
2.4 Pull Request Template.....	137
2.5 Setting Up the Design for a 12 Factor Application.....	138

2.6.....	138
2.7 The Twelve-Factor App.....	139
2.8 Categorizing the Twelve Factors.....	139
2.9 I. Codebase.....	140
2.10 II. Dependencies.....	140
2.11 III. Config.....	140
2.12 IV. Backing Services.....	141
2.13 V. Build, Release, Run.....	141
2.14 VI. Processes.....	142
2.15 VII. Port Binding.....	142
2.16 VIII. Concurrency.....	142
2.17 IX. Disposability.....	143
2.18 X. Dev/Prod Parity.....	143
2.19 XI. Logs.....	143
2.20 XII. Admin Processes.....	144
2.21 Summary.....	144
Appendix 3 - [Appendix] Introduction to Kafka.....	145
3.1 Introduction.....	145
3.2 Messaging Architectures – What is Messaging?.....	145
3.3 Messaging Architectures – Steps to Messaging.....	146
3.4 Messaging Architectures – Messaging Models.....	146
3.5 What is Apache Kafka?.....	147
3.6 What is Apache Kafka? (Contd.).....	147
3.7 Who Uses Kafka?.....	148
3.8 Kafka Overview.....	148
3.9 Kafka Overview (Contd.).....	149
3.10 Need for Kafka.....	149
3.11 When to Use Kafka?.....	150
3.12 When to Use Kafka? (Contd.).....	150
3.13 Kafka Architecture.....	151
3.14 Core concepts in Kafka.....	151
3.15 Kafka Topic.....	152
3.16 Kafka Topic (Contd.).....	152
3.17 Topic Example - ambulances_location.....	153
3.18 Kafka Partitions.....	153
3.19 Kafka Producers.....	154
3.20 Kafka Producers - Acknowledgment Modes.....	155
3.21 Kafka Consumers.....	155
3.22 Consumer Groups.....	156
3.23 Consumer Offsets.....	157
3.24 Kafka Broker.....	158
3.25 Kafka Broker and Replication.....	158
3.26 Connecting to a Kafka Broker.....	159
3.27 Kafka Cluster.....	159
3.28 Why Kafka Cluster?.....	160
3.29 Sample Multi-Broker Cluster.....	160

3.30 Kafka Broker, Leader, and ISR.....	161
3.31 Overview of ZooKeeper.....	161
3.32 Overview of Zookeeper (Contd.).....	161
3.33 Kafka Cluster & ZooKeeper.....	162
3.34 Zookeeper Leader and Followers.....	162
3.35 Summary.....	163
Appendix 4 - [Appendix] Overview of the Amazon Web Services (AWS).....	165
4.1 Amazon Web Services.....	165
4.2 The History of AWS.....	165
4.3 The Initial Iteration of Moving amazon.com to AWS.....	166
4.4 The AWS (Simplified) Service Stack.....	167
4.5 Accessing AWS.....	167
4.6 Direct Connect.....	167
4.7 Shared Responsibility Model.....	168
4.8 Trusted Advisor.....	168
4.9 The AWS Distributed Architecture.....	168
4.10 AWS Services.....	169
4.11 Managed vs Unmanaged Amazon Services.....	169
4.12 Amazon Resource Name (ARN).....	170
4.13 Compute and Networking Services.....	170
4.14 Elastic Compute Cloud (EC2).....	171
4.15 AWS Lambda.....	171
4.16 Auto Scaling.....	172
4.17 Elastic Load Balancing (ELB).....	173
4.18 Virtual Private Cloud (VPC).....	174
4.19 Route53 Domain Name System.....	174
4.20 Elastic Beanstalk.....	175
4.21 Security and Identity Services.....	175
4.22 Identity and Access Management (IAM).....	175
4.23 AWS Directory Service.....	176
4.24 AWS Certificate Manager.....	176
4.25 AWS Key Management Service (KMS).....	176
4.26 Storage and Content Delivery.....	176
4.27 Elastic Block Storage (EBS).....	176
4.28 Simple Storage Service (S3).....	177
4.29 Glacier.....	177
4.30 CloudFront Content Delivery Service.....	178
4.31 Database Services.....	178
4.32 Relational Database Service (RDS).....	178
4.33 DynamoDB.....	179
4.34 Amazon ElastiCache.....	179
4.35 Redshift.....	180
4.36 Messaging Services.....	180
4.37 Simple Queue Service (SQS).....	180
4.38 Simple Notifications Service (SNS).....	181
4.39 Simple Email Service (SES).....	181

4.40 AWS Monitoring with CloudWatch.....181

4.41 Other Services Example.....182

4.42 Summary.....182

Chapter 1 - Docker Introduction

Objectives

Key objectives of this chapter

- Docker introduction
- Linux cgroups and namespaces
- Docker vs traditional virtualization
- Docker command-line
- runC

1.1 What is Docker



- Docker is an open-source (and 100% free) project for IT automation
- You can view Docker as a system or a platform for creating virtual environments which are extremely lightweight virtual machines
- Docker allows developers and system administrators to quickly assemble, test, and deploy applications and their dependencies inside Linux containers supporting the multi-tenancy deployment model on a single host
- Docker's lightweight containers lend themselves to rapid scaling up and down
 - ◇ *Note:* A container is a group of controlled processes associated with a separate tenant executed in isolation from other tenants
- Written in the Go programming language

Notes:

The Go programming language (also referred to as *golang*) was developed at Google in 2007 and release in 2009. It is a compiled language – it does not require a VM to run it (like in C# or Java) – with automated garbage collection. Go offers a balance between type safety and dynamic type

capabilities; it supports imperative and concurrent programming paradigms.

1.2 Where Can I Run Docker?

- Docker runs on any modern-kernel 64-bit Linux distributions
- The minimum supported kernel version is 3.10
 - ◇ Kernels older than 3.10 lack some of the features required by Docker containers
- You can install Docker on VirtualBox and run it on OS X or Windows
- Docker can be installed natively on Windows using Docker Machine, but requires Hyper-V
- Docker can be booted from the small footprint Linux distribution *boot2docker*

1.3 Installing Docker Container Engine

- Installing on Linux:
 - ◇ Docker is usually available via the package manager of the distributions
 - ◇ For example, on Ubuntu and derivatives:

```
sudo apt-get update && sudo apt install docker.io
```

- Installing on Mac
 - ◇ Download and install the official Docker.dmg from docker.com
- Installing on Windows
 - ◇ Hyper-V must be enabled on Windows
 - ◇ Download the latest installer from docker.com

1.4 Docker Machine

- Though Docker runs natively on Linux, it may be desirable to have two different host environment, such as Ubuntu and CentOS
- To achieve this, VMs running Docker may be used

- To simplify management of different Docker host, it is possible to use Docker Machine
- Docker Machine is a tool that lets you install Docker Engine on virtual hosts, and manage the hosts with `docker-machine` commands
- Docker Machine enables you to provision multiple remote Docker hosts on various flavors of Linux.
- Additionally, Machine allows you to run Docker on older Mac or Windows systems as well as cloud providers such as AWS, Azure and GCP
- Using the `docker-machine` command, you can start, inspect, stop, and restart a managed host, upgrade the Docker client and daemon, and configure a Docker client to talk to your host

1.5 Docker and Containerization on Linux

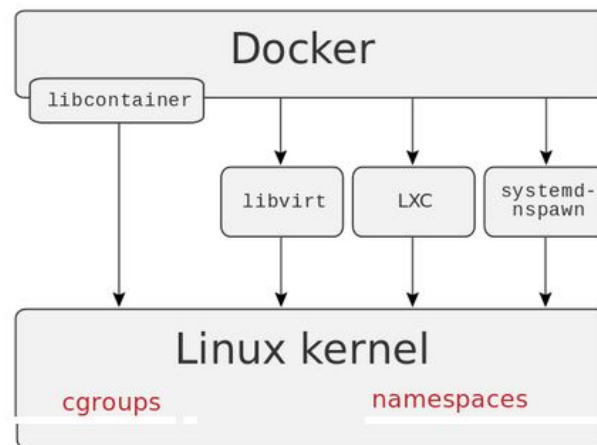
- Docker leverages resource isolation features of the modern Linux kernel offered by `cgroups` and kernel namespaces
 - ◇ The `cgroups` and kernel namespaces features allow creation of strongly isolated containers acting as very lightweight virtual machines running on a single Linux host
- Docker helps abstract operating-system-level virtualization on Linux using abstracted virtualization interfaces based on *libvirt*, *LXC* (Linux Containers) and `systemd-nspawn`
 - ◇ As of version 0.9, Docker has the capability to directly use virtualization facilities provided by the Linux kernel via its own *libcontainer* library

1.6 Linux Kernel Features: `cgroups` and namespaces

- The control group kernel feature (`cgroup`) is used by the Linux kernel to allocate system resources such as CPU, I/O, memory, and network subject to limits, quotas, prioritization, and other control arrangements
- The kernel provides access to multiple subsystems through the *cgroup* interface
 - ◇ Examples of subsystems (controllers):

- The memory controller for limiting memory use
- The *cpuacct* controller for keeping track of CPU usage
- The cgroups facility was merged into the Linux kernel version 2.6.24
- Systems that use cgroups: Docker, Linux Containers (LXC), Hadoop, etc.
- The namespaces feature is a related to cgroups facility that enables different applications to act as separate tenants with completely isolated views of the operating environment, including users, process trees, network, and mounted file systems

1.7 The Docker-Linux Kernel Interfaces



Source: Adapted from [http://en.wikipedia.org/wiki/Docker_\(software\)](http://en.wikipedia.org/wiki/Docker_(software))

1.8 Docker Containers vs Traditional Virtualization

- System virtualization tools or emulators such as VirtualBox, Hyper-V or VMware, boot virtual machines from a complete guest OS image (of your choice) and emulate a complete machine, which results in a high operational overhead
- Virtual environments created by Docker run on the existing operating system kernel of the host's OS without a need for a hypervisor
 - ◇ This leads to very low overhead and significantly faster container start-up time

- Docker-provisioned containers do not include or require a separate operating system (it runs in the host's OS)
 - ◇ This circumstance puts a significant limitation on your OS choices

1.9 Docker Containers vs Traditional Virtualization

- Overall, traditional virtualization has advantages over Docker in that you have a choice of guest OSes (as long as the machine architecture is supported)
 - ◇ You can get only some (limited) choice of Linux distros
 - You still have some choice: e.g. you can deploy a Fedora container on a Debian host
 - ◇ You can, however, run a Windows VM inside a Linux machine using virtual machine emulators like VirtualBox (with less engineering efficiency)
- With Linux containers, you can achieve a higher level of deployed application density compared with traditional VMs (10x more units!)
- Docker runs everything through a central daemon which is not a particularly reliable and secure processing model

1.10 Docker Integration

- Docker can be integrated with a number of IT automation tools that extend its capabilities, including
 - ◇ Ansible
 - ◇ Chef
 - ◇ Jenkins
 - ◇ Puppet
 - ◇ Salt
- Docker is also deployed on a number of Cloud platforms
 - ◇ Amazon Web Services

















- ◇ Google Cloud Platform
- ◇ Microsoft Azure
- ◇ OpenStack
- ◇ Rackspace

1.11 Docker Services

- Docker deployment model is application-centric and in this context provides the following services and tools:
 - ◇ A uniform format for bundling an application along with its dependencies which is portable across different machines
 - ◇ Tools for automatic assembling a container from source code: make, maven, Debian packages, RPMs, etc.
 - ◇ Container versioning with deltas between versions

1.12 Docker Application Container Public Repository

- Docker community maintains the repository for official and public domain Docker application images: <https://hub.docker.com>

 postgres  The PostgreSQL object-relational database system provides reliability and data integrity.	2 days ago	 593	 906990
 mongo  MongoDB document databases provide high availability and easy scalability.	21 hours ago	 518	 1076836
 mysql  MySQL is a widely used, open-source relational database management system (RDBMS).	2 days ago	 553	 3368805
 redis  Redis is an open source key-value store that functions as a data structure server.	2 days ago	 599	 2053739

1.13 Competing Systems

- Rocket container runtime from CoreOS (an open source lightweight Linux kernel-based operating system)
- LXD for Ubuntu from Canonical (the company behind Ubuntu)
- The LXC (Linux Containers), used by Docker internally
- Many more are on the way ...
- Other systems exist for non-Linux OSes

1.14 Docker Command Line

- The following commands are shown as executed by the root (privileged) user:

docker run ubuntu echo 'Yo Docker!'

- ◇ This command will create a docker container based on the *ubuntu* image, execute the *echo* command on it, and then shuts down

docker ps -a

- ◇ This command will list all the containers created by Docker along with their IDs

1.15 Starting, Inspecting, and Stopping Docker Containers

docker start -i <container_id>

- ◇ This command will start an existing stopped container in interactive (-i) mode (you will get container's STDIN channel)

docker inspect <container_id>

- ◇ This command will provide JSON-encoded information about the running container identified by *container_id*

docker stop <container_id>

- ◇ This command will stop the running container identified by *container_id*
- For the Docker command-line reference, visit

<https://docs.docker.com/engine/reference/commandline/cli/>

1.16 Docker Volume

- If you destroy a container and recreate it, you will lose data
- Ideally, data should not be stored in containers
- Volumes are mounted file systems available to containers
- Docker volumes are a good way of safely storing data outside a container
- Docker volumes can be shared across multiple containers
- Creating a Docker volume

```
docker volume create my-volume
```

- Mounting a volume

```
docker run -v my-volume:/my-mount-path -it ubuntu:12.04  
/bin/bash
```

- Viewing all volumes

```
docker volume ls
```

- Deleting a volume

```
docker volume rm my-volume
```

1.17 Dockerfile

- Rather than manually creating containers and saving them as custom images, it's better to use Dockerfile to build images
- Sample script

```
# let's use ubuntu docker image  
FROM openjdk
```

```
RUN apt-get update -y  
RUN apt-get install sqlite -y
```

```
# deploy the jar file to the container  
COPY SimpleGreeting-1.0-SNAPSHOT.jar  
/root/SimpleGreeting-1.0-SNAPSHOT.jar
```

- The Dockerfile filename is case sensitive. The 'D' in Dockerfile has to be uppercase.
- Building an image using docker build. (Mind the space and period at the end of the docker build command)

```
docker build -t my-image:v1.0 .
```

- Or, if you want to use a different file name:

```
docker build -t my-image:v1.0 -f mydockerfile.txt
```

1.18 Docker Compose

- A container runs a single application. However, most modern application rely on multiple service, such as database, monitoring, logging, messages queues, etc.
- Managing a forest of containers individually is difficult
 - ◇ Especially when it comes to moving the environment from development to test to production, etc.
- Compose is a tool for defining and running multi-container Docker applications on the same host
- A single configuration file, docker-compose.yml, is used to define a group of container that must be managed as a single entity

1.19 Using Docker Compose

- Define as many Dockerfile as necessary
- Create a docker-compose.yml file that refers to the individual Dockerfile
- Sample Dockerfile

```
version: '3'
services:
  greeting:
    build: .
    ports:
      - "8080:8080"
    links:
```

```
- mongodb
mongodb:
  image: mongodb
  environment:
    MONGO_INITDB_ROOT_USERNAME: wasadmin
    MONGO_INITDB_ROOT_PASSWORD: secret
  volumes:
    - my-volume:/data/db
volumes:
  my-volume: {}
```

1.20 Dissecting docker-compose.yml

- The Docker Compose file should be named either docker-compose.yml or docker-compose.yaml
 - ◇ Using any other names will require to use the -f argument to specify the filename
- The docker-compose.yml file is writing in YAML
 - ◇ <https://yaml.org/>
- The first line, version, indicates the version of Docker Compose being used
 - ◇ As of this writing, version 3 is the latest

1.21 Specifying services

- A 'service' in docker-compose parlance is a container
- Services are specified under the service: node of the configuration file
- You choose the name of a service. The name of the service is meaningful within the configuration
- A service (container) can be specified in one of two ways: Dockerfile or image name
- Use build: to specify the path to a Dockerfile
- Use image: to specify the name of an image that is accessible to the host

1.22 Dependencies between containers

- Some services may need to be brought up before other services
- In docker-compose.yml, it is possible to specify which service relies on which using the links: node
- If service C requires that service A and B be brought up first, add a link as follows:

```
A:
  build: ./servicea
```

```
B:
  build: ./serviceb
```

```
C:
  build: ./servicec
  link:
  - A
  - B
```

- It is possible to specify as many links as necessary
 - ◇ Circular links are not permitted (A links to B and B links to A)

1.23 Injecting Environment Variables

- In a microservice, containerized application, environment variables are often used to pass configuration to an application
- It is possible to pass environment variable to a service via the docker-compose.yml file

```
myservice:
  environment:
    MONGO_INITDB_ROOT_USERNAME: wasadmin
    MONGO_INITDB_ROOT_PASSWORD: secret
```

1.24 runC Overview

- Over the last few years, Linux has gradually gained a collection of features.
- Windows 10 and Windows Server 2016+, also added similar features.

- Those individual features have esoteric names like “control groups”, “namespaces”, “seccomp”, “capabilities”, “apparmor” and so on.
- Collectively, they are known as “OS containers” or sometimes “lightweight virtualization”.
- Docker makes heavy use of these features and has become famous for it.
- Because “containers” are actually an array of complicated, sometimes arcane system features, they are integrated into a unified low-level component called runC.
- runC now available as a standalone tool which is a lightweight, portable container runtime. It includes all of the plumbing code used by Docker to interact with system features related to containers. It is designed with the following principles in mind:
- It has no dependency on the rest of the Docker platform

1.25 runC Features

- Full support for Linux namespaces, including user namespaces
- Native support for all security features available in Linux: Selinux, Apparmor, seccomp, control groups, capability drop, pivot_root, uid/gid dropping, etc. If Linux can do it, runC can do it.
- Native support for live migration, with the help of the CRIU team at Parallels
- Native support of Windows 10 containers is being contributed directly by Microsoft engineers
- Planned native support for Arm, Power, Sparc with direct participation and support from Arm, Intel, Qualcomm, IBM, and the entire hardware manufacturers ecosystem.
- Planned native support for bleeding edge hardware features – DPDK, sr-iov, tpm, secure enclave, etc.

1.26 Using runC

- In order to use runc you must have your container in the format of an

Open Container Initiative (OCI) bundle.

- If you have Docker installed you can use its export method to acquire a root filesystem from an existing Docker container.

```
# create the topmost bundle directory
mkdir /mycontainer
cd /mycontainer
```

```
# create the rootfs directory
mkdir rootfs
```

```
# export busybox via Docker into the rootfs directory
docker export $(docker create busybox) | tar -C rootfs -xvf -
```

- After a root filesystem is populated you just generate a spec in the format of a config.json file inside your bundle.

```
runc spec
```

1.27 Running a Container using runc

- The first way is to use the convenience command run that will handle creating, starting, and deleting the container after it exits.

```
# run as root
cd /mycontainer
runc run mycontainerid
```

- The second way is to implement the entire lifecycle (create, start, connect, and delete the container), manually

```
# run as root
cd /mycontainer
runc create mycontainerid
```

```
# view the container is created and in the "created" state
runc list
```

```
# start the process inside the container
runc start mycontainerid
```

```
# after 5 seconds view that the container has exited and is now in the stopped state
runc list
```

```
# now delete the container
runc delete mycontainerid
```

1.28 Summary

- Docker is a system for creating virtual environments which are, for all intents and purposes, lightweight virtual machines
- Docker containers can only run the type of OS that matches the host's OS
- Docker containers are extremely lightweight (although not so robust and secure), allowing you to achieve a higher level of deployed application density compared with traditional VMs (10x more units!)
- On-demand provisioning of applications by Docker supports the Platform-as-a-Service (PaaS)–style deployment and scaling
- runC is a container runtime which has support for various containerization solutions

Chapter 2 - Introduction to Kubernetes

Objectives

Key objectives of this chapter

- What is Kubernetes?
- What Is a Container?
- Microservices and Orchestration
- Microservices and Infrastructure-as-Code
- Kubernetes Container Networking

2.1 What is Kubernetes

- Kubernetes is Greek for "helmsman" or "pilot"
- Originally founded by Joe Beda, Brendan Burns and Craig McLuckie
- Afterward, other Google engineers also joined the project
- The original codename of Kubernetes within Google was Project Seven, a reference to a Star Trek character. The seven spokes on the wheel of the Kubernetes logo is a nod to that codename
- Kubernetes is commonly referred to as **K8s**
- An open-source system for automating deployment, scaling, and management of containerized applications
- Originally designed by Google and donated to the Cloud Native Computing Foundation
- Provides a platform for automating deployment, scaling, and operations of application containers across clusters of hosts
- Supports a range of container tools, including Docker

2.2 What is a Container

- Over the past few years, containers have grown in popularity
- Containers provide operating-system-level virtualization

- It is a computer virtualization method in which the kernel of an operating system allows the existence of multiple isolated user-space instances, instead of just one
- Such instances are called containers
- A container is a software bucket comprising everything necessary to run the software independently.
- There can be multiple containers in a single machine and containers are completely isolated from one another as well as from the host machine.
- Containers are also called virtualization engines (VEs) or Jails (e.g. FreeBSD jail)
- Containers look like real computers from the point of view of programs running in them
- Items usually bundled into a container include:
 - ◇ Application, dependencies, libraries, binaries, and configuration files

2.3 Container – Uses

- OS-level virtualization is commonly used in virtual hosting environments
- A container is useful for packaging, shipping, and deployment of any software applications that are presented as lightweight, portable, and self-sufficient containers, that will run virtually anywhere.
- It is useful for securely allocating finite hardware amongst a large number of mutually-distributing users
- System administrators may also use it for consolidating server hardware by moving services on separate hosts into containers on a single server
- Container is useful for packaging everything the app needs into a container and migrating that from one VM to another, to server or cloud without having to refactor the app.
- Container usually imposes little to no overhead, because programs in virtual partitions use the OS' normal system call interface and do not need to be subjected to emulation or be run in an intermediate virtual machines
- Container doesn't require support in hardware to perform efficiently

2.4 Container – Pros

- Containers are fast compared to hardware-level virtualization, since there is no need to boot up a full virtual machine. A Container allows you to start apps in a virtual, software-defined environment much more quickly
- The average container size is within the range of tens of MB while VMs can take up several gigabytes. Therefore a server can host significantly more containers than virtual machines
- Running containers is less resource intensive than running VMs so you can add more computing workload onto the same servers
- Provisioning containers only take a few seconds or less, therefore, the data center can react quickly to a spike in user activity.
- Containers can enable you to easily allocate resources to processes and to run your application in various environments.
- Using containers can decrease the time needed for development, testing, and deployment of applications and services.
- Testing and bug tracking also become less complicated since you there is no difference between running your application locally, on a test server, or in production.
- Containers are a very cost effective solution. They can potentially help you to decrease your operating cost (less servers, less staff) and your development cost (develop for one consistent runtime environment).
- Using containers, developers are able to have truly portable deployments. This helps in making Continuous Integration / Continuous Deployment easier.
- Container-based virtualization are a great option for microservices, DevOps, and continuous deployment.

2.5 Container – Cons

- Compared to traditional virtual machines, containers are less secure.
 - ◇ Containers share the kernel, other components of the host operating system, and they have root access.

- ◊ This means that containers are less isolated from each other than virtual machines, and if there is a vulnerability in the kernel it can jeopardize the security of the other containers as well.
- A container offers less flexibility in operating systems. You need to start a new server to be able to run containers with different operating systems.
- Networking can be challenging with containers. Deploying containers in a sufficiently isolated way while maintaining an adequate network connection can be tricky.
- Developing and testing for containers requires training. Whereas writing applications for VMs, which are in effect the same as physical machines, is a straightforward transition for development teams.
- Single VMs often run multiple applications. Whereas containers promotes a one-container one-application infrastructure. This means containerization tends to lead to a higher volume of discreet units to be monitored and managed.

2.6 Composition of a Container

- At the core of container technology are
 - ◊ Control Groups (cgroups)
 - ◊ Namespaces
 - ◊ Union filesystems

2.7 Control Groups

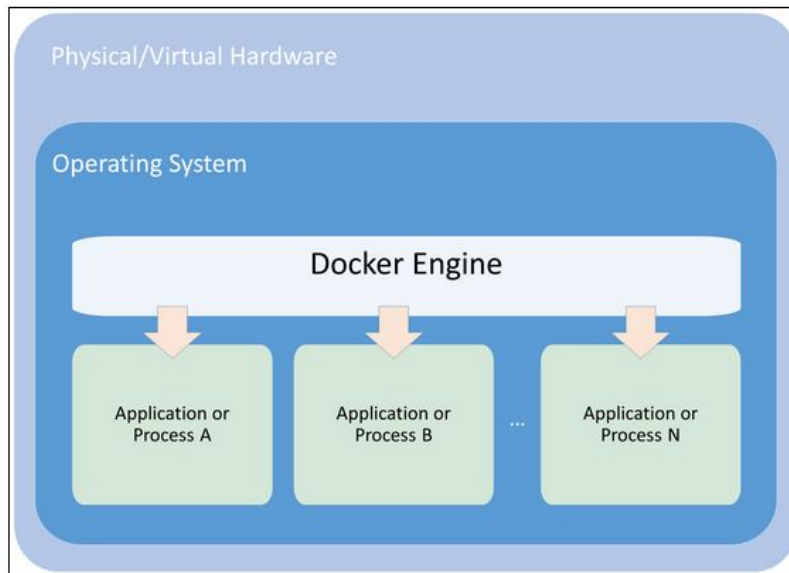
- Control groups (cgroups) work by allowing the host to share and also limit the resources each process or container can consume
- This is important for both, resource utilization and security
- It prevents denial-of-service attacks on host's hardware resources

2.8 Namespaces

- Namespaces offer another form of isolation for process interaction within

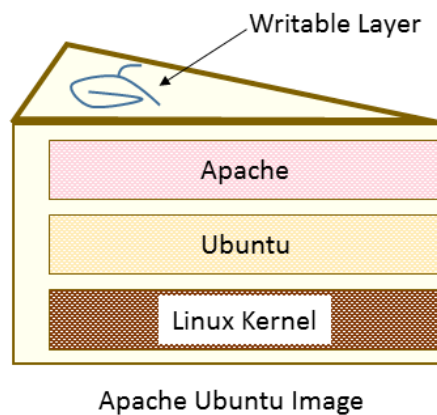
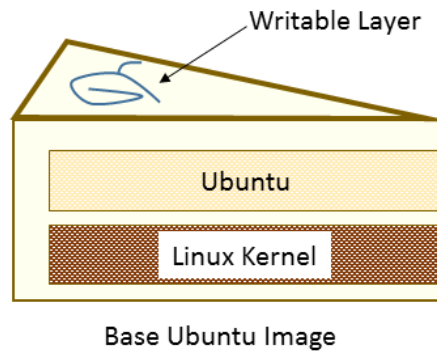
operating systems.

- It limits the visibility a process has on other processes, networking, filesystems, and user ID components
- Container processes are limited to see only what is in the same namespace
- Processes from containers or the host processes are not directly accessible from within the container process.



2.9 Union Filesystems

- Containers run from an image, much like an image in the VM or Cloud world, it represents state at a particular point in time.
- Container images snapshot the filesystems
- The snapshot tend to be much smaller than a VM
- The container shares the host kernel and generally runs a much smaller set of processes
- The filesystem is often layered or multi-leveled.
 - ◇ e.g. Base layer can be Ubuntu with an application such as Apache or MySQL stacked on top of it



2.10 Popular Containerization Software

- Docker
 - ◇ Docker Swarm
- Packer
- Kubernetes
- Rocket (rkt)
- Apache Mesos
- Linux Containers (LXC)
- CloudSang
- Marathon
- Nomad

- Fleet
- Rancher
- Containership
- OpenVZ
- Oracle Solaris Containers
- Tectonic

2.11 Microservices

- The microservice architectural style is an approach to developing a single application as a suite of small services
- Each service runs in its own process and communicates with lightweight mechanisms, often an HTTP resource API
- These services are built around business capabilities and independently deployable by fully automated deployment machinery

2.12 Microservices and Containers / Clusters

- Containers are excellent for microservices, as it isolates the services.
- Containerization of single services makes it easier to manage and update these services
- Docker has led to the emergence of frameworks for managing complex scenarios, such as:
 - ◇ how to manage single services in a cluster
 - ◇ how to manage multiple instances in a service across hosts
 - ◇ how to coordinate between multiple services on a deployment and management level
- Kubernetes allows easy deployment and management of multiple Docker containers of the same type through an intelligent tagging system
- With Kubernetes, you describe the characteristics of the image, e.g. number of instances, CPU, RAM, you would like to deploy

2.13 Microservices and Orchestration

- Microservices can benefit from deployment to containers
- Issue with containers is, they are isolated. Microservices might require communication with each other
- Container orchestration can be used to handle this issue
- Container orchestration refers to the automated arrangement, coordination, and management of software containers
- Container orchestration also helps in tackling challenges, such as
 - ◇ service discovery
 - ◇ load balancing
 - ◇ secrets/configuration/storage management
 - ◇ health checks, auto-[scaling/restart/healing] of containers and nodes
 - ◇ zero-downtime deploys

2.14 Microservices and Infrastructure-as-Code

- In the old days, you would write a service and allow the Operations (Ops) team to deploy it to various servers for testing, and eventually production.
- Infrastructure-as-Code solutions helps in shortening the development cycles by automating the set up of infrastructure
- Popular infrastructure-as-code solutions include Puppet, Chef, Ansible, Terraform, and Serverless.
- In the old days, servers were treated as part of the family.
 - ◇ Servers were named, constantly monitored, and carefully updated
- Due to containers and infrastructure-as-code solutions, these days the servers (containers) are often not updated. Instead, they are destroyed, then recreated.
- Containers and infrastructure-as-code solutions treat infrastructure as disposable.

2.15 Kubernetes Container Networking

- Microservices require a reliable way to find and communicate with each other.
- Microservices in containers and clusters can make things more complex as we now have multiple networking namespaces to bear in mind.
- Communication and discovery requires traversing of container IP space and host networking.
- Kubernetes benefits from getting its ancestry from the clustering tools used by Google for the past decade. Many of the lessons learned from running and networking two billion containers per week have been distilled into Kubernetes

2.16 Kubernetes Networking Options

- Docker creates three types of networks by default:
 - ◇ bridged – this is the default choice. In this mode, the container has its own networking namespace and is then bridged via virtual interfaces to the host network. In this mode, two containers can use the same IP range because they are completely isolated
 - ◇ host – in this mode, performance is greatly benefited since it removes a level of network virtualization; however, you lose the security of having an isolated network namespace
 - ◇ none – creates a container with no external interface. Only a loopback device is shown if you inspect the network interfaces
 - ◇ host, and none
 - ◇ In all these scenarios, we are still on a single machine, and outside of a host mode, the container IP space is not available, outside the machine. Connecting containers across two machines then requires NAT and port mapping for communication
- Docker user-defined networks
 - ◇ Docker also supports user-defined networks via network plugins
 - bridge driver – allows creation of networks somewhat similar to

default bridge driver

- overlay driver – uses a distributed key-value store to synchronize the network creation across multiple hosts
- Macvlan driver – uses the interface and sub-interfaces on the host. It offers a more efficient network virtualization and isolation as it bypasses the Linux bridge
- Weave
 - ◇ Provides an overlay network for Docker containers
- Flannel
 - ◇ Gives a full subnet to each host/node enabling a similar pattern to the Kubernetes practice of a routable IP per pod or group of containers
- Project Calico
 - ◇ Uses built-in routing functions of the Linux kernel.
 - ◇ It can be used for anything from small-scale deploys to large Internet-scale installations.
 - ◇ There is no need for additional NAT, tunneling, or overlays.
- Canal
 - ◇ It merges both Calico for network policy and Flannel for overlay into one solution

2.17 Kubernetes Networking – Balanced Design

- Using unique IP address at the host level is problematic as the number of containers grow.
 - ◇ Assigning an IP address to each container can also be overkill.
- In cases of sizable scale, overlay networks and NATs are needed in order to address each container.
 - ◇ Overlay networks add latency
- You have to pick between
 - ◇ fewer containers with multiple applications per container (unique IP

address for each container)

- ◇ multiple containers with fewer applications per container (Overlay networks / NAT)

2.18 Summary

- Kubernetes provides a platform for automating deployment, scaling, and operations of application containers across clusters of hosts
- Kubernetes supports a range of container tools, including Docker
- Containers are useful for packaging, shipping, and deployment of any software applications that are presented as lightweight, portable, and self-sufficient containers, that will run virtually anywhere.
- Microservices can benefit from containers and clustering.
- Kubernetes offers container orchestration

Chapter 3 - Kubernetes – From the Firehose

Objectives

Key objectives of this chapter

- Masters
- Nodes
- Pods
- Namespaces
- Resource Quota
- Authentication and Authorization
- Routing
- Registry
- Storage Volumes

3.1 What is Kubernetes?

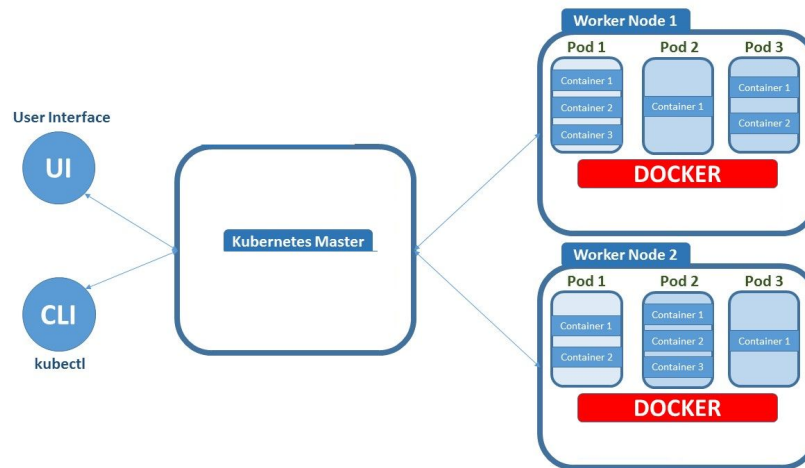
- Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications
- It groups containers that make up an application into logical units for easy management and discovery.
- Designed on the same principles that allows Google to run billions of containers a week, Kubernetes can scale without increasing your ops team.
- Whether testing locally or running a global enterprise, Kubernetes flexibility grows with you to deliver your applications consistently and easily no matter how complex your need is
- Kubernetes is open source giving you the freedom to take advantage of on-premises, hybrid, or public cloud infrastructure, letting you effortlessly move workloads to where it matters to you.
- Kubernetes can be deployed on a bare-metal cluster (real machines) or on a cluster of virtual machines.

3.2 Container Orchestration

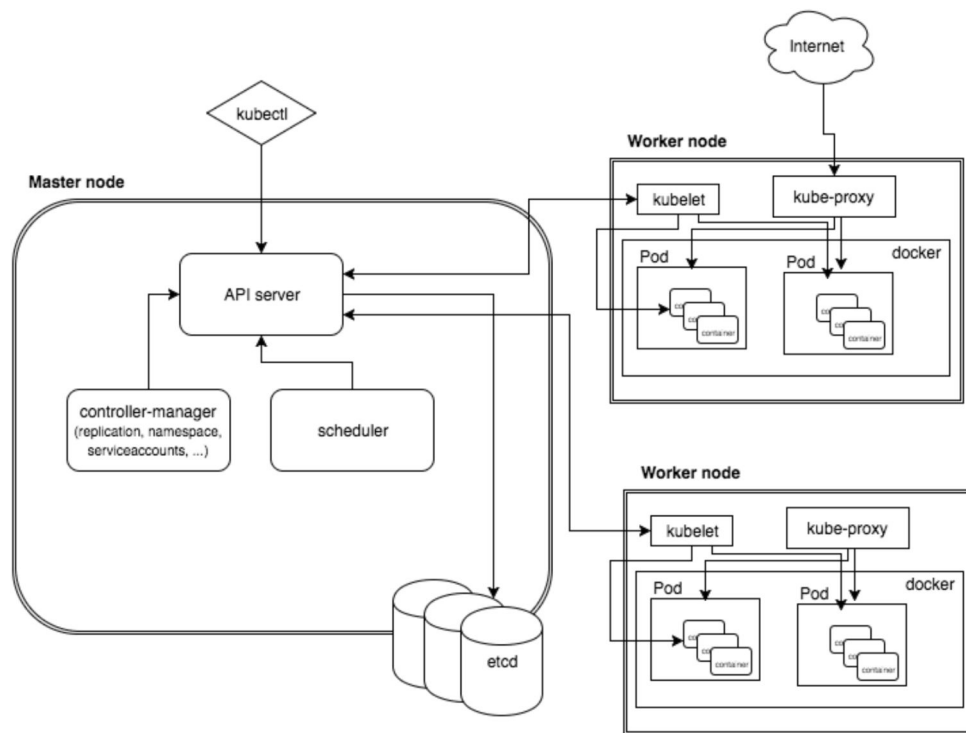
- The primary responsibility of Kubernetes is container orchestration.
- Kubernetes ensures that all the containers that execute various workloads are scheduled to run physical or virtual machines
- The containers must be packed efficiently following the constraints of the deployment environment and the cluster configuration
- Kubernetes keeps an eye on all running containers and replaces dead, unresponsive, or otherwise healthy containers.
- Kubernetes can orchestrate the containers it manages directly on bare-metal or on virtual machines
- A Kubernetes cluster can also be composed of a mix of bare-metal and virtual machines, but this is not very common.
- Containers are ideal to package microservices because, while providing isolation to the microservice, they are very lightweight compared virtual machines. This makes containers ideal for cloud deployment

3.3 Kubernetes Basic Architecture

- At a very high level, the following are the key concepts
 - ◇ Master
 - ◇ Nodes (old term used to be Minions)
 - ◇ Pods
 - ◇ Containers



3.4 Kubernetes Detailed Architecture



source: https://res.cloudinary.com/dukp6c7f7/image/upload/f_auto,fl_lossy,q_auto/s3-ghost/2016/06/o7leok.png

3.5 Kubernetes Concepts

- Cluster

- Namespace
- Master
- Node
- Pod
- Label
- Annotation

3.6 Kubernetes Concepts (contd.)

- Label Selector
- Replication Controller and replica set
- Services
- Persistent Volume
- Secret
- ConfigMap
- Workloads, such as Deployments, StatefulSet, Jobs, and DaemonSet
- Operators
- Custom Resource Definitions (CRDs)

3.7 Cluster and Namespace

- Cluster
 - ◇ A collection of physical resources, such as hosts storage and networking resources
 - ◇ The entire system may consist of multiple clusters
- Namespace
 - ◇ It is a virtual cluster
 - ◇ A single physical cluster can contain multiple virtual clusters segregated by namespaces

- ◇ Virtual clusters can communicate through public interfaces
- ◇ Pods can live in a namespace, but nodes can not.
- ◇ Kubernetes can schedule pods from different namespaces to run on the same node

3.8 Node

- A single host
- It may be a physical or virtual machine
- Its job is to run pods
- Each node runs several Kubernetes components, such as a kubelet and a kube proxy
- kubelet is a service which reads container manifests as YAML files that describes a pod.
- Nodes are managed by a Kubernetes master
- The nodes are worker bees of Kubernetes and shoulder all the heavy lifting
- In the past they were called minions.

3.9 Master

- The master is the control plane of Kubernetes
- It consists of components, such as
 - ◇ API server
 - ◇ a scheduler
 - ◇ a controller manager
- The master is responsible for the global, cluster-level scheduling of pods and handling events.
- Often, all the master components are set up on a single host
- For implementing high-availability scenarios or very large clusters, you will

want to have master redundancy.

3.10 Pod

- A pod is the unit of work on Kubernetes
- Each pod contains one or more containers
- Pods provide a solution for managing groups of closely related containers that depend on each other and need to cooperate on the same host
- Pods are considered throwaway entities that can be discarded and replaced at will (i.e. they are cattle, not pets)
- Each pod gets a unique ID (UID)
- All the containers in a pod have the same IP address and port space
- The containers within a pod can communicate using localhost or standard inter-process communication
- The containers within a pod have access to shared local storage on the node hosting the pod and is mounted on each container
- The benefit of grouping related containers within a pod, as opposed to having one container with multiple applications, are:
 - ◇ Transparency – making the containers within the pod visible to the infrastructure enables the infrastructure to provide services to those containers, such as process management and resource monitoring
 - ◇ Decoupling software dependencies – the individual containers maybe be versioned, rebuilt, and redeployed independently
 - ◇ Ease of use – users don't need to run their own process managers
 - ◇ Efficiency – because the infrastructure takes on more responsibility, containers can be more lightweight

3.11 Label

- Labels are key-value pairs that are used to group together sets of objects, often pods.

- Labels are important for several other concepts, such as replication controller, replica sets, and services that need to identify the members of the group
- Each pod can have multiple labels, and each label may be assigned to different pods.
- Each label on a pod must have a unique key
- The label key must adhere to a strict syntax
 - ◇ Label has two parts: prefix and name
 - ◇ Prefix is optional. If it exists then it is separated from the name by a forward slash (/) and it must be a valid DNS sub-domain. The prefix must be 253 characters long at most
 - ◇ Name is mandatory and must be 63 characters long at most. Name must begin with an alphanumeric character and contain only alphanumeric characters, dots, dashes, and underscores. You can create another object with the same name as the deleted object, but the UIDs must be unique across the lifetime of the cluster. UIDs are generated by Kubernetes
 - ◇ Values follow the same restrictions as names

3.12 Annotation

- Unlike labels, annotation can be used to associate arbitrary metadata with Kubernetes objects.
- Kubernetes stores the annotations and makes their metadata available
- Unlike labels, annotations don't have strict restrictions about allowed characters and size limits

3.13 Label Selector

- They are used to select objects based on their labels
- A value can be assigned to a key name using equality-based selectors, (=, ==, !=).

- ◇ e.g.
 - role = webserver
 - role = dbserver, application != sales
- in operator can be used as a set-based selector
 - ◇ .e.g
 - role in (dbserver, backend, webserver)

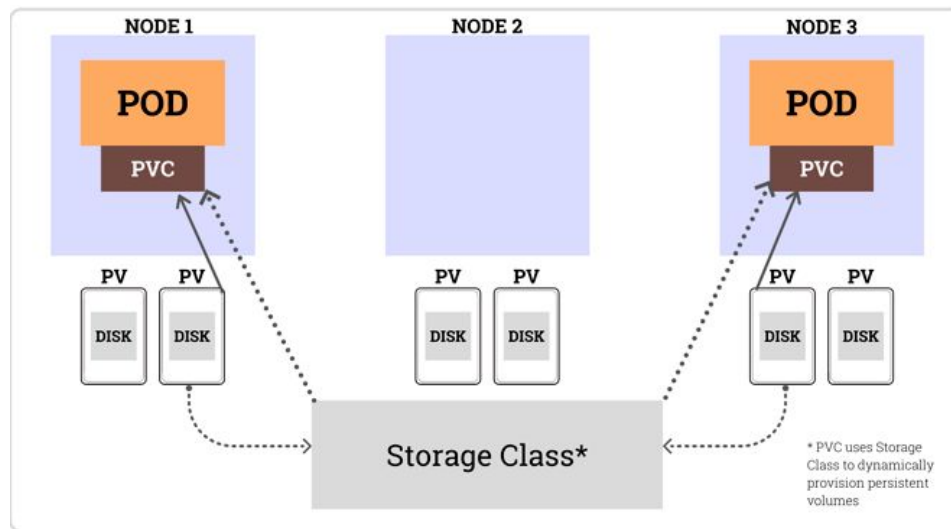
3.14 Replication Controller and Replica Set

- They both manage a group of pods identified by a label selector
- They ensure that a certain number of pods are always up and running
- Whenever the number drops due to a problem with the hosting node or the pod itself, Kubernetes fires up new instances
- If you manually start pods and exceed the specified number, the replication controller kills some extra pods
- Replication controllers test for membership by name equality, whereas replica sets can use set-based selection
- Replica sets are newer and considered as next-gen replication controllers

3.15 Service

- Services are used to expose some functionality to users or other services
- They usually involve a group of pods, usually identified by a label
- Kubernetes services are exposed through endpoints (TCP/UDP)
- Services are published or discovered via DNS, or environment variables
- Services can be load-balanced by Kubernetes

3.16 Persistent Volumes & Storage Class



source: <https://blog.mayadata.io/kubernetes-storage-basics-pv-pvc-and-storageclass>

3.17 Persistent Volumes & Storage Class

- When a pod is destroyed, the data used by the pod is also destroyed.
- If you want the data to outlive the pod or share data between pods, volume concept can be utilized.
- Kubernetes allows access to physical storage devices, such as SSDs, NVMe disks, NAS, and NFS servers via **Persistent Volumes (PV)**.
- A Kubernetes Pod can consume a Persistent Volume by using a **PersistentVolumeClaim (PVC)** object.
- To create the PV/PVC pair for your Pod, you use a **StorageClass** object.
- With a StorageClass, you do not need to create a persistent volume separately before claiming it.

3.18 Secret

- Secrets are small objects that contain sensitive info, such as credentials
- They are stored as plain-text but can be encrypted
- They can be mounted as files into pods

- The same secret can be mounted into multiple pods
- Internally, Kubernetes creates secrets for its components, and you can create your own secrets

3.19 ConfigMaps

- A ConfigMap is an API object used to store non-confidential data in key-value pairs.
- Pods can consume ConfigMaps as environment variables, command-line arguments, or as configuration files in a volume.
- ConfigMap does not provide encryption.
- Use a Secret rather than a ConfigMap to keep your data private.

3.20 Custom Resource Definitions (CRDs)

- Custom Resource allows you to extend Kubernetes by adding a custom resource.
- A resource is an endpoint in k8s API that allow you to store an API object of any kind.
- A custom resource allows you to create your own API objects and define your own kind just like Pod, Deployment, ReplicaSet, etc.
- Custom Resource Definition is what you use to define a Custom Resource.
- This is a powerful way to extend Kubernetes capabilities beyond the default installation.
- CRDs are YAML-based files that can be applied by using the `kubectl` command.

```
kubectl apply -f {my-crd.yaml}
```

3.21 Operators

- A Kubernetes operator is a software extension to Kubernetes and provides a method of packaging, deploying, and managing a Kubernetes

application.

- A Kubernetes operator is an application-specific controller that extends the functionality of the Kubernetes API to create, configure, and manage instances of complex applications on behalf of a Kubernetes user.
- In Kubernetes, controllers implement control loops that repeatedly compare the desired state of the cluster to its actual state. If the cluster's actual state doesn't match the desired state, then the controller takes action to fix the problem.
- Operator development often starts with automating the application's installation and self-service provisioning, and follows with more complex automation capabilities.
- There is also a Kubernetes operator software development kit (SDK) that can help you develop your own operator. The SDK provides the tools to build, test, and package operators with a choice of creating operators using Helm charts, Ansible Playbooks or Golang.

3.22 Resource Quota

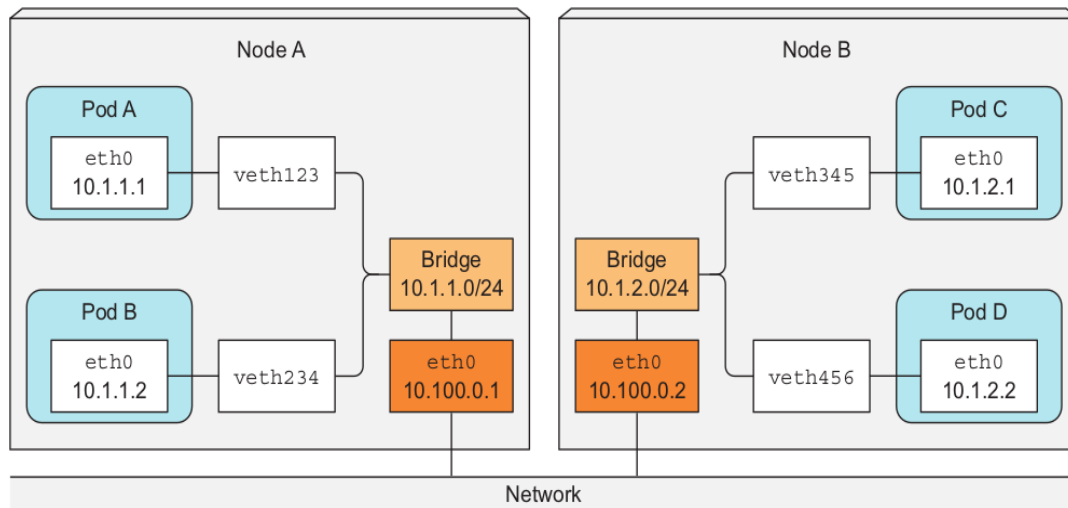
- Kubernetes allows management of different types of quota
- Compute resource quota
 - ◇ Compute resources are CPU and memory
 - ◇ You can specify a limit or request a certain amount
 - ◇ Uses fields, such as requests.cpu, requests. memory
- Storage resource quota
 - ◇ You can specify the amount of storage and the number of persistent volumes
 - ◇ Uses fields, such as requests.storage, persistentvolumeclaims
- Object count quota
 - ◇ You can control API objects, such as replication controllers, pods, services, and secrets
 - ◇ You can not limit API objects, such as replica sets and namespaces.

3.23 Authentication and Authorization

- Permission rules can be added to the Kubernetes system for more advanced management
- Applying authentication and authorization is a secure solution to prevent your data being accessed by others.
- Authentication is currently supported in the form of tokens, passwords, and certificates.
- Authorization supports three modes:
 - ◇ RBAC (Role-Based Access Control)
 - ◇ ABAC (Attribute-Based Access Control) – lets a user define privileges via attributes in a file
 - ◇ Webhook – allows for integration with third-party authorization via REST web service calls.

3.24 Routing

- Routing connects separate networks
- Routing is based on routing tables
- Routing table instructs network devices how to forward packets to their destination
- Routing is done through various network devices, such as routers, bridges, gateways, switches, and firewalls



source: <https://i.stack.imgur.com/Cwd7c.png>

3.25 Registry

- Container images aren't very useful if it's only available on a single machine
- Kubernetes relies on the fact that images described in a Pod manifest are available across every machine in the cluster
- Container images can be stored in a remote registry so every machine in the cluster can utilize the images.
- Registry can be public or private.
- Public registries allow anyone to download images (e.g. Docker Hub), while private registries require authentication to download images (e.g. Docker Registry)
- Docker Registry is a stateless, highly scalable-server side application that stores and lets you distribute Docker images.
- Docker Registry is open-source
- Docker Registry gives you following benefits
 - ◇ tight control where your images are being stored
 - ◇ fully own your images distribution pipeline
 - ◇ enterprises often use Jfrog's artifactory, Sonatype's Nexus, CNCF's

Harbor, or Redhat's Quay repositories as image repositories.

3.26 Using Docker Registry

- Default storage location is

```
/var/lib/registry
```

- Change storage location by creating an environment variable like this

```
REGISTRY_STORAGE_FILESYSTEM_ROOTDIRECTORY=/somewhere
```

- Start your registry (:2 is the version. Check Docker Hub for latest version)

```
docker run -d -p 5000:5000 -name registry registry:2.6
```

- Pull some image from the hub

```
docker pull ubuntu
```

- Tag the image so that it points to your registry

```
docker tag ubuntu localhost:5000/myfirstimage
```

- Push it

```
docker push localhost:5000/myfirstimage
```

- Pull it back

```
docker pull localhost:5000/myfirstimage
```

- Stop your registry and remove all data

```
docker stop registry && docker rm -v registry
```

3.27 Summary

- Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications
- The primary responsibility of Kubernetes is container orchestration.
- At a high level, Kubernetes involves Pods, Master, and Nodes
- Kubernetes also involves labels, annotations, replication controllers, replica sets, and secrets
- Container images can be deployed to a public or private registry

Chapter 4 - Kubernetes and the Twelve Factors

Objectives

Key objectives of this chapter

- Kubernetes and the Twelve Factors

4.1 Kubernetes and the Twelve Factors - 1 Codebase

- Kubernetes makes heavy use of declarative constructs.
- All parts of a Kubernetes application are described with text-based representations in YAML or JSON.
- The referenced containers are themselves described in source code as a Dockerfile
- Because everything from the image to the container deployment behavior is encapsulated in text, you are able to easily source control all the things, typically using git.

4.2 Kubernetes and the Twelve Factors - 2 Dependencies

- A microservice is only as reliable as its most unreliable dependency.
- Kubernetes includes *readinessProbes* and *livenessProbes* that enable you to do ongoing dependency checking.
- The *readinessProbe* allows you to validate whether you have backing services that are healthy and you're able to accept requests.
- The *livenessProbe* allows you to confirm that your microservice is healthy on its own.
- If either probe fails over a given window of time and threshold attempts, the Pod will be restarted.

4.3 Kubernetes and the Twelve Factors - 3 Config

- The Config factor requires storing configuration sources in your process environment table (e.g. ENV VARs).

- Kubernetes provides *ConfigMaps* and *Secrets* that can be managed in source repositories
 - ◇ *Secrets* should never be source controlled without an additional layer of encryption
- Containers can retrieve the config details at runtime.

4.4 Kubernetes and the Twelve Factors - 4 Backing Services

- When you have network dependencies, we treat that dependency as a “Backing Service”.
- At any time, a backing service could be attached or detached and our microservice must be able to respond appropriately.
- For example, you have an application that interacts with a web server, you should isolate all interaction to that web server with some connection details (either dynamic service discovery or via *Config* in a Kubernetes *Secret*). Then consider whether your network requests implement fault tolerance such that if the backing service fails at runtime, your microservice does not trigger a cascading failure. That service may also be running in a separate container or somewhere off-cluster. Your microservice should not care as all interactions then occur through APIs to interact with the database.

4.5 Kubernetes and the Twelve Factors - 5 Build, Release, Run

- Once you commit the code, a build occurs and the container image is built and published to an image registry.
- If you’re using Helm, your Kubernetes application may also be packaged and published into a Helm registry as well.
- These “releases” are then re-used and deployed across multiple environments to ensure that an unexpected change is not introduced somewhere in the process (by re-building the binary or image for each environment).

4.6 Kubernetes and the Twelve Factors - 6 Processes

- In Kubernetes, a container image runs as a container process within a Pod.

- Kubernetes (and containers in general) provide a facade to provide better isolation of the container process from other containers running on the same host.
- Using a process model enables easier management for scaling and failure recover (e.g. restarts).
- Typically, the process should be stateless to support scaling the workload out through replication.
- For any state used by the application, you should use a persistent data store that all instances of your application process will discover via your Config.
- In Kubernetes-based applications where multiple copies of pods are running, requests can go to any pod, hence the microservice cannot assume sticky sessions.

4.7 Kubernetes and the Twelve Factors - 7 Port Binding

- You can use Kubernetes Service objects to declare the network endpoints of your microservices and to resolve the network endpoints of other services in the cluster or off-cluster.
- Without containers, whenever you deployed a new service (or new version), you would have to perform some amount of collision avoidance for ports that are already in use on each host.
- Container isolation allows you to run every process (including multiple versions of the same microservice) on the same port (by using network namespaces in the Linux kernel) on a single host.

4.8 Kubernetes and the Twelve Factors - 8 Concurrency

- Kubernetes allows you to scale the stateless application at runtime with various kinds of lifecycle controllers.
- The desired number of replicas are defined in the declarative model and can be changed at runtime.
- Kubernetes defines many lifecycle controllers for concurrency including *ReplicationControllers*, *ReplicaSets*, *Deployments*, *StatefulSets*, *Jobs*, and

DaemonSets.

- Kubernetes supports autoscaling based on compute resource thresholds around CPU and memory or other external metrics.
- The *Horizontal Pod Autoscaler (HPA)* allows you to automatically scale the number of pods within a Deployment or *ReplicaSet*.

4.9 Kubernetes and the Twelve Factors - 9 Disposability

- Within Kubernetes, you focus on the simple unit of deployment of Pods which can be created and destroyed as needed—no single Pod is all that valuable.
- When you achieve disposability, you can start up fast and the microservices can die at any time with no impact on user experience.
- With the livenessProbes and readinessProbes, Kubernetes will actually destroy Pods that are not healthy over a given window of time.

4.10 Kubernetes and the Twelve Factors - 10 Dev/Prod Parity

- Containers (and to a large extent Kubernetes) standardize how you deliver your application and its running dependencies, meaning that you're able to deploy everything the same way everywhere.
- For example, if you're using MySQL in a highly available configuration in production, you can deploy the same architecture of MySQL in your dev cluster.
- By establishing parity of production architectures in earlier dev environments, you can typically avoid unforeseen differences that are important to how the application runs (or more importantly how it fails).

4.11 Kubernetes and the Twelve Factors - 11 Logs

- For containers, you will typically write all logs to stdout and stderr file descriptors.
- The important design point is that a container should not attempt to manage internal files for log output, but instead delegate to the container

orchestration system around it to collect logs and handle analysis and archival.

- Often in Kubernetes, you'll configure Log collection as one of the common services to manage Kubernetes.
- For example, you can enable an Elasticsearch-Logstash-Kibana (ELK) stack within the cluster.

4.12 Kubernetes and the Twelve Factors - 12 Admin Processes

- Within Kubernetes, the Job controller allows you to create Pods that are run once or on a schedule to perform various activities.
- A Job might implement business logic, but because Kubernetes mounts API tokens into the Pod, you can also use them for interacting with the Kubernetes orchestrator as well.
- By isolating these kinds of administrative tasks, you can further simplify the behavior of your microservice.

Chapter 5 - Leading Practices for Microservice Logging

Objectives

Key objectives of this chapter

- The Challenges of Logging in a Service Oriented Architecture
- A list of leading practices recommended for handling Microservice logging
- A discussion of those practices

5.1 Logging Challenges

- (Micro-)Service Architectures mean that solutions are distributed across a network topology, with many small processes involved in handling a single request.
- Simple logging solutions generally don't scale beyond a single process.
- How do we manage logs, which contain forensic information, when the requests have been distributed to many, potentially load-balanced, processes?

5.2 Leading Practices

- Correlate Requests with a Unique ID
- Include a Unique ID in the Response
- Send Logs to a Central Location
- Structure Your Log Data
- Add Context to Every Request
- Write Logs to Local Storage

Notes

We refer to these as leading practices because the notion of “best” is a value judgment left to the implementer. These are generally considered the right practices with which to guide, or lead, adopters.

Ref: <https://dzone.com/articles/microservices-logging-best-practices>

5.3 Correlate Requests with a Unique ID

- Requests will flow through various instances of our microservices.
- By attaching, propagating, and using a unique ID for each request, we can identify the log records associated with handling that request.
- Since each request represents a call between two services, when logging requests, we could use a compound key consisting of a unique ID for the main (originating) request, as well as source and target IDs of each service. That way we would be able to find all entries related to the main request, as well as all requests related to a specific source or target service.

5.4 Include a Unique ID in the Response

- When providing a response, whether a fault or a regular response, provide a Unique ID.
 - ◇ The ID is typically the one used to correlate the request.
- In the event that you need more detail about a response, the Unique ID can be used to extract information from the logging system.
 - ◇ Imagine extracting the log records for a particular loan application to investigate how the system determined its response.

5.5 Send Logs to a Central Location

- Microservice solutions are distributed across a network topology. If we do not collect log information in a central location, then when we have the need to analyze the logs, it will be extremely challenging to locate and gather all of the log records from wherever they may be in the network.
- To make matters even worse, if you log to local storage within a container, and that container is terminated, the log data may be lost entirely.
- For these reasons, centralized logging is now considered to be the norm, and a “solved problem” with well-known, packaged solutions.

5.6 Structure Your Log Data

- Microservice solutions tend to be extremely heterogeneous.
 - ◇ There may be no agreement on the content of the log records. Different services need different fields, and there is no point in having a bloat of unnecessary information.
 - ◇ Different technology stacks may use different raw logging formats and different mechanisms.
- Put your log information into a flexible, extensible, format such as JSON or XML.
 - ◇ Depending on your technology stack, you might do this natively, or via a logging sidecar service.

Notes

You may already be familiar with the notion of a sidecar, but if not, it will be discussed a bit later in this unit.

5.7 Add Context to Every Record

- The purpose for logging is to provide a record suitable for analysis and action.
- There are different audiences for the log.
 - ◇ Operations staff need complete, understandable, actionable, information on which to act in the event of issues. This allows for the timely remediation.
 - ◇ Developers may need more detailed information with which to debug. This level of detail may require knowledge of the source code in order to interpret.
 - ◇ Tools – we may be able to automate actions in response to certain log information, allowing seamless and rapid response without manual intervention.
- When logging, try to provide all of the context that any party may need in the future.

- ◇ In general, include whatever information you believe would be, or later discover would have been, useful to resolving an issue.
- ◇ Constantly evaluate the contents of your log records. After all, we choose to use a flexible and extensible format for reasons, including not being locked into a single set of content and allowing each audience segment to extract the log information relevant to it.

Notes

We refer to these as leading practices because the notion of “best” is a value judgment left to the implementer. These are generally considered the right practices with which to guide, or lead, adopters.

Ref: <https://dzone.com/articles/microservices-logging-best-practices>

5.8 Examples of Content

- Examples of useful log content include:
 - ◇ Timestamp – every log record should have a timestamp, and your network should be time synchronized
 - ◇ Exception trace – log records related to exceptions should include their detailed stack traces
 - ◇ The service name and location – each log record should identify the service instance that generated it.
 - ◇ A code locator – indicates from where in the code the log record originated.
 - ◇ API level information – the operation, parameters, return values, status code, etc. involved.
 - ◇ Network addresses – source and target IP addresses
 - ◇ Client information – identify the app and/or user-agent that generated the request.
- Again, evaluate log content based on your use. What information are you logging that is useful? What are you are not logging that would have been useful? What information might help automate handling?


Notes

We refer to these as leading practices because the notion of “best” is a value judgment left to the implementer. These are generally considered the right practices with which to guide, or lead, adopters.

Ref: <https://dzone.com/articles/microservices-logging-best-practices>

5.9 Write Logs to Local Storage

- Centralized logs are vital, but writing to them directly would be ill-advised. If you think of centralized logs as a service, you realize that writing to them is subject to the same issues as service to service communications.
- One approach is to write your logs locally, then use a sidecar service to transfer the local logs to the central logging service.
 - ◇ The sidecar can take care of transforming and transferring the local logs to the central logs in a timely, but efficient, manner. It also helps to decouple stack specific log formats from the canonical log format chosen for the central logging service.

 When writing logs to local storage, be mindful of how logs are rotated and retained so they don't fill up the disk

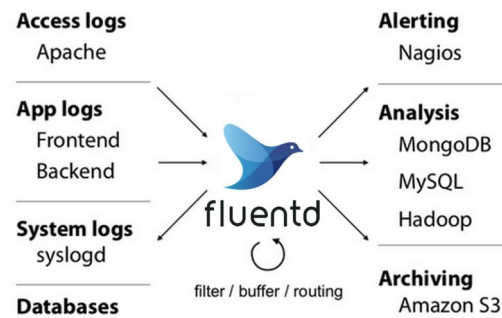
Notes

We refer to these as leading practices because the notion of “best” is a value judgment left to the implementer. These are generally considered the right practices with which to guide, or lead, adopters.

Ref: <https://dzone.com/articles/microservices-logging-best-practices>

5.10 Collecting Logs with Fluentd

- Fluentd is an open source data collector, which lets you unify the data collection and consumption for a better use and understanding of data.



- It's key features are:
 - ◇ Unified Logging with JSON
 - ◇ Pluggable Architecture (plugins both for sources and data outputs)
 - ◇ Minimum Resources Required (The vanilla instance runs on 30-40MB of memory and can process 13,000 events/second/core. If you have tighter memory requirements (-450kb)
 - ◇ Built-in Reliability (memory- and file-based buffering)

5.11 Leading Practices for Microservice Logging Summary

- We considered some of the challenges related to logging when your solutions are distributed and heterogeneous.
- We looked at a list of practices that can help you solve the logging challenges
- We discussed those practices.

5.12 Metrics Using Prometheus

- Overview
- Prometheus
- Service Discovery
- Exposing Metrics in Services
- Querying in Prometheus
- Grafana

- and others...

5.13 Overview

- The idea of maintaining performance metrics is an old and well-understood concept.
 - ◇ For Java, consider JSR 174: Monitoring and Management Specification
- The challenge in modern service-oriented architectures is that our solutions are fine-grained and distributed across network topologies. Metrics have to be gathered, correlated and organized.

Notes

JSR 174: <https://jcp.org/en/jsr/detail?id=174>

5.14 Prometheus

- Prometheus is an Open Source tool, originally developed by SoundCloud, for collecting, storing and analyzing metrics in a network environment.
- Prometheus gather metrics from many sources throughout your network, stores them as time-series data, supports queries, and generates alerts.

5.15 Prometheus

- Major features provided by Prometheus include:
 - A time-series database
 - A purpose-built query language
 - A web UI
 - Support for third party tools, e.g.
 - ◇ Grafana – a high-end, Open Source, solution for analyzing and visualizing time-series data
 - ◇ Tools, such as prophet, forecast-prometheus or prometheus-anomaly-detector, which use various techniques to attempt to predict future

failures based on patterns of change in metrics

Notes

Prometheus stores metrics in a time-series data model, q.v., https://en.wikipedia.org/wiki/Time_series

5.16 Prometheus

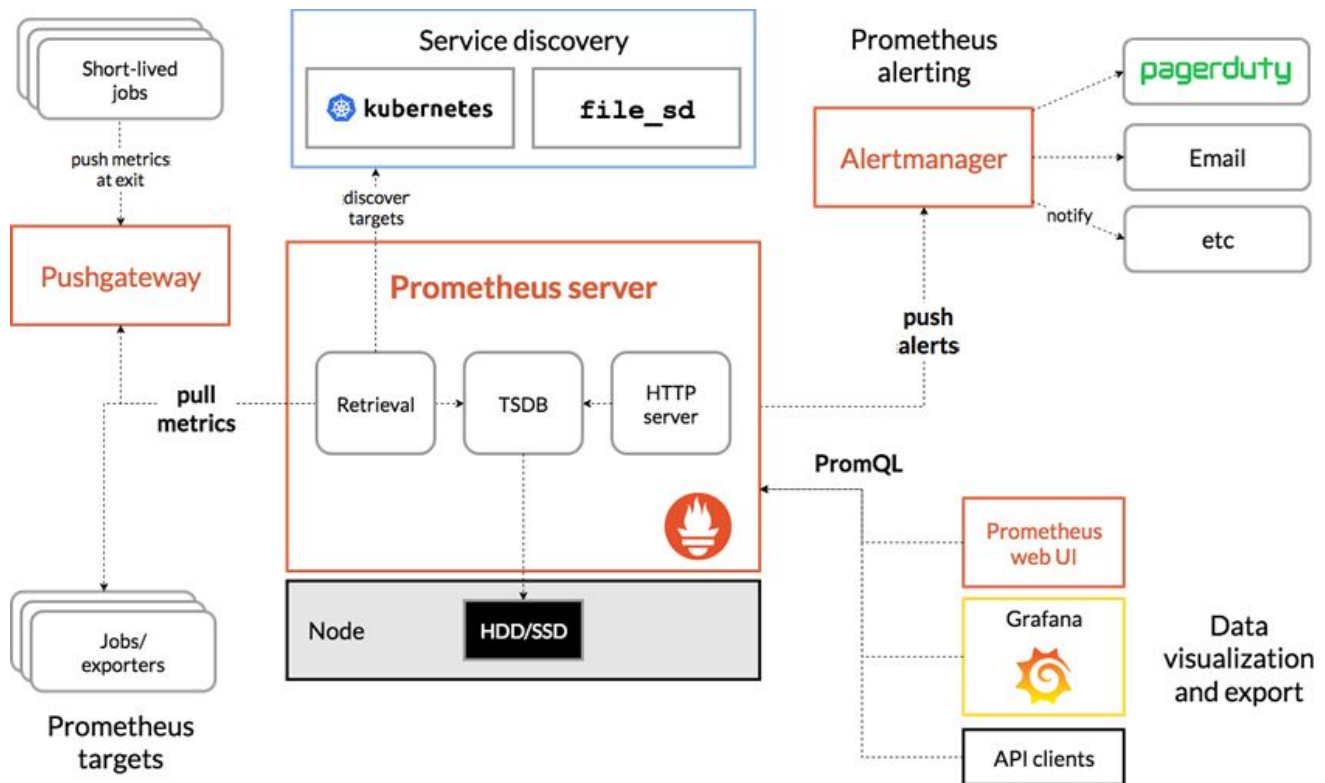
- Major components that make up Prometheus include:
 - ◇ A central server collecting metrics and hosting the time-series metrics database
 - ◇ Client libraries for instrumenting applications to expose metrics
 - ◇ Application/technology specific exporters that expose the metrics of existing, third-party, systems into Prometheus without having to specifically instrument those systems
 - ◇ An Alertmanager, which can process alerts synthesized from out-of-compliance metrics
 - ◇ A push gateway to support gathering metrics from short-lived processes

Notes

Prometheus pulls metrics from targets.. The purpose for the push gateway is that short lived processes won't be around for Prometheus to poll, so instead those processes are instrumented to push (send) their data to the gateway, which acts as a cache from which Prometheus can pull the metrics, even after the instrumented process has terminated.

You can instrument applications using the client libraries in order to expose metrics. There are also exporters which can expose metrics for existing applications without manually instrumenting them. For example, there is an exporter for Java that will export all JMX metrics to Prometheus.

5.17 Prometheus Architecture



5.18 Service Discovery

- In order to poll targets, Prometheus first needs to find them. This is referred to as Service Discovery (SD).
- Prometheus provides a pluggable approach to Service Discovery. Amongst the options are:
 - ◇ File Discovery (file_sd)
 - ◇ Kubernetes
 - ◇ Consul
 - ◇ Azure
 - ◇ EC2
 - ◇ DNS, and more.
- Prometheus recommends that if none of the existing solutions work for

you, that you use the file-based discovery service, and use a tool to write the necessary JSON content.

Notes

Prometheus' authors have gone so far as to declare a moratorium on adding any new Service Discovery implementations, and are promoting the file-based service discovery.

“There is currently a moratorium on new service discovery mechanisms being added to Prometheus due to a lack of developer capacity. In the meantime `file_sd` remains available.” --

<https://github.com/prometheus/prometheus/tree/master/discovery>

“If you need to use a service discovery system that is not currently supported, your use case may be best served by Prometheus' file-based service discovery mechanism, which enables you to list scrape targets in a JSON file (along with metadata about those targets).” --

<https://prometheus.io/docs/guides/file-sd/>

<https://prometheus.io/docs/prometheus/latest/configuration/configuration/>

5.19 File-based Service Discovery

- On the Prometheus server, configure `prometheus.yml` to define the scrape configuration
- In the scrape configuration, define the `file_sd` configuration. That configuration will include references to JSON files
- The JSON files will define the targets to be scraped.

Notes

A sample prometheus YAML file describing that we are going to poll metrics for a job (type) called “node”:

```
scrape_configs:  
- job_name: 'node'
```

file_sd_configs:

- files:
- 'targets.json'

That configuration tells us to poll that job using file-based service discovery using a file called “targets.json”

A sample targets.json:

```
[
  {
    "labels": {
      "job": "node"
    },
    "targets": [
      "localhost:9100"
    ]
  }
]
```

That file is what Prometheus recommends you populate using a tool. For example, in a Spring Cloud environment, one could write a tool to populate a JSON file based on information retrieved from Eureka.

5.20 Istio and Prometheus

- Istio ships with support for Prometheus
 - ◇ “Mixer comes with a built-in Prometheus adapter that exposes an endpoint serving generated metric values.”
 - ◇ “The Prometheus add-on is a Prometheus server that comes preconfigured to scrape Mixer endpoints to collect the exposed metrics.”

- Istio exposes the following metrics targets: Mixer-generated metrics, Mixer's own metrics, Envoy-generated metrics, Pilot-generated metrics, Gallery-generated metrics, and Policy-related metrics
- If necessary, other metrics targets in an Istio/Kubernetes environment could be discovered using the Kubernetes SD plugin that ships with Prometheus. However, in many cases this is unnecessary, because ...
- Istio is used to implement the Service Mesh Pattern. Therefore, Istio is in a position to generate many metrics for you. You don't need to instrument your services in order to gather metrics related to their service traffic; all of that can be done for you by Istio.

Notes

You can read details on Metrics in Istio at <https://istio.io/docs/tasks/telemetry/metrics/>.

The section Querying Metrics from Prometheus (<https://istio.io/docs/tasks/telemetry/metrics/querying-metrics>) discusses the pre-configured nature of the Prometheus add-on for Istio.

The section Collecting Metrics (<https://istio.io/docs/tasks/telemetry/metrics/collecting-metrics/>) discusses and demonstrates how to get Istio to gather telemetry metrics for your services. The section Collecting Metrics for TCP services (<https://istio.io/docs/tasks/telemetry/metrics/tcp-metrics/>) discusses and demonstrates how to automatically gather telemetry for TCP services. These two differ only in the set of available attributes available to use when defining the metrics gathering.

Should you have the need, Configuring Prometheus to use Kubernetes SD (https://prometheus.io/docs/prometheus/latest/configuration/configuration/#kubernetes_sd_config) covers configuring Prometheus to use Kubernetes SD.

5.21 Exposing Metrics in Services

- Prometheus metrics are defined by their type, value, and name
- Prometheus names metrics in a specific manner, using a name and a set of labels:
- `<metric name>{<name>=<value>, ...}`

- ◇ An example would be:

```
requests_total{service="flights", server="pod54", ...}
```

Notes

The structure of Prometheus metric names lets you define a single, common, name for a metric, such as `heap_used`, `heap_free`, `cpu_usage`, `requests_per_second`, and use the labels (metadata) to filter events later. Each label defines a dimension on which you can query.

5.22 Exposing Metrics in Services

- Prometheus understands four (4) types of metrics, which is stored in its time-series database.
 - ◇ Counter: monotonically increasing number
 - ◇ Gauge: a number that can increase or decrease
 - ◇ Histogram: a set of buckets tracking the number of observations that fall into each bucket
 - ◇ Summary: similar to a histogram, but supports quantiles

Notes

Prometheus Metric Types (https://prometheus.io/docs/concepts/metric_types)

A counter is a monotonically incrementing value. It starts at 0, and can only be reset on a restart. Examples of counters would include the number of requests, a running total of bytes, an error count, cars rented, or similar.

A gauge is also a numeric value, but it can increase and decrease. The current occupancy of a resource pool, RAM or CPU use, etc., would be examples of a gauge.

Think of a histogram as you might a bar chart. Each bar (a bucket in histogram terms) represents a classification of a sampled observation, and the value represents the number of such observations. So, for example, you could have a histogram showing the distribution of response sizes. The histogram also tracks the total sum of observation values and the total number of observations.

A summary is very similar to a histogram, but also maintains quantiles over a sliding time window. Quantile are cut points dividing a range of values into sub-ranges of equal size. If quintiles are defined on the range of 0 to 1, the 0.5-quantile would be the median, and the 0.95-quantile would be the 95th percentile.

5.23 Exposing Metrics in Services

- A Prometheus metric is written in plain text.
- The format of an individual metric is:
- Metric name {labels} value optional-timestamp
- To build on our earlier example, a metric might be:

```
response_time{method="post", service="flight",  
server="pod54", url="/flights"} 13 1560020895000
```

Notes

A metric (https://prometheus.io/docs/concepts/data_model) of the form##

```
http_response_time{method="post", service="flight", server="pod54", handler="/flights"}  
13 1560020895000
```

would be understood as the `http_response_time` metric, specifically related to a POST to the flight service on server `pod54` to the resource handler for the `/flights` URL with a value of 13(ms), with a UNIX timestamp (optional, and in millisecond precision).

Prometheus has a simple set of recommendations for metric names and labels (<https://prometheus.io/docs/practices/naming/>)

5.24 Exposing Metrics in Services

- As a reminder, depending upon your technology stack, you may not need to do anything to expose your metrics. There are existing components that can expose existing metrics from your stack to Prometheus.
 - ◇ ~200 (and counting) technologies, including hardware, and software

such as Oracle DB, MongoDB, JIRA, Kafka, Kubernetes, and Istio, either have an Exporter or are directly integrated with Prometheus.

- For JVM-based processes, there is a generic JMX Exporter to export all JMX-based metrics.
- For microservices written in Python using Django, Nodejs, or Spring Boot w/Hystrix, there are 3rd party libraries that can integrate those services directly into Prometheus on your behalf.
- For systems and/or metrics for which support isn't already provided, you would provide your own, using the client library for any of the 20 or so languages currently supported.

Notes

For a list of Exporters provided for Prometheus, see Exporters and Integrations (<https://prometheus.io/docs/instrumenting/exporters>). That page also lists software that is directly integrated with Prometheus.

Prometheus officially provides Client Libraries (<https://prometheus.io/docs/instrumenting/clientlibs>) for Java, Python, Go and Ruby, but there are client libraries for over a dozen other languages.

5.25 Exposing Metrics in Services

- You use one of the Prometheus Client Libraries to define and expose metrics in your application.
- A Client Library provides the infrastructure to define metrics, as well as the HTTP infrastructure allowing it to be polled by Prometheus.
- This is a Spring Bean instrumented with a Prometheus metric, using Spring AOP support in the official Java Client:

```
@Controller
public class MyController {
    @RequestMapping("/")
    @PrometheusTimeMethod(
        name = "my_controller_path_duration_seconds",
        help = "Some helpful info here")
    public Object handleMain() {...}
```

- See the student notes for a complete example in Python.

Notes

This sample Python Application copied from the Python Client Library (https://github.com/prometheus/client_python) exposes a simple Summary metric:

```
from prometheus_client import start_http_server, Summary
import random
import time

# Create a metric to track time spent and requests made.
REQUEST_TIME = Summary('request_processing_seconds', 'Time spent processing request')

# Decorate function with metric.
@REQUEST_TIME.time()
def process_request(t):
    """A dummy function that takes some time."""
    time.sleep(t)

if __name__ == '__main__':
    # Start up the server to expose the metrics.
    start_http_server(8000)
    # Generate some requests.
    while True:
        process_request(random.random())
```


5.26 Exposing Metrics in Services

- The official Java Client library provides:
 - ◇ Classes to define any of the four types of Prometheus metrics
 - ◇ Support for Spring Framework allowing us to instrument methods with a Summary metric using Spring AOP and a decorator.
 - ◇ Collectors – these are objects that will collect for you metrics from the JVM, JMX, logging frameworks, Guava cache, Hibernate SessionFactory, and Jetty Server.
 - The client library also provides a framework allowing you to write custom collectors for code that you cannot instrument
- On the prior slide, we gave an example using Spring AOP
- Likewise, there are multiple ways to expose metrics to Prometheus. The client library includes multiple implementations, including:
 - ◇ HTTPServer – embedded HTTP server
 - ◇ MetricsServlet – a servlet to use with a Java Web Container
 - ◇ PrometheusMvcEndpoint – an extension for the Spring Boot Actuator
 - ◇ MetricsHandler – a Vert.x handler to expose metrics

Notes

The official Java Client Library (https://github.com/prometheus/client_java) provides many ways to define metrics and expose them.

This article, <https://www.callicoder.com/spring-boot-actuator-metrics-monitoring-dashboard-prometheus-grafana>, goes into more detail on the Actuator endpoint provided for Prometheus.

For details on Spring Boot actuator, refer to the article at <https://www.baeldung.com/spring-boot-actuators>

5.27 Exposing Metrics in Services

- Normally, Prometheus polls your service to read its metrics. But what if your process is short-lived?
- As mentioned earlier, and shown on the Architecture slide, Prometheus provides a Push gateway that acts as a cache. Short-lived processes can proactively publish their metrics to the gateway, and Prometheus will pull the metrics from the gateway.
- The Java Client Library provides a PushGateway class so that short-lived processes can push their metrics to the Push gateway.

Notes

See https://github.com/prometheus/client_java#exporting-to-a-pushgateway for an example in the official documentation.

5.28 Querying in Prometheus

- Prometheus provides a bespoke language, PromQL, for querying the time-series database. The elements of PromQL include:
 - ◇ Data types
 - instant vector – a set of time series values all sharing the same timestamp
 - range vector – a set of time series containing a range of data points over time
 - scalar – a single, floating-point, value
 - string – a string value
 - ◇ Time Series Selectors
 - Instant Time Series Selectors
 - Range Vector Time Series Selectors
 - Offset Modifiers
 - ◇ Subqueries – runs an instant query for a given range and resolution, resulting in a range vector.

- ◇ Operators – many logical and function operators, such as comparison and math.
- ◇ Functions – built-in functions that can operate on time series data

Notes

Querying Prometheus (<https://prometheus.io/docs/prometheus/latest/querying/basics/>) is a primer on the Prometheus Query Language PromQL.

5.29 Querying in Prometheus

- Prometheus queries can be:
 - ◇ Simple time series selection – just the name of a metric, or a metric with a set of labels and values to match.
 - ◇ Subquery – a query containing a subquery, such as querying the rate based on totals over a period of time.
 - ◇ Queries that use PromQL functions and/or operators – an example of this might be a query that fetched a max heap size and the average used heap size over the past 30 minutes, and then reported the average amount of free heap in that period.
- Queries can match string values using regular expressions.

Notes

Querying Prometheus (<https://prometheus.io/docs/prometheus/latest/querying/basics/>) is a primer on the Prometheus Query Language PromQL.

5.30 Querying in Prometheus

- Examples:
 - ◇ Retrieve `http_requests_total` metrics#`http_requests_total`
 - ◇ Retrieve specific `http_requests_total` metrics over a 5 minute range#`http_requests_total{method="GET", handler="/flight"}[5m]`

- ◇ Return the 5-minute rate of the `http_requests_total` metric for the past 30 minutes, with a resolution of 1 minute `#rate(http_requests_total[5m])[30m:1m]`
- ◇ Compute the per-second rate for all time series with the `http_requests_total` metric name, as measured over the last 5 minutes `#rate(http_requests_total[5m])`
- ◇ Compute the per-second rate for all time series with the `http_requests_total` metric name, as measured over the last 5 minutes, then reduce to a single value per job `name:#sum(rate(http_requests_total[5m])) by (job)`

Notes

Querying Prometheus (<https://prometheus.io/docs/prometheus/latest/querying/basics/>) is a primer on the Prometheus Query Language PromQL.

5.31 Grafana

- Grafana can lay claim to being the leading Open Source solution for time series analytics.
- Grafana allows you to bring data from various sources, including Elasticsearch and Prometheus, and visualize them with beautiful graphs.
 - ◇ Yes, Prometheus ships with its own GUI, but Grafana is vastly more capable, and can also handle multiple sources of data, of which Prometheus metrics would be just one.
- Grafana also lets you set alert rules based on the monitored data. When an alert rule fires, it can notify you over multiple channels.
- Grafana dashboards can be quickly designed and shared.
 - ◇ Some of the most popular dashboards in the Grafana community are for Prometheus.

Notes

This article, <https://www.callicoder.com/spring-boot-actuator-metrics-monitoring-dashboard->

prometheus-grafana, goes into more detail on the Actuator endpoint provided for Prometheus, and then uses Grafana to visualize the data.

5.32 Grafana

- Grafana can be broken down into a few functional areas
 - ◇ Visualization – Grafana uses panel plugins to provide many different ways to visualize metrics
 - ◇ Unify Data Sources – Grafana uses a plug-in architecture to query data sources. Out of the box, over 30 different source types are supported.
 - ◇ Alerting – Grafana allows you to write rules about metrics. Those rules are then evaluated
 - ◇ Notification – Grafana sends notifications from alerts, supporting many transports, such as Slack, PagerDuty, e-mail, and more.
 - ◇ Share – 100s of predefined dashboards and plug-ins are available through Grafana's library.

5.33 Grafana

- Grafana has out of the box support for Prometheus.
 - ◇ Prometheus is supported as one of the data source types
 - ◇ Grafana provides a PromQL query editor with metric name lookup
 - ◇ Predefined, templated, dashboards for Prometheus
 - ◇ The ability to use aliases for time series names, allowing shorter display names
- In addition, not only can Grafana visualize Prometheus metrics, Grafana can expose its own metrics to Prometheus.
- The next few slides show sample Grafana dashboards that are visualizing Prometheus metrics.

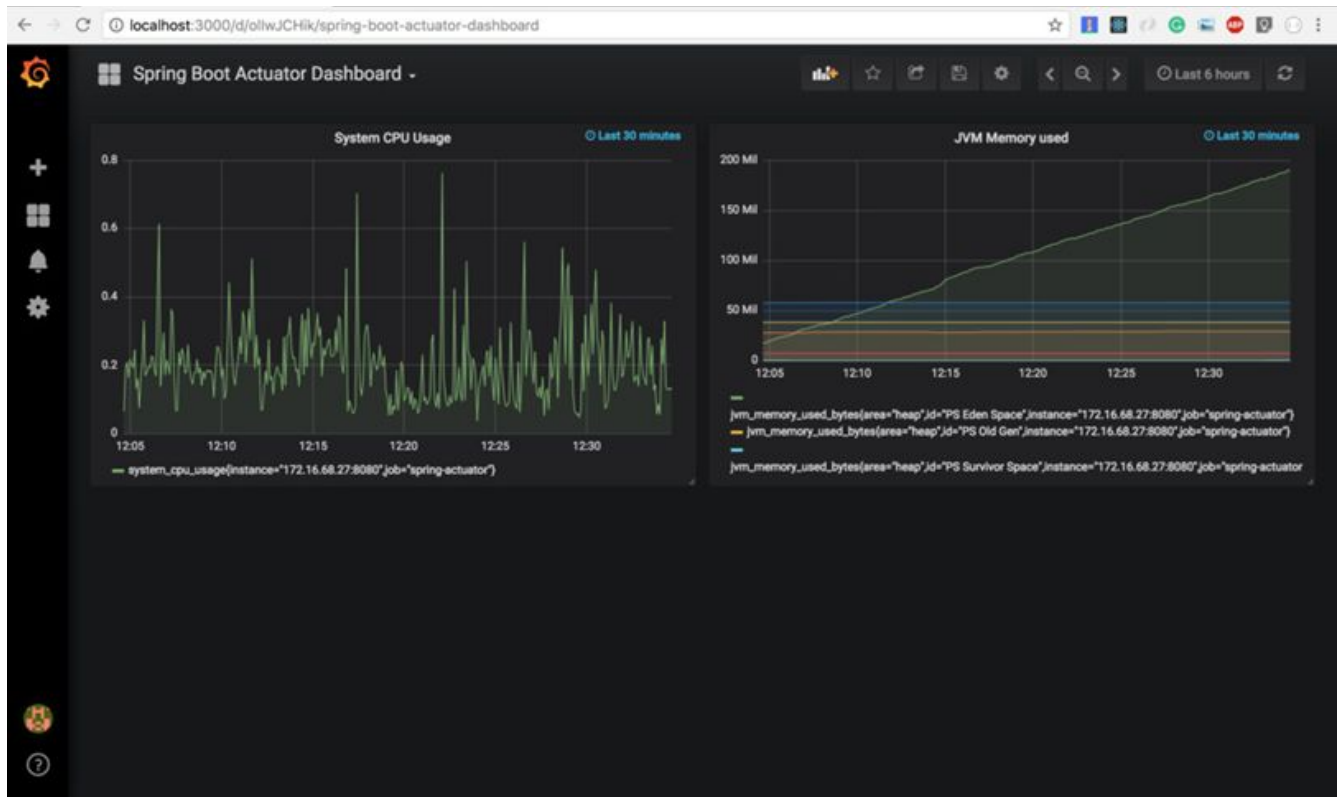
5.34 Grafana

- A sample Grafana Dashboard for Prometheus



5.35 Grafana

- A sample Grafana Dashboard for Prometheus



5.36 Grafana

- A sample Grafana Dashboard for Prometheus



This sample comes from Prometheus' documentation for Grafana (<https://prometheus.io/docs/visualization/grafana/>)

5.37 Business Metrics

- Developers often focus on metrics that relate primarily to process performance
 - ◇ Example: ops / sec, resource (CPU, RAM, pools, etc.) use
- Business Analysts care about metrics that correspond to business goals. These are commonly known as Key Performance Indicators (KPI).

- ◇ For example, the average amount of damage to rental vehicles
- ◇ KPIs are often synthesized from time-series data, rather than being recorded as a single metric. Synthesis may happen long after the metrics were recorded, as previously unrecognized relationships are discovered by Business Analysts.
- Coverage of this topic is currently out of scope for this unit, but the student notes include useful supplemental material.

Notes

A short presentation on the topic of visualizing KPIs with Prometheus and Grafana:

<https://www.slideshare.net/vasster/business-metrics-visualization-with-grafana-and-prometheus>

A much longer talk, From Technical Metrics to Business Observability:

<https://www.slideshare.net/roidelapluie/prometheus-from-technical-metrics-to-business-observability>

5.38 Metrics Using Prometheus Summary

- Overview
- Prometheus
- Service Discovery
- Exposing Metrics in Services
- Querying in Prometheus
- Grafana

5.39 Tracing Using Jaeger

- OpenTracing and its Fundamental Concepts: span and trace
- Jaeger – a Distributed Tracing System, from Über
- Jaeger Client Libraries
- Agent
- Collector

- Query
- Ingester

5.40 OpenTracing

- Tracing is a process of logging the path through code that a request takes. Tracing can help pinpoint failures and performance issues.
- In the case of a distributed architecture, such as a microservices architecture, this takes on both greater import and greater challenge.
 - ◇ Our “applications” are really a network of inter-connected microservices
 - ◇ When we have many such intertwined microservices working together, it’s difficult to map their inter-dependencies, and to understand the execution of an individual request.
 - ◇ Logging does not provide a complete solution.
 - For example, logging alone does not tell us which service was the first in the requests’ flow, nor the sequence taken through the solution topology by the request.
- The solution to this problem is to use a Distributed Tracing technology.

5.41 OpenTracing

- Various vendors had implemented distributed tracing in their own product stacks, but because a microservices solution may be built on many different technology stacks, a unified, vendor and technology neutral, solution is desired.
- The OpenTracing project provides API, frameworks and libraries that allow developers to instrument their code without vendor lock-in.
 - ◇ Many of the concepts and vocabulary come from Google’s Dapper project.
 - ◇ OpenTracing is vendor-neutral. Jaeger is one implementation of OpenTracing.

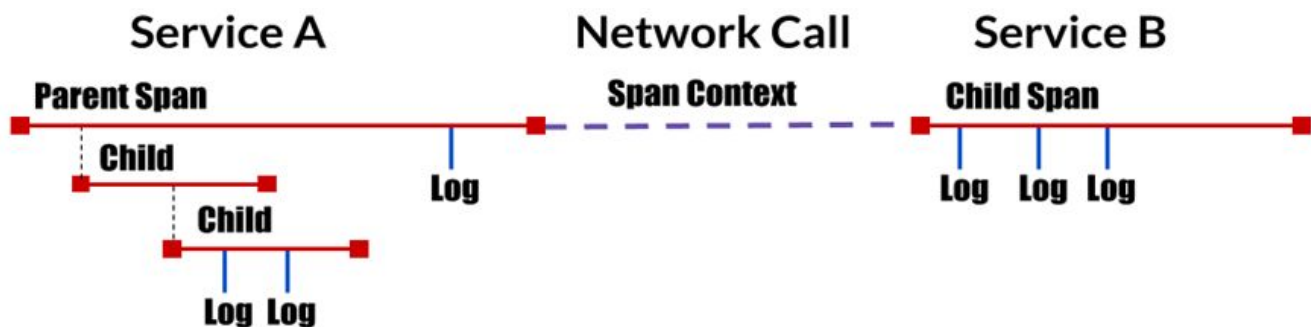
Notes

OpenTracing Project home page: <https://opentracing.io/>

Google Dapper white paper: <https://ai.google/research/pubs/pub36356>

5.42 OpenTracing

- Vocabulary and Data Model
 - ◇ Trace – the description of a transaction as it moves throughout the system
 - ◇ Span – a named, timed, operation within the overall flow. Each span can have metadata key-value pairs as well as fine-grained, timestamped, logs
 - ◇ Span Context – flows with the transaction containing metadata, such as identifiers so that information can be correlated



OpenTracing Specifications: <https://github.com/opentracing/specification>

5.43 OpenTracing

- The primary concept is the span. A span is a named, timed operation contains the information relating to a single, logical, unit of work.
- As per the specification, a span consists of
 - ◇ An operation name
 - ◇ A start timestamp and finish timestamp
 - ◇ A set of key:value span Tags

- key:value pairs that enable user-defined annotation of spans in order to query, filter, and comprehend trace data.
- ◇ A set of key:value span Logs
 - key:value pairs that are useful for capturing span-specific logging messages and other debugging or informational output from the application itself
- ◇ A SpanContext, which is propagated throughout the distributed trace
 - implementation-dependent identifiers for the span and trace
 - “Baggage Items” – key-value pairs containing information to be propagated

Notes

OpenTracing Specification (<https://opentracing.io/specification/>)

5.44 OpenTracing

- A conceptual span example from the OpenTracing Specification

```
span_time=x
end_time=y
operation: db_query
```

Tags:

```
- db.instance: "jdbc:mysql://127.0.0.1:3306/customers"
- db.statement: "SELECT * FROM mytable WHERE foo='bar';"
```

Logs:

```
- message: "Can't connect to mysql server on
'127.0.0.1' (10061) "
```

SpanContext:

```
- trace_id: "abc123"
- span_id: "xyz789"
- Baggage Items:
  - special_id: "vsid1738"
```

- The student notes reference an article with multiple span examples for

Jaeger

Notes

The Life of a Span (<https://medium.com/jaegertracing/the-life-of-a-span-ee508410200b>) discusses spans in detail, with multiple examples, and sample code, using Jaeger.

One thing to notice is that Jaeger stores the starting time stamp and the operation's duration.

5.45 Jaeger

- Jaeger, originally developed by Über, is a Distributed Tracing System inspired Apache Zipkin, which was, in turn inspired by Google Dapper
 - ◇ In fact, Jaeger provides support for Zipkin instrumented code, so that applications already working with Zipkin can simply switch to a Jaeger back-end

Notes

Apache Zipkin: <https://zipkin.apache.org/>

Jaeger Compatibility with Zipkin: <https://www.jaegertracing.io/docs/1.12/features/#backwards-compatibility-with-zipkin>

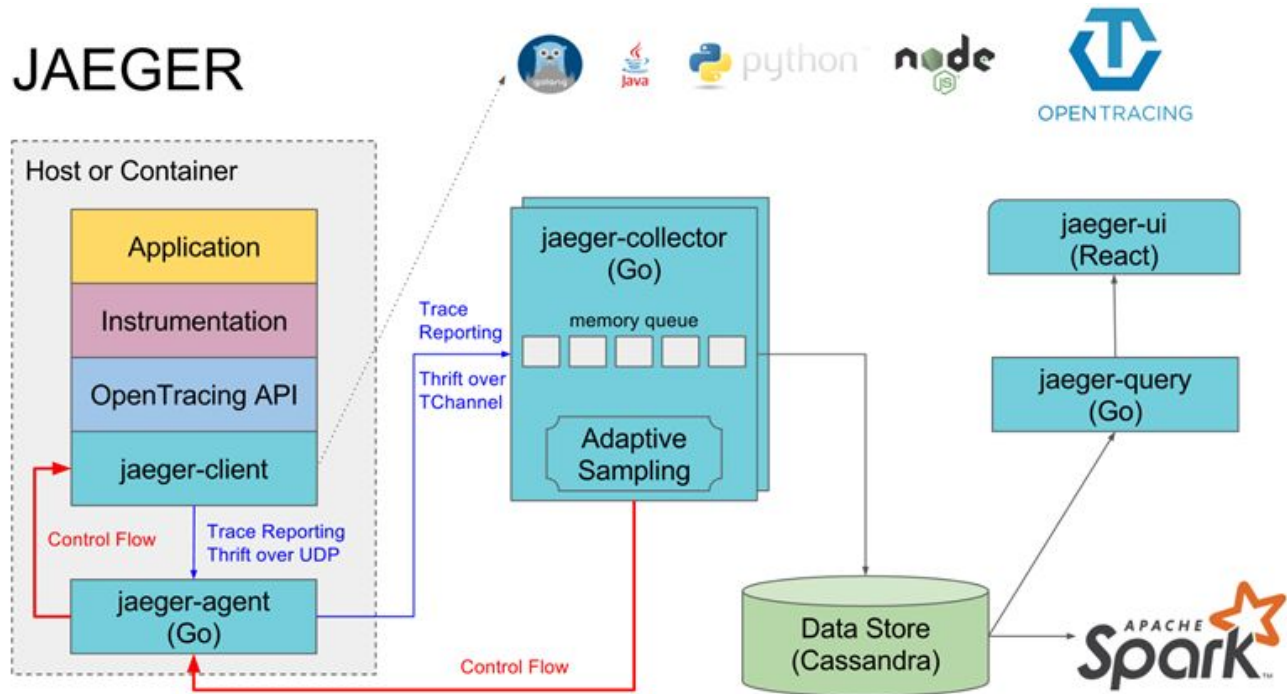
Über published an article explaining how and why Jaeger came about and evolved: <https://eng.uber.com/distributed-tracing/>

5.46 Jaeger

- The major components that make up Jaeger are:
 - ◇ Jaeger Client Libraries
 - ◇ Agent
 - ◇ Collector
 - ◇ Query
 - ◇ Ingester

5.47 Jaeger Architecture Diagram

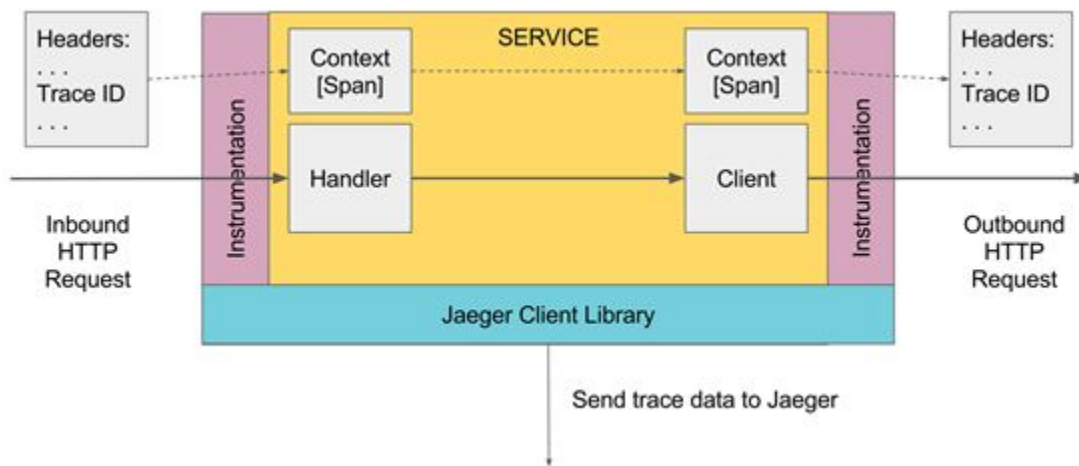
- Jaeger Architecture Diagram



Jaeger Architecture (<https://www.jaegertracing.io/docs/1.12/architecture/>)

5.48 Jaeger Client Libraries

- Jaeger Client Libraries are language specific implementations of the OpenTracing API
- Instrumented code use a client library to generate spans, and propagate the SpanContext when invoking other services



5.49 Jaeger Sampling

- To prevent performance degradation, Jaeger uses the concept of sampling
- Sampled traces are marked for processing and storage
 - ◇ Once a trace is marked for sampling, or not, that decision is propagated
 - ◇ By default, Jaeger samples only 1 trace in 1000
 - ◇ The sampling strategy is pluggable and configurable
 - Istio uses a different strategy. See Student Notes for details
- Spans for sampled traces are emitted from the process

Notes

Sampling: <https://www.jaegertracing.io/docs/1.12/sampling/>

Istio Sampling: <https://istio.io/docs/tasks/telemetry/distributed-tracing/overview/#trace-sampling>

5.50 Jaeger Agent

- The Agent is a network daemon intended to be installed as a sidecar in each container

- The Agent listens to messages (containing spans) sent using UDP from the Client Library
- Spans are locally batched, and batches sent to the Collector

5.51 Jaeger Collector

- The Jaeger Collector service receives messages from Jaeger Agents.
- The messages are placed into a pipeline for processing.
- Messages in the pipeline are validated, indexed, transformed, and stored.
- Collector storage is pluggable. Jaeger currently supports Apache Cassandra, Elasticsearch and Apache Kafka.
 - ◇ Kafka is intended to be used as a transport between Jaeger and actual storage.
 - ◇ The use of Kafka also opens up additional post-processing opportunities.
- The Jaeger Collector optionally provides support for Apache Zipkin.
 - ◇ A Zipkin v1 compatible REST API `/api/v1/spans` accepts Thrift and JSON
 - ◇ A Zipkin v2 compatible REST API `/api/v2/spans` accepts only for JSON

Notes

Collector Zipkin Compatibility: <https://www.jaegertracing.io/docs/1.12/getting-started/#migrating-from-zipkin>

5.52 Query and Ingestor Services

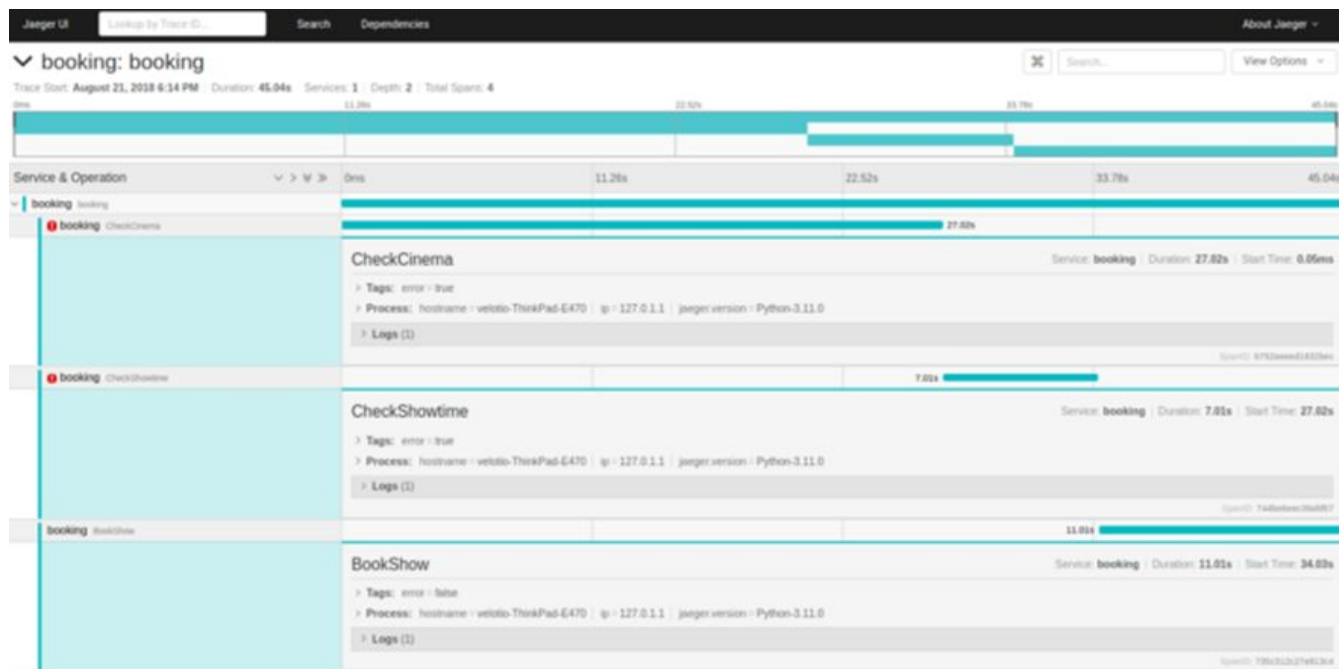
- Jaeger Query is a service that hosts a REST-based query API and provides Jaeger's React-based UI.
- The Jaeger Ingestor is a service that reads spans from Kafka topics, and writes them to another backend, e.g., Apache Cassandra or Elasticsearch.

Notes

Jaeger Injester: <https://www.jaegertracing.io/docs/1.8/deployment/#ingester>

Jaeger Query: <https://www.jaegertracing.io/docs/1.8/deployment/#query-service-ui>

5.53 Jaeger UI Example



A Comprehensive Tutorial to Implementing OpenTracing With Jaeger (<https://medium.com/velotio-perspectives/a-comprehensive-tutorial-to-implementing-opentracing-with-jaeger-a01752e1a8ce>) is a useful supplemental resource.

5.54 Jaeger and Prometheus

- Jaeger, itself, being a microservice-based solution, and being part of the Cloud Native Computing Foundation ecosystem, exposes its own metrics using Prometheus.

Notes

Jaeger Metrics: <https://www.jaegertracing.io/docs/1.8/monitoring/>

5.55 Jaeger and Istio

- Istio supports OpenTracing out of the box, using either Jaeger or Apache Zipkin.
- Istio can automatically generate and send spans on behalf of applications, but applications are required to propagate specific HTTP headers from incoming requests to outgoing requests.
 - ◇ Details are in the student notes

Notes

Distributed Tracing (<https://istio.io/docs/tasks/telemetry/distributed-tracing/>) for Istio covers how to configure Istio for distributed tracing.

Configuring Istio to use Jaeger: <https://istio.io/docs/tasks/telemetry/distributed-tracing/jaeger/>

The HTTP headers necessary to propagate, and sample source code illustrating the task:
<https://istio.io/docs/tasks/telemetry/distributed-tracing/overview/#understanding-what-happened>

5.56 Tracing Using Jaeger Summary

- OpenTracing and its Fundamental Concepts: span and trace
- Jaeger – a Distributed Tracing System, from Über
- Jaeger Client Libraries
- Agent
- Collector
- Query
- Ingester

5.57 Summary

- In this module we explored logging and tracing solutions
- Microservices need to be highly monitorable

- Logs are treated as streams of events
- We have explored how to use Prometheus to instrument our application
- Jaeger can be used to implement tracing in our applications

Chapter 6 - Shared Libraries

Objectives

Key objectives of this chapter

- Defining Shared Libraries
- Using Shared Libraries
- Solutions Delivery Platform
- Jenkins Templating Plugin
- SDP Pipeline Libraries

6.1 Extending with Shared Libraries

- Parts of Pipelines can be shared between various projects to reduce redundancies and keep code DRY (Don't Repeat Yourself).
- A Shared Library is defined with a name, a source code retrieval method such as by SCM, and optionally a default version.
- Version can be anything understood by the SCM, e.g. branches, tags, and commit hashes.
- A library can be loaded implicitly or explicitly

6.2 Directory Structure

```
(root)
+- src                                # Groovy source files
|   +- org
|       +- foo
|           +- Bar.groovy # for org.foo.Bar class
+- vars
|   +- foo.groovy         # for global 'foo' variable
|   +- foo.txt            # help for 'foo' variable
+- resources              # resource files (external libraries only)
|   +- org
|       +- foo
|           +- bar.json   # static helper data for org.foo.Bar
```

- Can be defined at various levels

- src directory should look like standard Java source directory structure. This directory is added to the classpath when executing Pipelines.
- The vars directory hosts scripts that define global variables accessible from Pipeline. The basename of each *.groovy file should be a Groovy identifier, conventionally camelCased. The matching *.txt can contain documentation.
- The resources directory allows usage from an external library to load associated non-Groovy files.

6.3 Sample Groovy Code

```
package com.abcinc;

def checkout() {
    node {
        stage 'Checkout'
        git url: 'C:\\Software\\repos\\SimpleGreeting.git'
    }
}
```

6.4 Defining Shared Libraries

- Shared libraries can be defined at various levels:
 - ◇ Global Shared Libraries
 - Manage Jenkins > Configure System > Global Pipeline Libraries
 - Folder-level Shared Libraries
 - Automatic Shared Libraries
 - e.g. GitHub Organization Folder

The screenshot shows the Jenkins Shared Library configuration interface. It includes fields for 'Name' (my-shared-library), 'Default version', 'Load implicitly' (checkbox), and 'Allow default version to be overridden' (checkbox, checked). Below these are sections for 'Retrieval method' (Modern SCM) and 'Source Code Management' (Git). The 'Project Repository' field contains 'git://git.example.com/pipeline-library.git'. The 'Credentials' dropdown is set to 'none' with an 'Add' button. 'Ignore on push notifications' is a checkbox. The 'Repository browser' dropdown is set to '(Auto)'.

6.5 Using Shared Libraries

- Shared libraries can be utilized in various places

- ◇ Pipeline
- ◇ Execute system Groovy script
- ◇ Execute Groovy script
- ◇ Execute PostBuild script

- Loading libraries explicitly

```
@Library('my-shared-lib') _  
@Library('my-shared-lib@master') _  
@Library(['my-shared-lib', 'another-shared-lib']) _
```

- Conventionally the annotation goes on an **import** statement

```
@Library('my-shared-lib')  
import com.abcinc.utils;
```

6.6 Sample Shared Library Usage Code

```
@Library('my-shared-lib')  
import com.abcinc.utils;
```

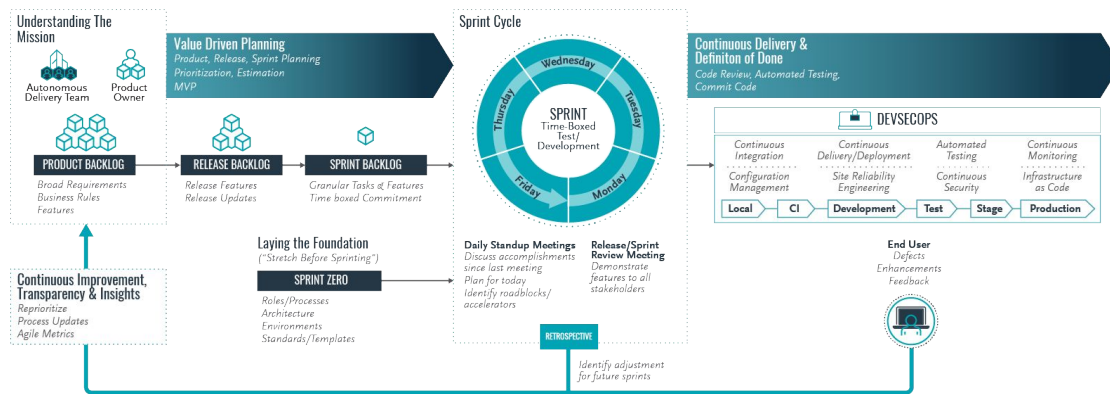
```
def u = new utils();  
u.checkout();
```

6.7 Defining Global Variables

```
// vars/acme.groovy  
def setName(value) {  
    name = value;  
}  
def getName() {  
    return name;  
}  
  
// src/com/abcinc/sample.groovy  
def myFunction() {  
    name = "Bob";  
}
```

6.8 Solutions Delivery Platform

- The Solutions Delivery Platform (SDP) is an accumulation of Booz Allen's lessons learned implementing DevSecOps practices across multiple client projects and RFP Tech Challenges such as NSF, GSA IAE, CMS PECOS, and Grants.Gov.
- At Booz Allen, DevSecOps is a foundational element of our Modern Software development Approach
- Over the past two years, we've developed an open-source and reusable pipeline framework that jump starts projects.
- SDP has allowed the typical time to develop a pipeline from 3 to 4 months down to just a week.
- Instead of creating per-application pipelines, SDP allows you to create tool-agnostic, templated workflows that can be used by multiple teams to achieve organizational governance.



6.9 SDP - Components

- **Container Orchestration Platform:** The container orchestration platform, typically a kubernetes based solution, is responsible for hosting the DevOps tools; such as Jenkins or SonarQube.
 - ◇ There is no strict dependency on OpenShift or Kubernetes, though this will enable a more robust platform.
- **Jenkins Templating Engine:** The Jenkins Templating Engine is a custom plugin written by Booz Allen that enables the capabilities of the Solutions Delivery Platform; such as organizational governance through templating and hierarchical configuration files.
 - ◇ This plugin is available in the Jenkins Update Center under the name Templating Engine Plugin.
- **SDP Pipeline Libraries:** The SDP Pipeline Libraries are our reusable tool integrations that contain the actual technical implementations of pipeline actions; such as static code analysis, penetration testing, and deployments.

6.10 Jenkins Templating Plugin

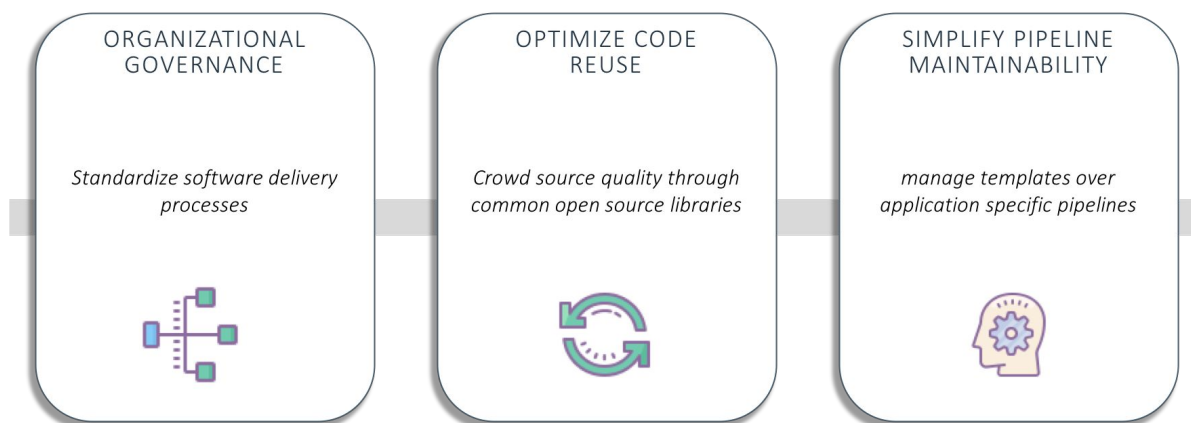
- The Jenkins Templating Engine (JTE - <https://plugins.jenkins.io/templating-engine>) is a plugin developed by Booz Allen Hamilton
- JTE enables pipeline templating and governance.
- JTE allows you to consolidate pipelines into shareable workflows that

define the business logic of your software delivery processes

- JTE allows optimal pipeline code reuse by pulling out tool specific implementations into library modules.

6.11 Why Templating?

- Pipelines are typically defined on a per application basis via a Jenkinsfile in the source code repository.
- Often, common code can be pulled out into Shared Libraries to reduce code duplication but a few problems remain.
- Organization's application portfolios often have a diverse technology stack, each requiring their pipeline-as-code integrations.
- It becomes increasingly complex to manage these different implementations across an organization while standardizing on the required software delivery processes.
- For organizations with strict governance requirements having a Jenkinsfile within the source code repository can allow developers to bypass these mandatory requirements.



6.12 Why Templating? (Contd.)

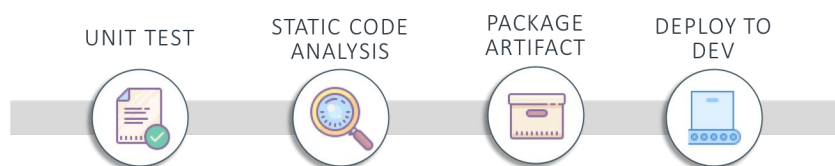
- **Organizational Governance:** The real power of JTE comes from creating shareable workflows and using them across teams.
 - ◇ JTE allows you to create a centralized hierarchical governance for your

pipeline configurations that aligns to your organizational structure.

- ◇ With governance tiers, organizations can accelerate enterprise adoption by creating inheritable pipelines that share common configurations.
- **Optimize Code Reuse:** JTE allows you to share pipeline code effectively across multiple teams through the use of common pipeline libraries.
- **Simplify Pipeline Maintainability:** When supporting pipelines for multiple applications it becomes more challenging to maintain the associated complexity.
 - ◇ Multiple tech stacks and tool integrations can lead to largely similar Jenkinsfiles with distinct differences around how specific parts of the pipeline function.

6.13 Pipeline Templating - Templates

- When developing a pipeline used by multiple teams, you'll often find that the workflow is the same while the specifics of what each step in the process does may differ between applications.
- Whether the application is a Java app packaged as a war and deployed to an AWS EC2 instance or a React app being statically bundled and deployed to an S3 bucket or Nginx instance, the steps in your pipeline are the same.
- In this example, both applications could have a pipeline workflow that performs unit testing, static code analysis, packaging an artifact, and deploying that artifact to an application environment.



- You would represent this workflow in a template! Templates are built by calling steps and referencing primitives defined in your pipeline configuration file.

6.14 Pipeline Templating - Steps

- Pipeline steps are methods supplied by pipeline libraries that implement a specific action in your pipeline template.
- In your pipeline configuration file, you will specify what libraries should be loaded.
- Each of these libraries will contribute steps that can be called from your pipeline template.
- In our simple example, a pipeline template could be defined as follows:

```
unit_test()  
static_code_analysis()  
build()  
deploy_to dev
```

Note:

The template on the slide could be reused by multiple teams using different tools by creating libraries which implement the pipeline steps `unit_test`, `static_code_analysis`, `build`, and `deploy_to`. Once these libraries are created, a configuration file is created to specify which libraries to load.

For the deployment step, you'll notice that a dev application environment was referenced to specify where the deployment should take place. This dev application environment is an example of a primitive that can be defined alongside your template in the configuration file.

6.15 Pipeline Templating - Primitives

- Primitives are template constructs whose purpose is to make the pipeline template as simple and easy to read as possible.
- The default primitives currently supported are Application Environments, Keywords, and Stages.
 - ◇ **Application Environments:** The Application Environment primitive exists to simplify referencing deployment targets from pipeline templates. The `application_environments` key is used to begin declaring application environments.
 - ◇ **Keywords:** Keywords provide some syntactic sugar for defining variables outside of the pipeline template. Within the keywords block,

any configuration defined will be made available within the pipeline template.

- ◊ **Stages:** Stages are a primitive that allows you to group steps to be called at once to avoid repetition in the pipeline template. Stages are defined through the stages key, with subkeys becoming available as steps within your pipeline template.
- It is possible to write a Jenkins Plugin that extends the Jenkins Templating Engine to add a custom primitive.

Note:

Application Environments

An example specification defining a dev and prod environment would be:

```
application_environments{
  dev{
    long_name = "Development"
  }
  prod{
    long_name = "Production"
  }
}
```

Then within a pipeline template, assuming there is a pipeline step called `deploy_to` that accepts an application environment as an argument, you could reference these objects via

```
build()
deploy_to dev
test()
deploy_to prod
```

Keywords:

The default Keywords are:

```
keywords{
  master = /^[Mm]aster$/
  develop = /^[Dd]evelop(ment|er)$/
  hotfix = /^[Hh]ot[Ff]ix-/
  release = /^[Rr]elease-(\d+.)*\d$/
}
```

Stages:

A common example would be creating a continuous integration stage:

```
stages{
```

```
    continuous_integration{
        unit_test
        static_code_analysis
        build
        scan_artifact
    }
}
```

and then in your template:

```
continuous_integration()
deploy_to dev
```

6.16 Pipeline Templating - Configuration Files

- Configuration files provide the information necessary to populate a pipeline template with what's needed to execute.
- For our example, we can define a common pipeline configuration to be inherited by both applications that define the development application environment and a common SonarQube pipeline library to perform the static code analysis:

```
// define common application environment
application_environments{
    dev{
        long_name = "Development"
        ec2{
            ips = [ "1.2.3.4", "1.2.3.5" ]
            credential_id = "ec2_ssh"
        }
        s3{
            url = "https://s3.example.com"
            path = "content/"
            credential_id = "s3_bucket"
        }
    }
}

// define pipeline libraries common between applications
libraries{
    merge = true
    sonarqube // supplies the static_code_analysis step
}
```

6.17 Pipeline Libraries

- There are various reusable pipeline libraries available at: <https://boozallen.github.io/sdp-libraries/>
- Currently, the following libraries are available:
 - ◇ The A11y Machine
 - ◇ Docker
 - ◇ GitHub
 - ◇ GitHub Enterprise
 - ◇ OpenShift
 - ◇ OWASP Dependency Check
 - ◇ OWASP ZAP
 - ◇ Protractor
 - ◇ SDP
 - ◇ Slack
 - ◇ SonarQube
 - ◇ Twistlock

6.18 SDP Pipeline Libraries - Examples

- Here's a sample Docker library configuration snippet

```
libraries{
  docker {
    build_strategy = "dockerfile"
    registry = "docker-registry.default.svc:5000"
    cred = "openshift-docker-registry"
    repo_path_prefix = "proj-images"
  }
}
```

6.19 Summary

- Shared Libraries allow workflow/business logic reuse

- The Solutions Delivery Platform (SDP) is an accumulation of Booz Allen's lessons learned implementing DevSecOps practices across multiple client projects and RFP Tech Challenges such as NSF, GSA IAE, CMS PECOS, and Grants.Gov.
- Jenkins Templating Plugin enables pipeline templating and governance.

Chapter 7 - Best Practices for Jenkins [OPTIONAL]

Objectives

Key objectives of this chapter

- Best Practices.

7.1 Best Practices - Secure Jenkins

- Always secure Jenkins.
 - ◇ Without security, the Jenkins dashboard makes your source code available to anyone
 - ◇ Also, pre-release artifacts
- Even if most of your users are trusted, the setup of projects is sometimes complicated
 - ◇ Securing those projects helps prevent accidents
- See the security chapter

7.2 Best Practices - Users

- Jenkins will need to have user credentials on other systems
 - ◇ e.g. to read from SCM, deploy software, etc
- **Don't let users put their real credentials into jobs or pipelines!**
- Create a user account for Jenkins itself
- Don't publish the login credentials for this "server account"
 - ◇ Use the credentials system in Jenkins
 - ◇ You can grant usage of the credentials through Jenkins security subsystem

7.3 Best Practices - Backups

- Everything of any interest is stored in the Jenkins Home directory
 - ◇ Back it up regularly
- JENKINS_HOME/jobs
 - ◇ contains all the job information
 - ◇ You can copy jobs from one Jenkins installation to another
- JENKINS_HOME/workspace
 - ◇ is where the checkouts from version control go to
 - ◇ Could be skipped in backups if space is a problem
- Archive unused jobs before removing them.
 - ◇ Store a copy of the project's folder under JENKINS_HOME/jobs
 - ◇ If you need to restore, copy the archived folder back into 'jobs', then 'Manage Jenkins --> Reload config from disk'
 - ◇ Can be done while Jenkins is running

7.4 Best Practices - Reproducible Builds

- Ensure a build is reproducible, the build must be a clean build, which is built fully from Source Code Control, plus authorized binaries
- All code including build scripts, release notes, etc. must be checked into Source Code Control.
 - ◇ If it isn't in Version Control, it didn't happen!
- Dependencies in Version Control?
 - ◇ This is a contentious issue
 - ◇ Old-school says "put the dependency jars in version control"

- ◊ Modern approaches usually use binary repository like Nexus, Artifactory, or Apache Archiva for dependencies
- ◊ Discuss, set policy and follow it
- Repeatable build is always the goal!

7.5 Best Practices - Testing and Reports

- Always configure your job to generate trend reports and automated testing when running a Java build
- Make your build self-testing
 - ◊ Run tests as part of the build process
 - ◊ Provide rapid feedback
 - define a test pipeline with distinct test phases
- Take steps to ensure failures are reported as soon as possible.
 - ◊ For example, it may be appropriate to run a limited set of "sniff tests" before the full suite.
- Integrate with your issue management system as much as possible
- Set up email notifications mapping to ALL developers in the project, so that everyone on the team has his or her pulse on the project's current status.
- Tag, label, or baseline the code-base after a successful build.

7.6 Best Practices - Large Systems

- For integration tests, and multiple jobs, allocate a different port for parallel project builds and avoid scheduling all jobs to start at the same time
- If builds conflict, look into the "Locks and Latches" plugin, or the "Throttle Concurrent Builds" plugin

- Multiple jobs running at the same time can cause resource overload.
 - ◇ Try to avoid scheduling all jobs to start at the same time.
 - e.g. Use 'H/5' rather than '*/5' in the schedule entries.
 - Spreads out the start time
- Write jobs for your maintenance tasks, such as cleanup operations to avoid full disk problems.

7.7 Best Practices - Distributed Jenkins

- In larger systems, use distributed builds and disallow build on the master machine. This ensures that the Jenkins master can scale to support many more jobs than if it had to process build jobs directly as well.
- Be aware that Jenkins may use a lot of disk space!
- Make sure your installation process for Jenkins Master and Agents are documented and automated
 - ◇ Either use a configuration management system (Ansible, Puppet, Chef, etc)
 - ◇ Or use a containerized agent (Docker)

7.8 Best Practices - Summary

- ◇ Backup!
- ◇ Repeatable builds!
- ◇ Communicate!
- ◇ Use Jenkins' features
 - Testing and Reporting
 - Distributed Jenkins

Chapter 8 - CI/CD with Docker, Kubernetes, Jenkins, and Blue Ocean [OPTIONAL]

Objectives

Key objectives of this chapter

- Jenkins
- Jenkins Pipeline
- Integrating Docker and Kubernetes with Jenkins
- Blue Ocean

8.1 Jenkins Continuous Integration

- Originally developed at Sun by Kohsuke Kawaguchi?
 - ◇ Originally “Hudson” on java.net circa 2005
 - ◇ Jenkins forked in November 2010
 - ◇ Hudson is still live, part of Eclipse Foundation
 - ◇ But Jenkins seems to be far more active

8.2 Jenkins Features

- Executes jobs based on a number of triggers
 - ◇ Change in a version control system
 - ◇ Time
 - ◇ Manual Trigger
- A Job consists of some instructions
 - ◇ Run a script
 - ◇ Execute a Maven project or Ant File
 - ◇ Run an operating system command
- User Interface can gather reports
 - ◇ Each job has a dashboard showing recent executions

8.3 Running Jenkins

- You can run Jenkins Standalone or inside a web container
- You can setup distributed instances that cooperate on building software
- Can setup jobs in place of what might have been script commands.

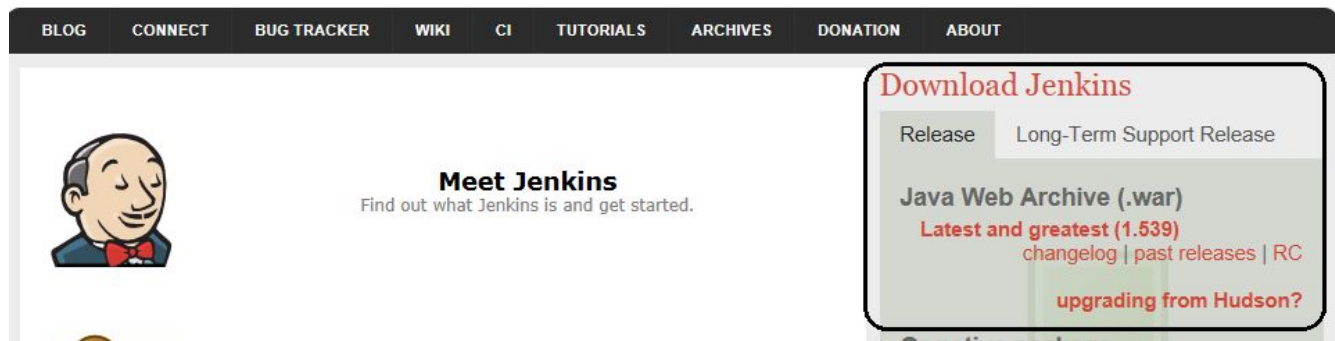
8.4 Downloading and Installing Jenkins

- Download Jenkins from the Jenkins website (<http://jenkins-ci.org>)
 - Jenkins is a dynamic project, and new releases come out at a regular rate.
- Jenkins distribution is bundled in Java web application (a WAR file).
- Windows users, there is a graphical Windows installation MSI package for Jenkins.



Jenkins

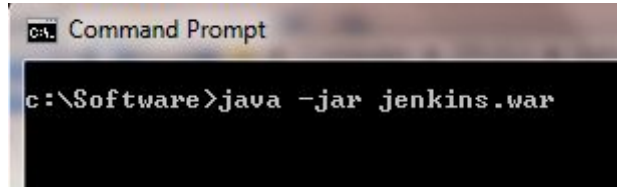
An extendable open source continuous integration server



8.5 Running Jenkins as a Stand-Alone Application

- Jenkins comes bundled as a WAR file that you can run directly using an embedded Servlet container.
- Jenkins uses the lightweight Servlet engine to allow you to run the server out of the box.

- Flexible to install plug-ins and upgrades on the fly.
- To run Jenkins using the embedded Servlet container, just go to the command line and type the following:



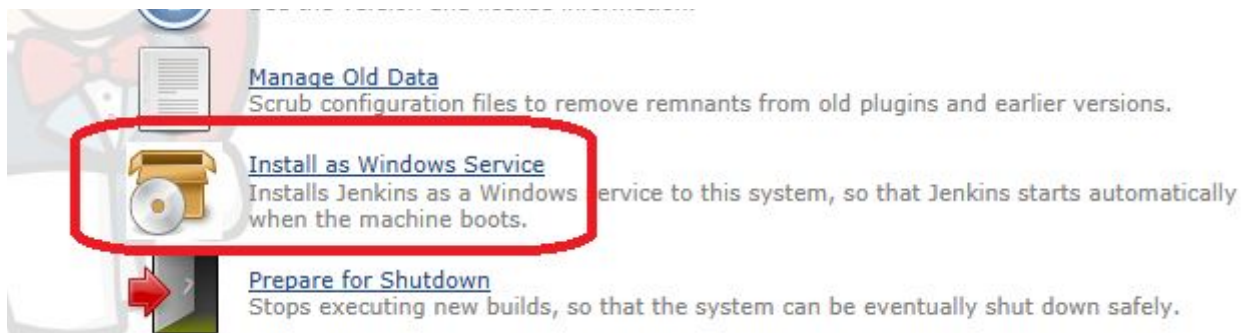
- The Jenkins web application will now be available on port 8080.
- Jenkins can be access directly using the server URL (http://localhost:8080).
- To stop Jenkins, just press Ctrl-C.
- Useful Options:
 - --httpPort
 - By default, Jenkins will run on the 8080 port.
 - Jenkins can be start on different port using the --httpPort option:
 - java -jar jenkins.war --httpPort=8081
 - --logfile
 - By default, Jenkins writes its logfile into the current directory.
 - Option to redirect your messages to other file:
 - java -jar jenkins.war --logfile=C:/Software/log/jenkins.log
- These options can also be set at JENKINS_HOME/jenkins.xml config file.

8.6 Running Jenkins on an Application Server

- Jenkins distribution WAR file can be easily deploy to standard Java application server such as Apache Tomcat, Jetty, or GlassFish.
- Jenkins will be executed in its own web application context (typically "jenkins").
 - URL : <http://localhost:8080/jenkins>.

8.7 Installing Jenkins as a Windows Service

- In production, installation of Jenkins on a Windows box is essential to have it running as a Windows service.
 - Jenkins will automatically start whenever the server reboots
 - Can be managed using the standard Windows administration tools.
- Jenkins using the windows installer automatically runs Jenkins as a windows service.
 - No need to do anything.
- First, Start the Jenkins server on your machine:
 - `java -jar jenkins.war`
- Connect to Jenkins by going to the following URL `http://<hostname>:8080`
- Look for "Install as Windows Service" link in the "Manage Jenkins" page.



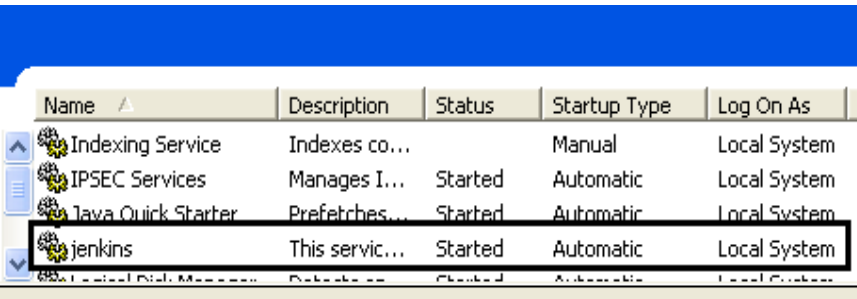
- Clicking this link shows you the installation screen:



- Choose a directory where Jenkins will be installed, JENKINS_HOME and

used to store data files and programs

- Upon successful completion of the installation, you should see a page asking you to restart Jenkins.
- At this point you can use the service manager to confirm that Jenkins is running as a service.

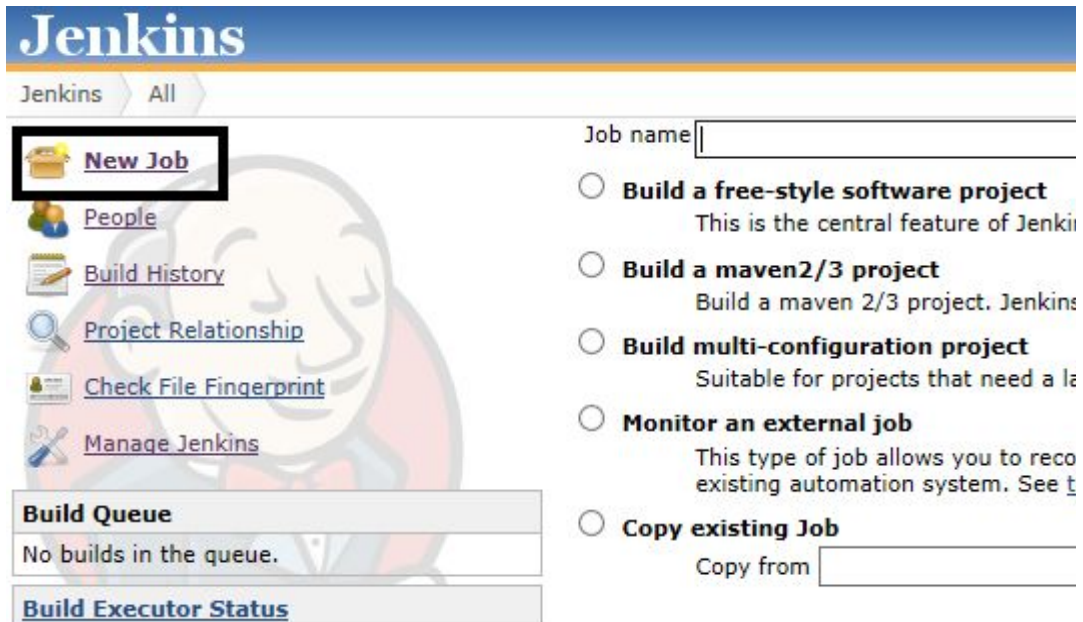


A screenshot of the Windows Services console. The 'jenkins' service is highlighted with a black rectangle. The table below represents the data visible in the screenshot.

Name	Description	Status	Startup Type	Log On As
Indexing Service	Indexes co...		Manual	Local System
IPSEC Services	Manages I...	Started	Automatic	Local System
Java Quick Starter	Prefetches...	Started	Automatic	Local System
jenkins	This servic...	Started	Automatic	Local System
Local Disk (C:)	Default...	Started	Automatic	Local System

8.8 Different types of Jenkins job

- Jenkins supports several different types of build jobs
 - Freestyle software project:
 - Freestyle projects are general purpose and allow to configure any sort of build job.
 - Highly flexible and very configurable.
 - Maven project:
 - Jenkins understands Maven pom files and project structures.
 - Reduce the work needed to do to set up the project.
 - Monitor an external job:
 - Monitoring the non-interactive execution of processes, such as cron jobs.
 - Multi-configuration job:
 - Run same build job in many different configurations.
 - Powerful feature, useful for testing an application in many different environments.



8.9 Configuring Source Code Management(SCM)

- Monitors version control system, and checks out the latest changes as they occur.
- Compiles and tests the most recent version of the code.
- Simply check out and build the latest version of the source code on a regular basis.
- SCM configuration options in Jenkins are identical across all sorts of build jobs.
- Jenkins supports CVS and Subversion out of the box, with built-in support for Git
- Integrates with a large number of other version control systems via plugins.

8.10 Working with Subversion


- Simply provide the corresponding Subversion URL
 - Supported protocols http, svn, or file.

- Jenkins will check that the URL is valid as soon as you enter it.
- If authentication needed, Jenkins will prompt you for the corresponding credentials automatically, and store them for any other build jobs that access this repository.
- Fine-tune Jenkins to obtain the latest source code from your Subversion repository by selecting an appropriate value in the Check-out Strategy drop-down list.
- Choose check-out Strategy as “Use ‘svn update’ as much as possible, with ‘svn revert’ before update”
 - No local files are modified, though it will not remove any new files that have been created during the build process.
 - You might want other options, depending on the load on your svn server.

8.11 Working with Subversion (cont'd)

Source Code Management

☐ CVS
☐ None
☒ Subversion

Modules Repository URL
 **Repository URL is required.**

Local module directory (optional)

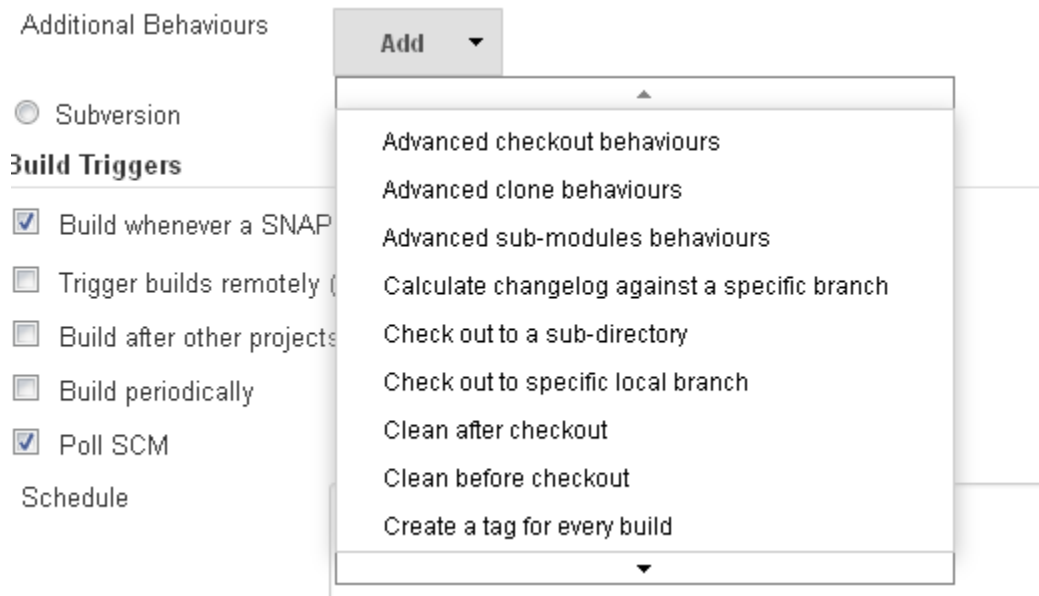
Check-out Strategy
Use 'svn update' whenever possible, making the build faster. But this causes the artifacts from the previous build to remain when a new build starts.

Repository browser

8.12 Working with Git

- Oddly enough, Git support isn't enabled by default
- Enable the 'Git Plugin' through the plugin management screen
- Any type of remote repository can be used:
 - ◇ Could be https, ssh, or local
- Jenkins will check that the URL is valid as soon as you enter it.

- You can store ssh credentials or http credentials
- There are a wide variety of other "Additional Checkout Behaviors" that are possible



8.13 Build Triggers

- In a Freestyle build, there are three basic ways a build job can be triggered
 - Start a build job once another build job has completed
 - Kick off builds at periodical intervals
 - Poll the SCM for changes

Build Triggers

☒ Build after other projects are built
Project names

⊖ No project specified
Multiple projects can be specified like 'abc, def'

☒ Build periodically
Schedule

☒ Poll SCM
Schedule

Ignore post-commit hooks ☐

8.14 Schedule Build Jobs

- Build job at regular intervals.
- For all scheduling tasks, Jenkins uses a cron-style syntax, consisting of five fields separated by white space in the following format:
 - ◇ MINUTE : Minutes within the hour (0–59)
 - ◇ HOUR : The hour of the day (0–23)
 - ◇ DOM : The day of the month (1–31)
 - ◇ MONTH : The month (1–12)
 - ◇ DOW : The day of the week (0–7) where 0 and 7 are Sunday.
- There are also a few short-cuts:
 - ◇ “*” represents all possible values for a field. For example, “* * * * *” means “once a minute.”
 - ◇ You can define ranges using the “M–N” notation. For example “1-5” in the DOW field would mean “Monday to Friday.”
 - ◇ You can use the slash notation to defined skips through a range. For

example, “*/5” in the MINUTE field would mean “every five minutes.”

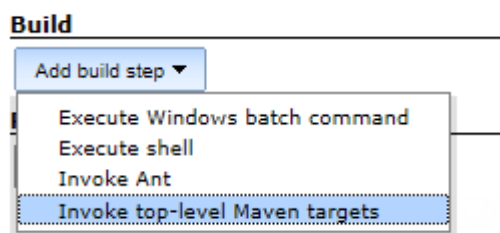
- ◇ A comma-separated list indicates a list of valid values. For example, “15,45” in the MINUTE field would mean “at 15 and 45 minutes past every hour.”

8.15 Polling the SCM

- Poll SVN server at regular intervals if any changes have been committed.
- Jenkins kicks off a build, source code in the project, .
- Polling frequently to ensure that a build kicks off rapidly after changes have been committed.
- The more frequent the polling is, the faster the build jobs will start, and the more accurate.
- In Jenkins, SCM polling is easy to configure, and uses the same cron syntax we discussed previously.

8.16 Maven Build Steps

- Jenkins has excellent Maven support, and Maven build steps are easy to configure and very flexible.
- Select “Invoke top-level Maven targets” from the build step lists.



- Select a version of Maven to run (if you have multiple versions installed)
- Enter the Maven goals you want to run. Jenkins freestyle build jobs work fine with both Maven 2 and Maven 3.
- The optional POM field lets you override the default location of the Maven pom.xml file.

Build

Invoke top-level Maven targets

Maven Version

Goals

8.17 Jenkins / Kubernetes Pipeline

■ Same Pipeline code

```
node {
  stage('Checkout') {
    git url: '/home/osboxes/LabWorks/node-app'
  }
  stage('Build Docker Image') {
    sh 'docker build -t node-app:v1.0 .'
  }
  stage('Delete Service') {
    sh 'kubectl delete service node-app'
  }
  stage('Delete Deployment Configuration') {
    sh 'kubectl delete deployment node-app'
  }
  stage('Redeploy App') {
    sh 'kubectl run node-app --image=node-app:v1.0'
  }
  stage('Expose Service') {
    sh 'kubectl expose deployment node-app --
type=LoadBalancer --port=9090'
  }
}
```

8.18 Jenkins / Kubernetes Pipeline Output

Stage View



8.19 The Blue Ocean Plugin

- Blue Ocean is a project that rethinks the user experience of Jenkins
- It models and presents the process of software delivery by surfacing information that's important to development teams with as few clicks as possible
- It rethinks Jenkins while still staying true to the extensibility that is core to Jenkins.
- Jenkins users can install Blue Ocean side-by-side with the Jenkins Classic UI via a plugin

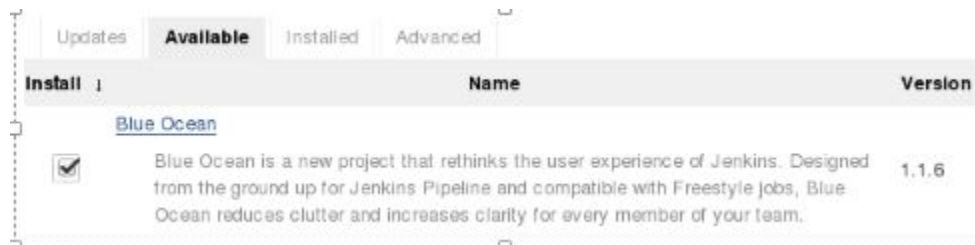
8.20 Blue Ocean Plugin Features

- New modern user experience
- Advanced Pipeline visualizations with built-in failure diagnosis
- Branch and Pull Request awareness
- Personalized View

8.21 Installing Jenkins Plugins

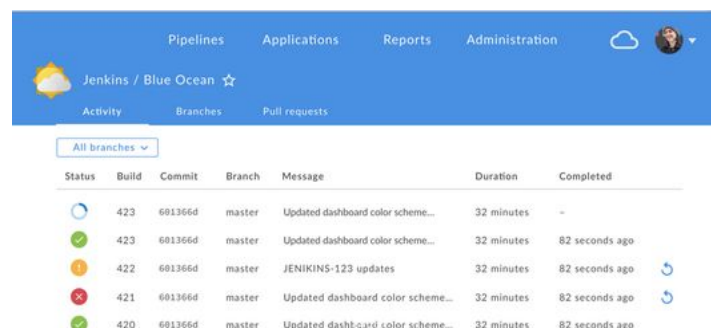
- Click **Manage Jenkins**
- Then on click the **Manage Plugins** link:

- Select the **Available** tab:
- For quick access, enter plugin name on the filter input box, located at right top corner.
- Scroll down the list of plugins to find the Blue Ocean Plugin.
- Select the check box next to Blue Ocean Plugin.
- Click **Download now and install after restart**



8.22 New modern user experience

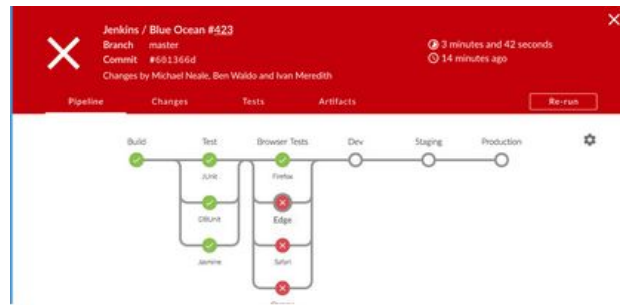
- The UI aims to improve clarity, reduce clutter and navigational depth to make the user experience very concise.
- A modern visual design gives developers much needed relief throughout their daily usage and screens respond instantly to changes on the server without requiring manual page refreshes



8.23 Advanced Pipeline visualizations with built-in failure diagnosis

- Pipelines are visualized on screen along with the steps and logs to allow simplified comprehension of the continuous delivery pipeline – from the

simple to the most sophisticated scenarios.



8.24 Branch and Pull Request awareness

- Modern pipelines make use of multiple Git branches, and Blue Ocean is designed with this in mind.
- When Jenkinsfile defining a pipeline is dropped into a Git repository, it automatically discovers and starts automating any branches and validating pull requests.

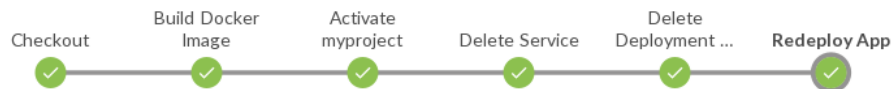
Health	Status	Branch	Latest commit	Latest message	Completed
		master	601366d	Updated dashboard color scheme...	-
		feature1	601366d	Updated dashboard color scheme...	82 seconds ago
		feature2	601366d	Updated dashboard color scheme...	82 seconds ago
		feature3	601366d	Updated dashboard color scheme...	82 seconds ago

8.25 Personalized View

- Blue Ocean offers a dashboard.
- Jobs that need user attention, such as Pipeline awaiting approval or a failing job, appear on the top of the dashboard.



8.26 OpenShift Pipeline Output

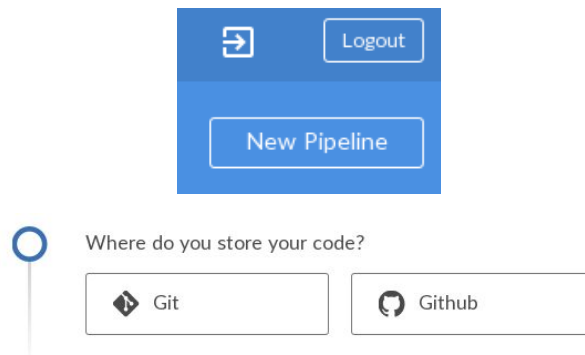


Steps Redeploy App

✓ > oc new-app --docker-image="node-app:v1.0" -- Shell Script

8.27 Creating OpenShift Blue Ocean Pipeline

- You can also create an OpenShift Blue Ocean pipeline from scratch



The image displays the Jenkins Blue Ocean interface. At the top, there's a section titled "Connect to a Git repository" with a subtext: "Any repository containing a Jenkinsfile will be built automatically. Not sure what we are talking about? [Learn more about Jenkinsfile's.](#)". Below this, there's a "Repository URL" field containing "/var/lib/jenkins/repos/hello-app". A "Credentials" dropdown menu is set to "System Default", with an "Add" button next to it. A blue "Create Pipeline" button is at the bottom of this section.

Below the configuration section is a table showing the pipeline run details:

Status	Run	Commit	Branch	Message	Duration	Completed
	1	fbefa91	master	Branch indexing	8s	

Below the table is a visual representation of the pipeline steps: Checkout, Build Docker Image, Activate myproject, Delete Service, Delete Deployment ..., and Redeploy App. Each step is marked with a green checkmark, indicating successful completion.

Below the steps is a section titled "Steps Redeploy App" showing a single step with a green checkmark and the command: `> oc new-app --docker-image="node-app:v1.0" -- Shell Script`.

8.28 Summary

- Jenkins is an open-source CI/CD software.
- Blue Ocean plugin revamps Jenkins UI.
- Kubernetes can be combined with Jenkins and Blue Ocean plugin to implement CI / CD.

Appendix 1 - [Appendix] Operational Readiness

Objectives

Key objectives of this chapter

- What is Operational Readiness?
- Telemetry
- End-to-end requirements traceability?
- Log Strategy
- Monitoring Strategy
- Runbooks
- Operational Readiness Checklist

1.1 What is Operational Readiness

- Ensuring Application Development teams are aware of the constraints of handing off and running applications in production
- Teams in OpenShift are given a sandbox to allow them to test, operationalize and make guardrail mistakes
- Understand the process for Operational Readiness via Appian and the “long pole in the tent”
- Use of consistent tooling for AppDev, CI/CD, Log Management, Release Management
- Creating documentation to show the path to operational readiness
- Automation of release and application tasks in preparation for operational readiness

1.2 Telemetry

- Everything fails eventually
- Telemetry captures the information from the application
- What is your application doing at a point in time? How do you look back

for a baseline or reference?

- Telemetry captures:
 - ◇ System unavailability (for example, if the database is down for a long time, by using an appropriate back-off policy on the telemetry to not exacerbate the failing notifications).
 - ◇ Capture retry logic for command execution for some set of errors.
 - ◇ Include decision streams on retry failures based on the command and error.
 - ◇ Seam telemetry around all external component and service calls to view the end to end processing.
 - ◇ Use consistent asynchronous I/O and APIs where appropriate.
 - ◇ Document recovery scaling up and down (for example, capturing the increasing requests to a newly recovered service).
- Telemetry is the logging mechanism for all of this stuff to be able to track down failures, escalate resolution, identify patterns, and provide reference data.

1.3 End-to-end Requirements Traceability

- Requirements trace from ideation to production.
- Unique tagging for all applications – CIPE, VIBE, NPT.
- All messages from the system are tagged with an identifier.
- CIPE-176 – Update to allow for some requirement down to the task or user story
- This identification traces through all code, infrastructure-as-code or updates to show their path from idea to production.
- In system support the identifiers allow us to trace to the release, code, process, and people who own the issue
- This processing allows us to eventually limit or eliminate all-hands-on-deck

1.4 Log Strategy

- All applications need to have a consistent log strategy
- The log strategy is initially implemented as documentation
- Transition the log strategy to rule-sets in tools like SonarQube to allow for feedback on each asset that is checked into version control.
- Similar patterns for updates in release management, program support office, and any software circle of life task, process, technology.
- Strategies always require us to look at a baseline (where are we now), envision the future state (where do we want to go in the future), and the roadmap (how do we get from now to the future state)
- How would the organization benefit from consistency in logging in all applications?
- What area in your day-to-day needs a logging strategy?

1.5 Monitoring Strategy

- All applications need to have a consistent log strategy
- Is the strategy for your portfolio, group, application, people, and processes clear to everyone inside and outside your group?
- Groups of tasks, processes, applications, and people are monitored together as an affinity
- How does your monitoring strategy change based on priority, characteristics and business goals?
- Strategies always require us to look at a baseline (where are we now), envision the future state (where do we want to go in the future), and the roadmap (how do we get from now to the future state)
- How would the organization benefit from consistency in monitoring in all applications?
- Do you use Digital Performance Monitoring (DPM), Application Performance Monitoring (APM), and purpose-built solutions to identify, escalate, resolve, and document issues proactively?

- Is your team using the CA APM solution currently? Are you aware of the process for improving the monitoring capability for your people, process, applications, and tools?

1.6 Runbooks

- Runbooks are:
 - ◇ Compilation of routine procedures and operations that the system administrator or operator carries out.
 - ◇ System administrators in IT departments and NOCs use runbooks as a reference.
 - ◇ Runbooks can be in either electronic or in physical book form.
 - ◇ Typically, a runbook contains procedures to begin, stop, supervise, and debug the system.
 - ◇ The runbook may also describe procedures for handling special requests and contingencies.
 - ◇ An effective runbook allows other operators, with prerequisite expertise, to effectively manage and troubleshoot a system.
 - ◇ Runbooks processes can be carried out using software tools in a predetermined manner AKA automation.

1.7 Operational Readiness Checklist

- Operational readiness checklist involves the following:
 - ◇ Strategy
 - ◇ Planning
 - ◇ Architecture
 - ◇ Design
 - ◇ Development (Acquire/Build)
 - ◇ Transition
 - ◇ Acceptance

◇ Evaluation

1.8 Strategy

- In organizational strategy, Operational Readiness is as important a goal as customer satisfaction, usability, and ROI.
- Both executives and customers understand the importance of operational readiness
- Systems are not “steamrolled” into production until operationally ready.
- There is a liaison function between Operations, Architecture, Projects, and Development
- Continual improvement pipeline includes Operational Readiness

1.9 Planning

- The Operations team are involved in all service planning
- The PMO supports Operational Readiness and their project templates cover it
- Business cases include Operational Readiness costs and benefits

1.10 Architecture

- Service models include an Operating Model
- Service models include Non-Functional Requirements
- Architectural standards include Operational Readiness

1.11 Design

- Service design includes Non-Functional Requirements for availability, data, operations management, continuity, supplier contracts
- Service Design specifies Operational Acceptance Criteria in conjunction with Operations function.

- Service Design sign-off includes a readiness checklist
- Service added to Service Catalog

1.12 Development (Acquire/Build)

- Systems are developed to meet requirements for availability/performance, data integrity, operations management, and continuity
- Solution development (acquisition, integration and/or build) includes operational procedures and documentation
- Development includes negotiating SLAs with the customers to meet the Non-Functional Requirements. There should be an organizational commitment to the ongoing operational funding and resources to achieve that
- Development includes the negotiation of supplier contracts that align with and support the SLAs

1.13 Transition

- SLAs are adjusted and renegotiated as necessary to recognize the realities of the new system
- Infrastructure is improved as necessary to meet SLAs
- Testing includes an Operational Acceptance phase
- One or more operational staff are involved in testing
- Operating model is demarcated and agreed down to the procedural level
- Roles and accountabilities are assigned and accepted
- Functional and hierarchical escalation paths documented and agreed
- Support staff are properly trained
- Users are properly trained
- Service Catalog is up to date and accurate
- Critical service dependencies on IT components are recorded
- Change implementation plan (Release) approved by all business and IT

stakeholders

1.14 Acceptance

- Operations sign-off is required to put a release into Production
- There are defined deliverables from a release for operability, e.g. support handbook
- Sign-off depends on the adequate transfer of knowledge
- Acceptance includes the project team operating the system for a warranty period

1.15 Evaluation

- Service evaluation includes evaluating that it continues to meet readiness criteria

1.16 Summary

- Operational readiness is a process that develops alongside software engineering
- Telemetry provides common feedback from applications on operations and health
- Strategies for logging and monitoring allow for organizations to create a consistency that benefits everyone in the long run
- Runbooks are key elements of operational readiness

Appendix 2 - [Appendix] Reusable Design Patterns from BAH

Objectives

Key objectives of this chapter

- Code Reviews for Every Pull Request
- Setting Up the Design for a 12 Factor Application

2.1 Code Reviews for Every Pull Request

- In modern software development, code reviews must be performed on every single line of code that gets merged into a deployable branch of a project's version control system.
- No developer should be able to push directly to one of the main trunk branches (dev, staging, and master) without a code review.
- Whether developing new features, fixing bugs, or correcting a single typo, developers should:
 - ◇ branch off of a trunk branch into a feature branch
 - ◇ commit their changes to a feature branch
 - ◇ submit a Pull Request of their changes back to the trunk branch

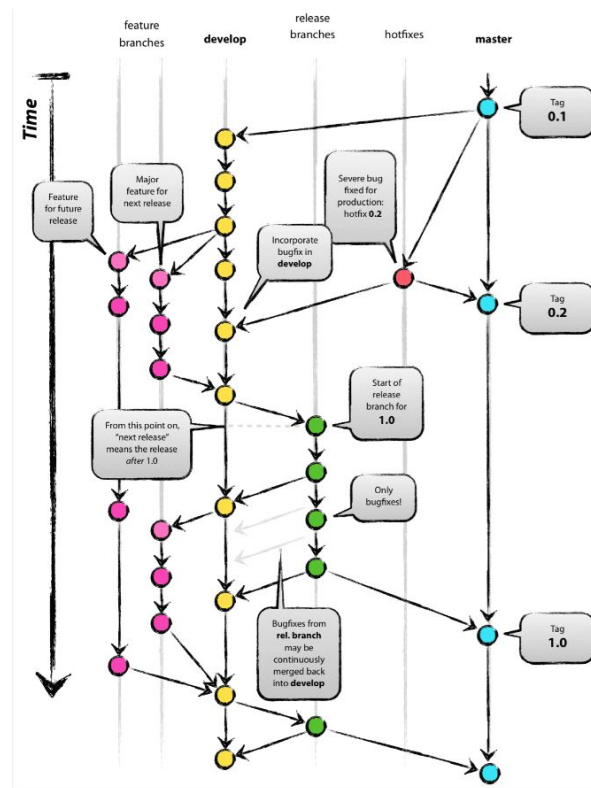


Figure 1 - Branch and Merge Process
(2010 Vincent Driessen)

2.2 Code Reviews for Every Pull Request (Contd.)

- The Pull Requests, commonly referred to as PRs, need to be code reviewed and pass a series of checks – both manual and automated – before being approved and merged into the trunk branch.
- Code reviews serve important functions within a modern software development team:
 - ◇ encourage knowledge sharing
 - ◇ enable de-facto mentoring
 - ◇ help identify bugs before users find them
 - ◇ embolden code consistency and quality

2.3 Code Reviews for Every Pull Request - Implementation Overview

- When a PR is submitted, both the submitter and the reviewer have responsibilities to ensure a review can be performed correctly.
- The submitter should enable the review to understand as much as possible about the reason for the PR and how to go about testing the functionality.
- For teams using GitHub, a .github/pull_request_template.md file can be created at the root of the repository that will automatically populate a template in the description of every new PR.

2.4 Pull Request Template

- The template should include, at minimum, the following elements:
 - ◇ JIRA Ticket:
 - ◇ Description:
 - ◇ Testing Instructions:
 - ◇ Screenshot (optional):
 - ◇ Related Component Library PR Link (optional):
 - ◇ Acceptance Checklist:
 - Code Review (Developer)
 - Design Review (UI/UX)
 - Responsiveness Review (Developer + UI/UX)
 - 508 Review (Developer)

Note:

Code Review (Developer): Is this code using best practices? Is it efficient? Is it extensible? Does this code pass lint? Does this code have high-quality, high-value tests that cover a variety of inputs and data states?

Design Review (UI/UX): Does the page match the mocks? Does the functionality work as intended? Do the interactions function as designed?

Responsiveness Review (Developer + UI/UX): Does this feature look good and function on mobile? Does changing the browser width produce inconsistent UI states? If this PR affects a feature that is currently broken on mobile, fix it as part of the PR.

508 Review (Developer): Run the aXe plugin on the page and ensure there are no accessibility issues. Check the jsx-a11y warnings/errors on the linting output. Does this page introduce any accessibility issues? Are there accessibility issues already present on this page? If so, fix them with the PR.

For frontend code, it is not just developers who should review PRs – designers, both UI and UX, should be able to run the code locally to ensure the visuals match the mockups and the interactions match the intended UX.

Code reviewers should first ensure the code is free of obvious errors, stylistic mistakes, confusing variable names, typos and commented out code, that complicated portions of the code include comments, and that the code has associated tests. Second, reviewers should run the code using the provided instructions to ensure it meets the acceptance criteria of the JIRA ticket, handles unexpected input, and covers edge cases. Essentially – try to break it. Third, after understanding what the code is doing, the reviewer should perform a deeper dive into the code to ensure it is optimized, makes the best use of external libraries, does not replicate existing code/functionality, and has been tested appropriately.

2.5 Setting Up the Design for a 12 Factor Application

- The Twelve-Factor App by Adam Wiggins 2017 (<https://12factor.net>) is a methodology for architecting and composing discrete parts of an application in such a way that the application is free of environment-specific code and the configuration is not directly tied to the environment.
- Each factor covers a core part of the architecture of web service or software-as-a-service (SaaS) applications. With slight modifications to some factors, the methodology is useful for microservice architectures.
- Adopting each factor into your architecture will ensure your application will be cloud-portable and have minimal vendor lock-in.

2.6

2.7 The Twelve-Factor App

#	Factor	Description
I	Codebase	One codebase tracked in revision control; many deploys
II	Dependencies	Explicitly declare and isolate dependencies
III	Config	Store config in the environment
IV	Backing Services	Treat backing services as attached resources
V	Build, Release, Run	Strictly separate build and run stages
VI	Processes	Execute the app as one or more stateless processes
VII	Port Binding	Export services via port binding
VIII	Concurrency	Scale out via the process model
IX	Disposability	Maximize robustness with fast startup and graceful shutdown
X	Dev/Prod Parity	Keep development, staging, and production as similar as possible
XI	Logs	Treat logs as event streams
XII	Admin Processes	Run admin/management tasks as off-off processes

2.8 Categorizing the Twelve Factors

- Code
 - ◇ Codebase
 - ◇ Build, Release, Run
 - ◇ Dev/Prod parity
- Deploy
 - ◇ Dependencies
 - ◇ Config
 - ◇ Processes
 - ◇ Backing Services
 - ◇ Port Binding
- Operate
 - ◇ Concurrency
 - ◇ Disposability
 - ◇ Logs

◇ Admin Processes

2.9 I. Codebase

- **Description:** One codebase tracked in revision control, many deploys
- Each microservice is an application and the rule here is that each application should reside in its own repository.
- Keeping all of the microservices together is not advisable, as it will make change-tracking and deployment more difficult.

2.10 II. Dependencies

- **Description:** Explicitly declare and isolate dependencies
- Working with microservice and Docker best practices helps us here.
- Keeping containers small and compact dictates that we shouldn't be adding anything that won't be used by the process.
- Specialized containers that do specific work can be created and scaled independently.
- Use Docker multi-stage builds to keep deployable containers as small as possible by compiling libraries and copying them as needed.
- Keep the installation of system packages to a minimum.
- If there is too much of a difference between what is run in production than what is run on a developer's local system, it can lead to defects that are hard to detect and resolve.
- Installing dependencies into the container with Dockerfile commands should be secondary to the installation of dependencies using the dependency manager for the source code.

2.11 III. Config

- **Description:** Store config in the environment
- When building user interface (UI), use continuous integration/continuous

delivery (CI/CD) to pass in environment variables that will be used by the Webpack or another build process to produce the correct environment-specific UI.

- Backend services should use environment variables exclusively to configure themselves.
- Docker and the tools designed to run the containers can supply these ENV variables and allow quick maintenance and zero-downtime when making configuration changes.
- Do not use the ENV command in Dockerfile to configure any process, as this would require a rebuild of the container to change.
- Load certifications and other secure configurations with environment variables that contain paths to protected secrets.

2.12 IV. Backing Services

- **Description:** Treat backing services as attached resources
- To a microservice process, everything outside of its domain should be remote including other microservices.
- Microservices must be able to connect to data stores, vendor application program interfaces (APIs) and other services without being directly coupled.
- Locally, a developer must be able to connect to shared development resources, or choose to run and connect to local resources, without making any changes to the code.

2.13 V. Build, Release, Run

- **Description:** Strictly separate build and run stages
- Use a Continuous Integration/Continuous Deployment (CI/CD) tool to build deployable Docker containers.
- Promote containers - do not rebuild for every stage.
- Development should receive new builds each time a watched repository branch is updated.

- Staging and Production will receive a new version of the container either during a manual or automated CI/CD process.

2.14 VI. Processes

- **Description:** Execute the app as one or more stateless processes
- Containers should always be treated like ephemeral file systems, and nothing should be written into one that cannot be lost or relied on existing.
- All of the data that a microservice works with should be kept and read from a backing service (IV).
- When deploying specialized containers that write to their file system, data caches and the like should be avoided so that scaling a microservice doesn't create issues where one container serves data differently from another.

2.15 VII. Port Binding

- **Description:** Export services via port binding
- The use of containers helps enforce process isolation and limits access through the use of ports.
- The API that a microservice exports is the only way to access the data within its domain.
- Sharing a data store and the data it holds would be a cause for concern.
- Each microservice is an API around a black box, and other microservices should never be aware of any backing services (IV) another microservice might use.

2.16 VIII. Concurrency

- **Description:** Scale out via the process model
- A microservice that does too much is a “microlith,” so every service should be designed to perform as focused a task as possible.
- When bottlenecks are noticed, each can be scaled independently.

- Using containers with something like Kubernetes allows the DevOps or Site Reliability Engineering (SRE) teams to easily scale the system based on the load.
- Here the **Zero-One-Infinity** (https://en.wikipedia.org/wiki/Zero_one_infinity_rule) rule also applies, and no service should be written in a way that only one can exist, or a specific number should exist for the system to function.

2.17 IX. Disposability

- **Description:** Maximize robustness with fast startup and graceful shutdown
- Use and deploy services in Docker containers.
- Use backing services instead of storing data within the container or an attached volume.
- Losing a container should never result in loss of data, or application state.
- Treat all services containers as ephemeral, read-only file systems.

2.18 X. Dev/Prod Parity

- **Description:** Keep development, staging, and production as similar as possible
- Build your containers the one time just before their first deployment into development.
- Re-use them in each of the other environments.

2.19 XI. Logs

- **Description:** Treat logs as event streams
- Treating each container like an ephemeral, read-only file system means there shouldn't be any application logging written into it.
- Write timestamped logs to standard output (STDOUT) or standard error (STDERR), and aggregate these using tools like Splunk.

2.20 XII. Admin Processes

- **Description:** Run admin/management tasks as off-off processes
- Run privileged operations in Docker containers that are started, used and then shutdown.
- Containers make this very easy, as you can spin up a container just to run a task and then shut it down.
- For example, doing data cleanup

2.21 Summary

- All code needs to be reviewed before being deployed
- Code should be reviewed both up and down – junior developers **should** be reviewing code of senior developers
- Make it easy for reviewers to understand what they're reviewing
- Code should be reviewed for obvious errors first, then tested for functionality, and then reviewed in more depth
- Checklists of acceptance criteria need to be created and followed when determining when code is ready to be merged.
- Follow the Twelve-Factor App methodology for architecting and composing an application
- Adopting each factor into your architecture will ensure your application will be cloud-portable and have minimal vendor lock-in.

Appendix 3 - [Appendix] Introduction to Kafka

Objectives

Key objectives of this chapter

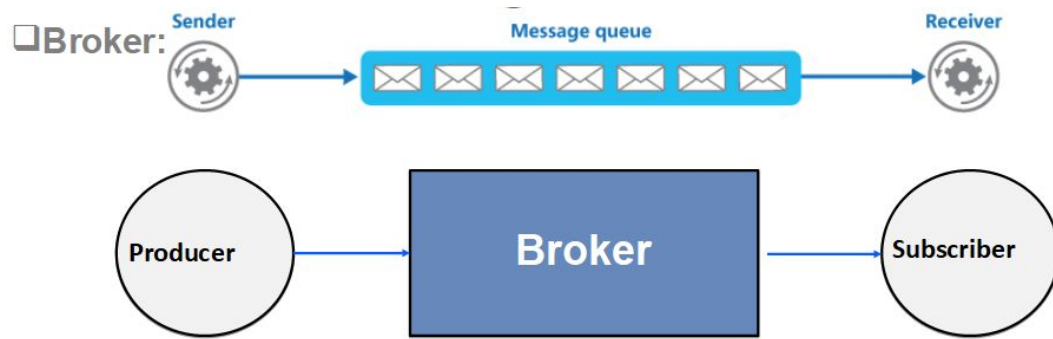
- Messaging Architectures
- What is Kafka?
- Need for Kafka
- Where is Kafka useful?
- Architecture
- Core concepts in Kafka
- Overview of ZooKeeper
- Cluster, Kafka Brokers, Producer, Consumer, Topic

3.1 Introduction

- A typical organization has multiple systems that requires integration or data interchange
- Each integration poses different challenges:
 - ◇ Protocol: how the data is transported (TCP, HTTP, REST, FTP, JDBC, ...)
 - ◇ Data format: how the data is parses (binary, CSV, TSV, JSON, Avro, ...)
 - ◇ Data schema & versioning: how the data is modeled and versioned

3.2 Messaging Architectures – What is Messaging?

- Messaging is application-to-application communication or systems integration technique
- It supports asynchronous operations
- A message is a self-contained package of business data and network routing headers.

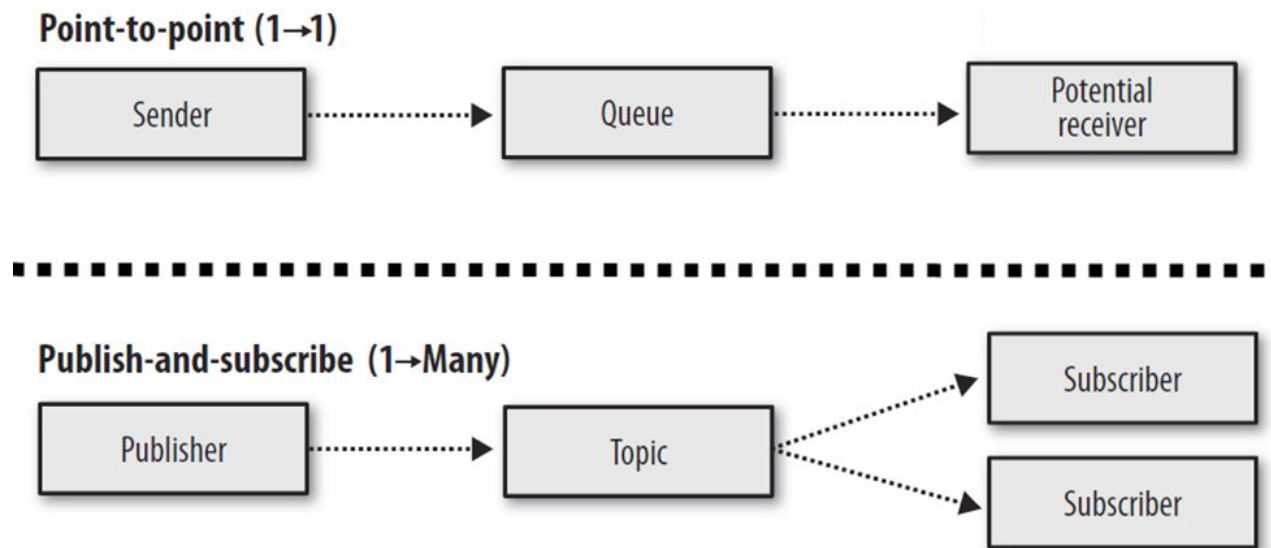


3.3 Messaging Architectures – Steps to Messaging

- Messaging connects multiple applications in an exchange of data.
- Messaging uses an encapsulated asynchronous approach to exchange data through a network.
- A traditional messaging system has two models of abstraction:
 - ◇ Queue – a message channel where a single message is received exactly by one consumer in a point-to-point message-queue pattern. If there are no consumers available, the message is retained until a consumer processes the message.
 - ◇ Topic - a message feed that implements the publish-subscribe pattern and broadcasts messages to consumers that subscribe to that topic.
- A single message is transmitted in five steps:
 - ◇ Create
 - ◇ Send
 - ◇ Deliver
 - ◇ Receive
 - ◇ Process

3.4 Messaging Architectures – Messaging Models

- 1. Point to Point
- 2. Publish and Subscribe



3.5 What is Apache Kafka?

- Apache Kafka is an Open Source Project created by LinkedIn
- Kafka is mainly maintained by Confluent
- Kafka supports horizontal scalability with:
 - ◇ hundreds of brokers
 - ◇ millions of messages per second
- In modern applications, real-time information is continuously generated by applications (publishers/producers) and routed to other applications (subscribers/consumers)
- Kafka has a very low network latency and works in real time
 - ◇ latency of less than 10ms
- Kafka allows integration of information of producers and consumers to avoid any kind of rewriting of an application at either end.

3.6 What is Apache Kafka? (Contd.)

- Kafka is a unique distributed publish-subscribe messaging system written in the Scala language with multi-language support and runs on the Java Virtual Machine (JVM).

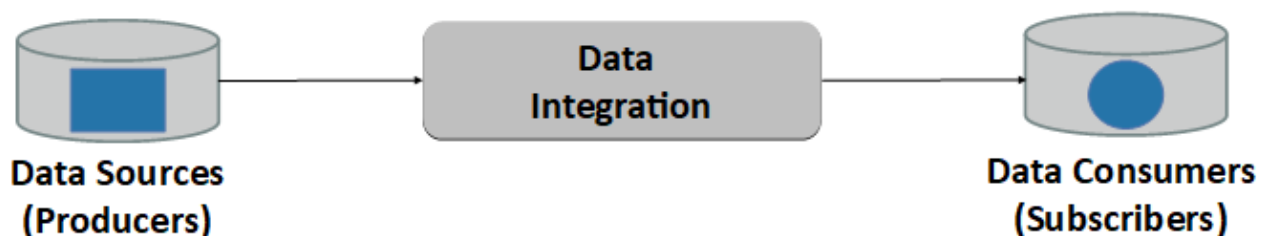
- Kafka relies on another service named Zookeeper – a distributed coordination system – to function.
- Kafka has high-throughput and is built to scale-out in a distributed model on multiple servers.
- Kafka persists messages on disk and can be used for batched consumption as well as real-time applications.

3.7 Who Uses Kafka?

- LinkedIn
- Netflix
- Spotify
- Twitter
- Foursquare
- airbnb
- ...

3.8 Kafka Overview

- When used in the right way and for the right use case, Kafka has unique attributes that make it a highly attractive option for data integration.

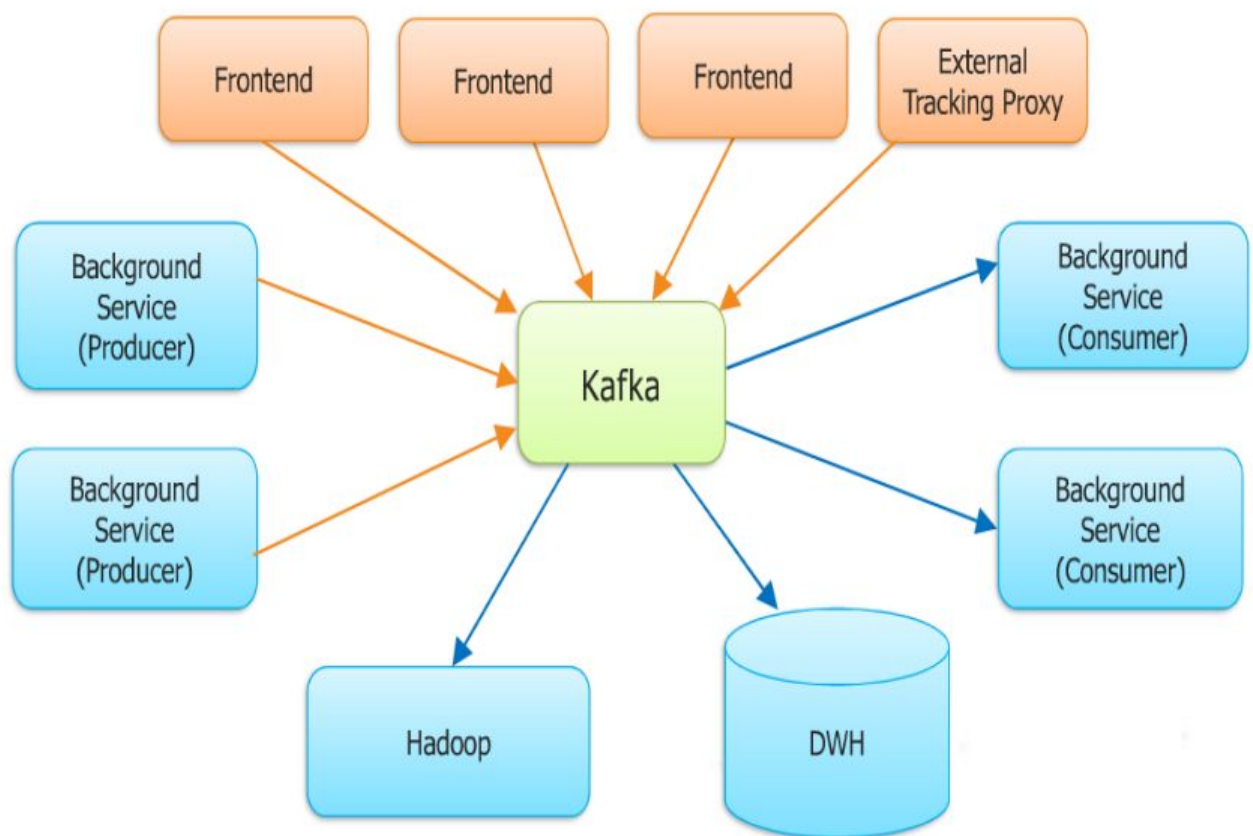


- Data Integration is the combination of technical and business processes used to combine data from disparate sources into meaningful and valuable information.
- A complete data integration solution encompasses discovery, cleansing,

monitoring, transforming and delivery of data from a variety of sources

- Messaging is a key data integration strategy employed in many distributed environments such as the cloud.
- Messaging supports asynchronous operations, enabling you to decouple a process that consumes a service from the process that implements the service.

3.9 Kafka Overview (Contd.)



3.10 Need for Kafka

- High Throughput
 - ◇ Provides support for hundreds of thousands of messages with modest hardware

- Scalability
 - ◇ Highly scalable distributed systems with no downtime
- Replication
 - ◇ Messages can be replicated across a cluster, which provides support for multiple subscribers and also in case of failure balances the consumers
- Durability
 - ◇ Provides support for persistence of messages to disk which can be further used for batch consumption
- Stream Processing
 - ◇ Kafka can be used along with real-time streaming applications like spark, flink, and storm
- Data Loss
 - ◇ Kafka with proper configurations can ensure zero data loss

3.11 When to Use Kafka?

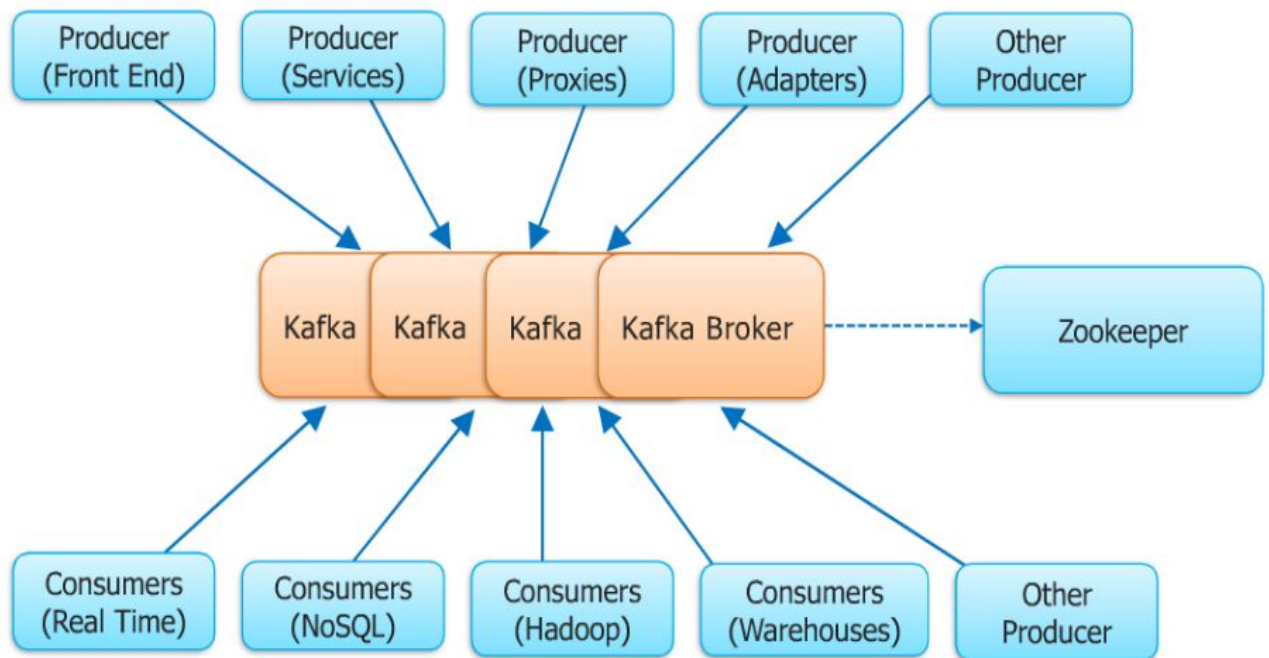
- Kafka is highly useful when you need the following:
 - ◇ a highly distributed messaging system
 - ◇ de-coupling of system dependencies
 - ◇ a messaging system which can scale out exponentially
 - ◇ high throughput on publishing and subscribing
 - ◇ varied consumers having varied capabilities by which to subscribe these published messages in the topics

3.12 When to Use Kafka? (Contd.)

- ◇ gather metrics from many different locations
- ◇ integration with Spark, Flink, Storm, Hadoop, and other Big Data technologies

- ◇ stream processing, such as with Kafka Streams API or Spark
- ◇ a fault tolerance operation
- ◇ durability in message delivery
- ◇ all of the above without tolerating performance degrade

3.13 Kafka Architecture



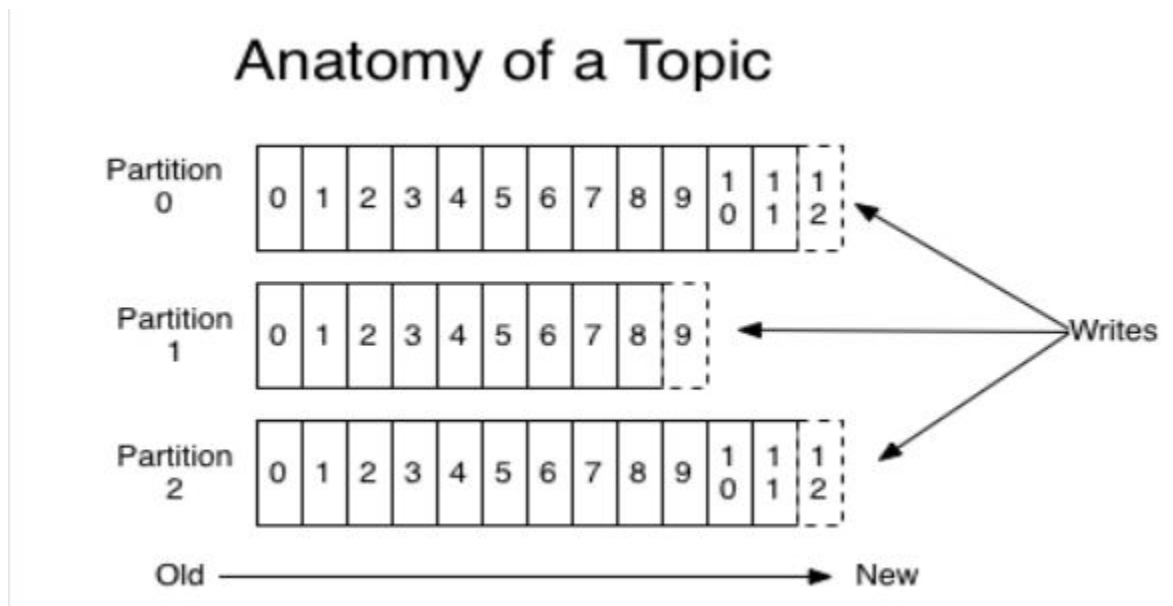
3.14 Core concepts in Kafka

- Topic
 - ◇ A category or feed to which messages are published
- Producer
 - ◇ Publishes messages to Kafka Topic
- Consumer
 - ◇ Subscribes and consumes messages from Kafka Topic
- Broker

- ◇ Handles hundreds of megabytes of reads and writes

3.15 Kafka Topic

- Similar to a database table without all the constraints
- A user defined category where the messages are published
- For each topic, a partition log is maintained
- Each partition basically contains an ordered, immutable sequences of messages where each message assigned a sequential ID number called offset
- Messages are appended to a topic-partition in the order they are sent
- Reading messages from partition can be from the beginning and also can rewind or skip to any point in a partition by supplying an offset value

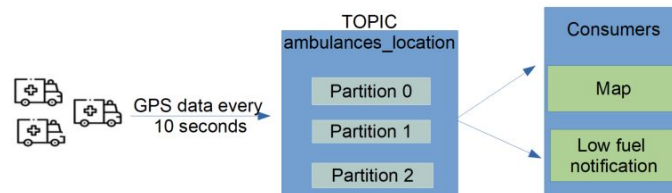


3.16 Kafka Topic (Contd.)

- By default, data is kept in a topic for one week
- If retention is configured for n days, then messages once published, it is available for consumption for configured n days and thereafter it is discarded

- Offset always increments even after data has been deleted
- Consumers read messages in the order stored in a topic-partition

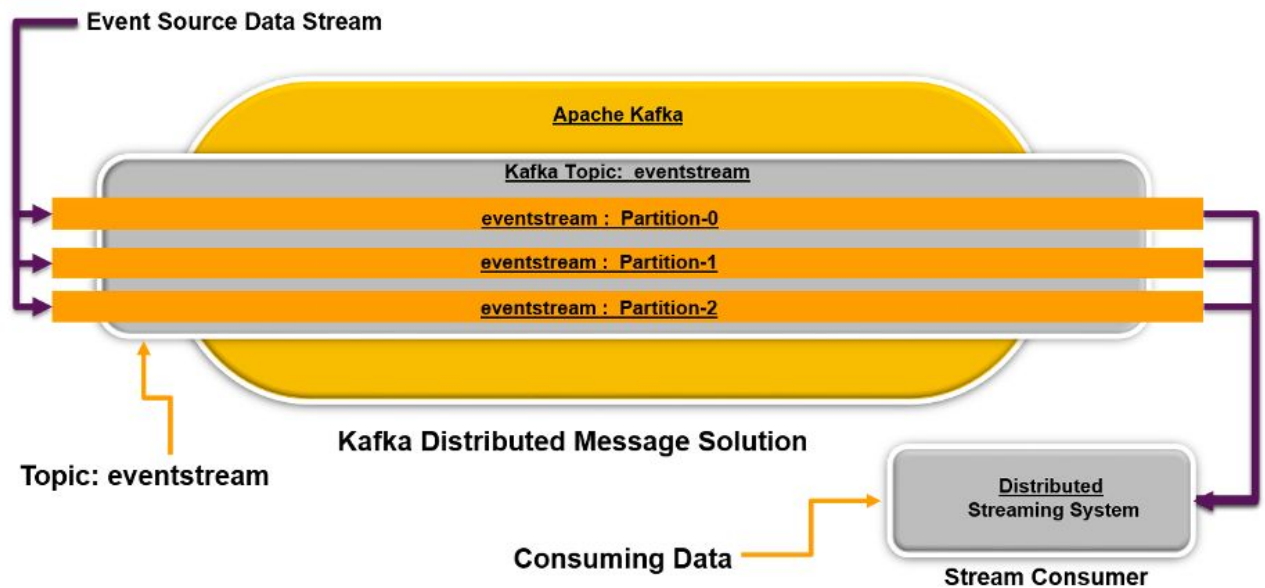
3.17 Topic Example - ambulances_location



- We want to track ambulance id and location
- A single topic, `ambulances_location`, can hold data from all the ambulances
- The topic can be stored in multiple partitions. More on this later in this chapter.

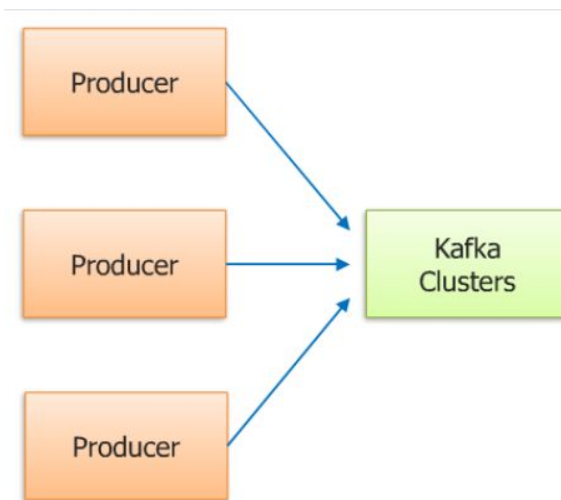
3.18 Kafka Partitions

- In Kafka, a topic can be assigned to a group of messages of a similar type that will be consumed by similar consumers.
- Partitions (a unit of parallelism in Kafka), parallelize the consumption of messages in a Kafka topic with the total number in a cluster not less than the number of consumers in a consumer group.
- Multiple partitions are recommended so the load can be balanced to many brokers



3.19 Kafka Producers

- Application publishes messages to the topic in Kafka Cluster
- Can be of any kind, such as web front-end and, streaming
- While writing messages, it is also possible to attach a key to the message
 - ◇ a key is any type of data, such as string and number
 - ◇ similar to a database table primary key
- If the number of partitions remains unchanged for a topic, the same key will always go to the same partition
 - ◇ key hashing is utilized by Kafka to determine a partition for message with a certain key
- By attaching key the producers basically provide a guarantee that all messages with the same key will arrive in the same partition
- Without the key, data is written in round robin manner
- Supports both async and sync modes
 - ◇ Publishes as many messages as fast as the broker in a cluster can handle



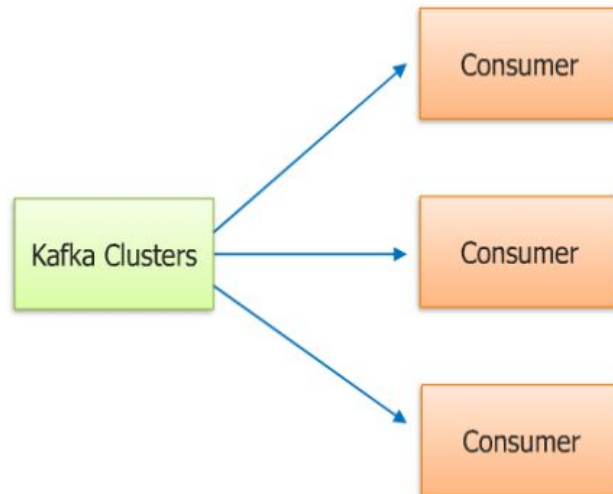
3.20 Kafka Producers - Acknowledgment Modes

- It is optional for Producers to receive acknowledgment of data writes
- There are 3 modes:
 - ◇ acks=0: Producer won't wait for any acknowledgment
 - Possible data loss
 - ◇ acks=1: Producer will wait for leader acknowledgment
 - Limited data loss
 - ◇ acks=all: Leader + replicas acknowledgment
 - No data loss

3.21 Kafka Consumers

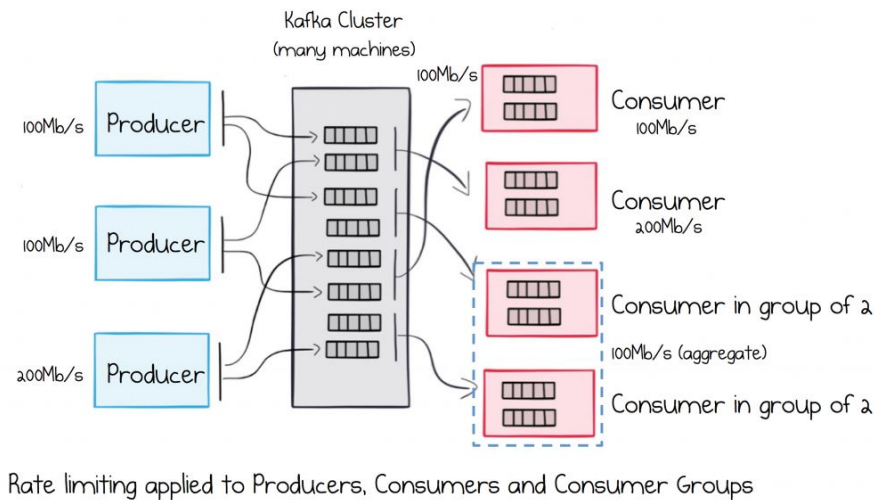
- Application subscribes and consumes messages from brokers in Kafka Cluster
- Consumers know which broker to read from
- Data is read in order within each partition
- Can be of any kind, like real-time consumers, NoSQL consumers etc.
- Messages with the same key arrive at the same consumer

- Supports both Queuing and Publish-Subscribe
- Consumers have to maintain the number of messages consumed



3.22 Consumer Groups

- During consumption of messages from a topic a consumer group can be configured with multiple consumers.
- Each consumer in a consumer group reads data from a different partition
 - ◇ parallelism improves performance
- If there are more consumers than partitions, some consumers will be inactive
 - ◇ If a consumer crashes/stops, the inactive consumer will take over
- Consumers automatically use a GroupCoordinator and a ConsumerCoordinator to assign a consumer to a partition



3.23 Consumer Offsets

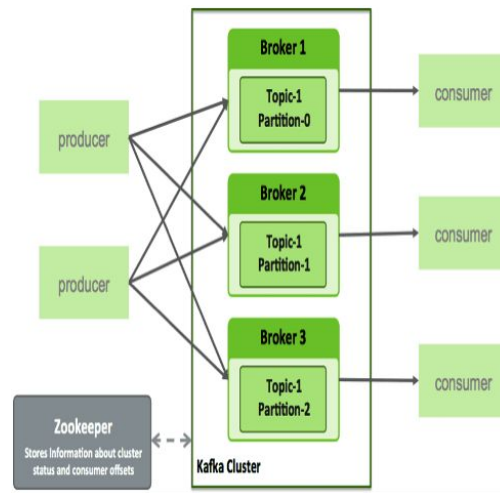
- When Consumer Group is utilized, Kafka stores the offsets at which a consumer group has been reading
- Kafka stores the offsets committed in **__consumer__offsets** predefined Kafka topic
- If a consumer crashes, it will be able to resume where it left off.
- Consumers can choose to commit offsets in of the 3 possible ways:
 - ◇ At most once (Not recommended)
 - commits offsets as soon as a message is received
 - The message will be lost if anything goes wrong while processing the data
 - ◇ At least once (Preferred)
 - commits offsets after the message is processed
 - The message will be read again if anything goes wrong while processing the data
 - ◇ Exactly once
 - Can be achieved for Kafka to Kafka workflows using Kafka Streams API

3.24 Kafka Broker

- Kafka cluster basically is comprised of one or more servers
- Each of the servers in the cluster is called a broker or a “bootstrap server”
- When you connect to any broker, you will be connected to the entire cluster
- Each broker has a unique ID
- Each broker contains certain topic partitions
- Handles hundreds of megabytes of writes from producers and reads from consumers
- Retains all the published messages irrespective of whether it is consumed or not
- Topic is automatically replicated to all brokers in the same cluster

3.25 Kafka Broker and Replication

- Topics should have a replication factor greater than 1
 - ◇ 2 is the minimum for fault-tolerance
- With a replication factor of N, producers and consumers can tolerate up to N-1 brokers being down
- It is recommended to have 3 brokers to allow for one broker to be taken down for maintenance and another broker to be taken down unexpectedly



3.26 Connecting to a Kafka Broker

- All brokers (bootstrap servers) keep track of metadata
- Metadata includes information about all brokers, topics, and partitions.
- When a Kafka client connects to a Kafka broker, the following steps are performed:
 - ◇ Connection + Metadata request
 - ◇ Receive broker list
 - ◇ Connect to the broker

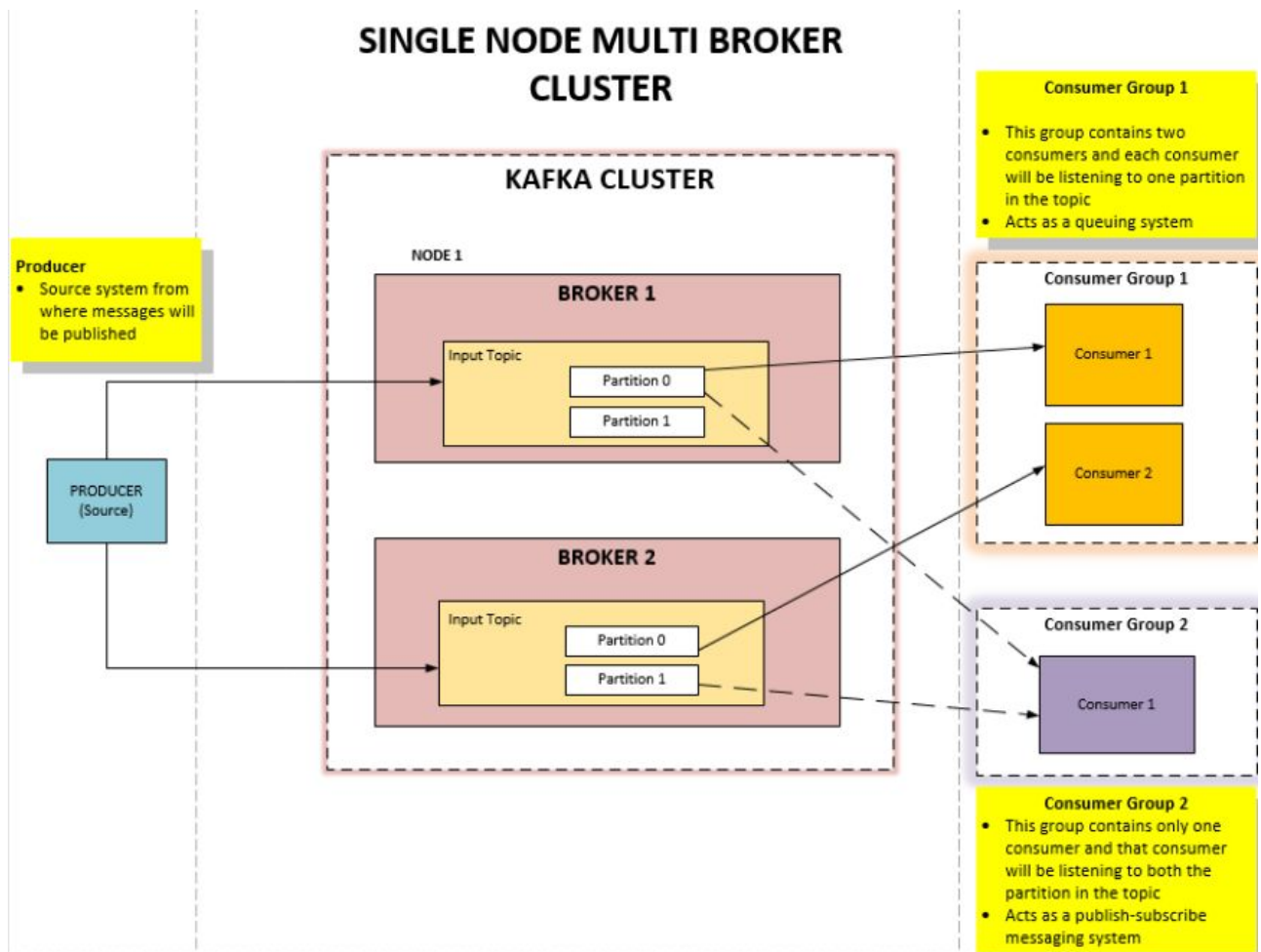
3.27 Kafka Cluster

- A Kafka Cluster is generally fast, highly scalable messaging system
- A publish-subscribe messaging system
- Can be used effectively in place of ActiveMQ, RabbitMQ, Java Messaging System (JMS), and Advanced Messaging Queuing Protocol (AMQP)
- Can be integrated with Hadoop Ecosystem
- Expanding of the cluster can be done with ease
- Effective for applications which involve large-scale message processing

3.28 Why Kafka Cluster?

- Kafka is preferred in place of more traditional brokers like JMS and AMQP
 - ◇ With Kafka, we can easily handle hundreds of thousands of messages in a second, which makes Kafka a high throughput messaging system
 - ◇ The cluster can be expanded with no downtime, making Kafka highly scalable
 - ◇ Messages are replicated, which provides reliability and durability
 - ◇ Fault-tolerant

3.29 Sample Multi-Broker Cluster



3.30 Kafka Broker, Leader, and ISR

- In a multi-broker Kafka cluster, one one broker can be a leader for a given partition
- The leader can receive and serve data for a partition
- The remaining brokers are are called In-Sync Replica (ISR)
- The other brokers synchronize the data
- If a partition leader fails, Kafka choose a new ISR as the new leader
- After the failed leader is brought online, it will become an ISR and try to become the leader again

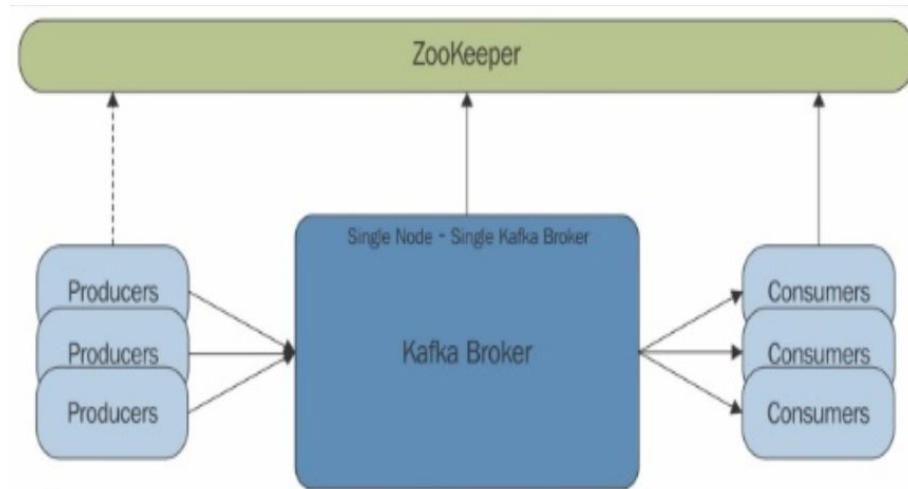
3.31 Overview of ZooKeeper

- An open source Apache project
- Kafka cannot work without Zookeeper
 - ◇ at least 1 Zookeeper is required
- Provides a centralized infrastructure and services that enable synchronization across a cluster

3.32 Overview of Zookeeper (Contd.)

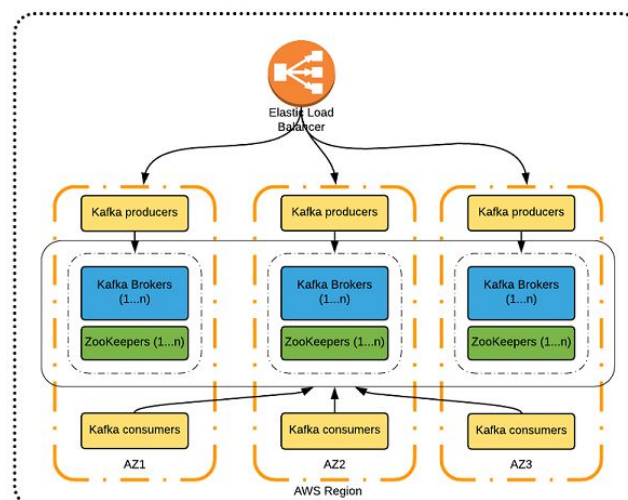
- Helps in performing leader election for partitions, sends notification to Kafka in case of changes, such changes to topics and broker status changes
- Common objects used across the large cluster environments are maintained in Zookeeper
- Objects such as configuration, hierarchical naming space etc. are maintained in Zookeeper
- Zookeeper services are used by large scale applications to coordinate distributed processing across large clusters

3.33 Kafka Cluster & ZooKeeper



3.34 Zookeeper Leader and Followers

- For better scalability & fail-over, Zookeeper operates with an odd number of servers (3, 5, ...)
- When multiple Zookeepers are used, there's one leader and the rest of the servers are followers
- Leader handles write operations and Followers handle read operations



3.35 Summary

- Kafka is a unique distributed publish-subscribe messaging system written in the Scala language with multi-language support and runs on the Java Virtual Machine (JVM).
- Kafka relies on another service named Zookeeper – a distributed coordination system – to function.
- Kafka has high-throughput and is built to scale-out in a distributed model on multiple servers.
- Kafka persists messages on disk and can be used for batched consumption as well as real-time applications.

Appendix 4 - [Appendix] Overview of the Amazon Web Services (AWS)

Objectives

Key objectives of this chapter

- Overview of the Amazon Web Services

4.1 Amazon Web Services

- The Amazon Web Services (AWS) cloud platform offers a variety of infrastructure services that clients can start using with administration costs significantly lower compared with those required to support their own infrastructure
- With AWS, you can take advantage of the following cloud computing benefits:
 - ◇ Just-in-time on-demand infrastructure provisioning
 - ◇ Scalable and elastic computing and storage capabilities
 - ◇ Variable costs over fixed costs
 - ◇ Pay-as-you go (usage-based) costing model
 - ◇ Shorter time to market for your applications
 - ◇ Global-as-local reach
 - The global network of AWS Edge locations consists of 54 points of presence worldwide
 - ◇ Innovation platform supporting efficient development cycles (agility)

Notes:

Efficient development in the cloud is supported, in part, by the ability to clone and easily scale environments, e.g. Dev → QA → Staging → Production

4.2 The History of AWS

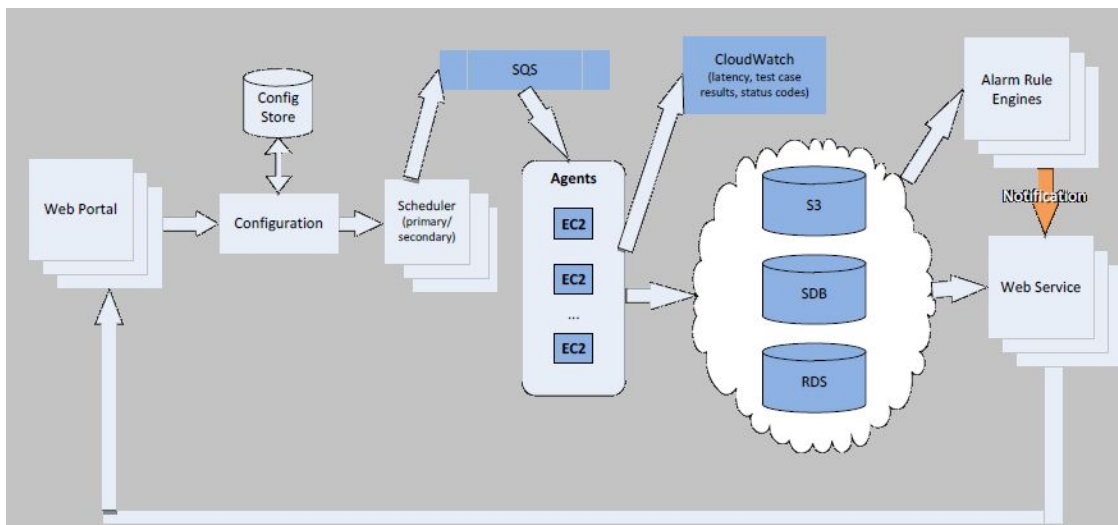
- In 2003, Chris Pinkham and Benjamin Black of Amazon presented a short

internal paper "*describing a vision for Amazon infrastructure that was completely standardized, completely automated, and relied extensively on web services for things like storage*" [<http://blog.b3k.us/2009/01/25/ec2-origins.html>]

- The first public AWS service was Simple Queue Service (SQS) launched in November 2004
- S3 was launched in March 2006
- EC2 followed with a launch in August 2006
- As a cloud platform providing on-line services for clients, AWS was launched in 2006
- In November 2010, all of Amazon.com retail web services were moved to AWS

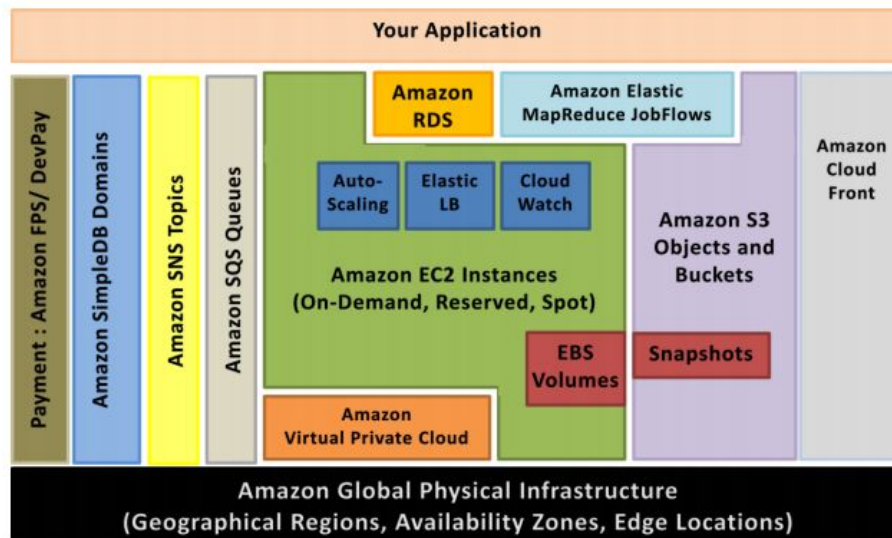
4.3 The Initial Iteration of Moving amazon.com to AWS

- The amazon.com's infrastructure circa 2008 (*amazon.com* web servers were moved to AWS in 2009/2010):



- Source: The "*amazon.com's Journey to the Cloud*" presentation by Jon Jenkins [jjjenkin@amazon.com]

4.4 The AWS (Simplified) Service Stack



Source: Amazon Web Services

4.5 Accessing AWS

- You have the following means of accessing AWS:
 - ◇ The AWS Management Console (Web UI)
 - ◇ The AWS Command-line Interface (AWS CLI)
 - ◇ The AWS Software Development Kits (SDKs)

4.6 Direct Connect

- AWS Direct Connect (ADC) allows you to establish a dedicated network connection from your on-premise resource to AWS
- You get an increased bandwidth throughput at lower costs and a more consistent network experience than Internet-based connections
 - ◇ You have a choice between 1 and 10 Gbps network bandwidth
- You can integrate with ADC to build hybrid environments using on-premise resources and your VPC's and EC2's
- ADC is an alternative to Public Internet-based VPN connections

Notes:

AWS Direct Connect uses the 802.1q VLAN industry standard, which, basically, allows traffic for multiple VLANs on a single trunking interface between two Ethernet switches.

To help you establish network circuits between an AWS Direct Connect location and your data center, AWS recommends contacting the APN Technology and Consulting Partners (formerly called AWS Direct Connect Solution Providers), <https://aws.amazon.com/directconnect/partners/>

4.7 Shared Responsibility Model

- While AWS is responsible for providing cloud-grade infrastructure with the perimeter security and intrusion detection in place, you are responsible for security of your accounts, networks, and applications, including
 - ◇ User access control
 - ◇ User roles
 - ◇ Application passwords
 - ◇ Instance OS patching
 - ◇ Network configuration (public / private)

4.8 Trusted Advisor

- AWS on-line expert service that can analyze your AWS environment and offers recommendations that can help you reduce cost, tighten security, increase performance, improve fault tolerance, and optimally provision resources following AWS best practices
- There are two tiers of Trusted Advisor service:
 - ◇ **Free** - available to all AWS customers
 - This type performs only core checks
 - ◇ **Business or Enterprise** - based on support plans

4.9 The AWS Distributed Architecture

- The AWS cloud platform is deployed across a number of geographical

regions (20 as of 2019)

- Each region includes two or more "Availability Zones" (AZ), which are isolated from each other data centers (60 AZ's as of 2019)
 - ◇ Some Amazon services can operate across AZ's (e.g. S3)
 - ◇ Other services require you to set up and configure replication across AZ's to achieve service resilience against data center outages

Notes:

The data you store in a region stays in the region; for high availability, you may want to move or replicate data across regions, and when doing so, you should be aware of local jurisdiction and compliance requirements. For example, AWS created a dedicated GovCloud US region designed to serve the needs of the US state and federal government agencies.

4.10 AWS Services

- As of 2019, AWS has around 100 services, including compute, security and identity, storage, networking, relational databases and NoSQL solutions, analytics, the IoT, etc.
- AWS uses a number of ODM (original design manufacturer) products designed and developed per Amazon's specs
- Not all services are available in all regions
- In subsequent slides, we will quickly review some of the commonly used services

4.11 Managed vs Unmanaged Amazon Services

- Depending on the type of an AWS service, its scalability and high availability properties are handled either by
 - ◇ AWS (the managed services) or
 - ◇ By you (the unmanaged services)

4.12 Amazon Resource Name (ARN)

- ARN is used to uniquely identify AWS resources.
- Examples of the variety of resources need to be unambiguously across all of AWS
 - ◇ IAM policies
 - ◇ Amazon Relational Database Service (Amazon RDS) tags
 - ◇ API calls
- Examples of ARN
 - ◇ <!-- IAM user name -->
arn:aws:iam::123456789012:user/David
 - ◇ <!-- Amazon RDS instance used for tagging -->
arn:aws:rds:eu-west-1:123456789012:db:mysql-db
 - ◇ <!-- Object in an Amazon S3 bucket -->
arn:aws:s3:::my_corporate_bucket/exampleobject.png

4.13 Compute and Networking Services

- The main services in this category include:
 - ◇ Elastic Compute Cloud (EC2)
 - ◇ AWS Lambda
 - ◇ ECS ([Docker] Elastic Container Service)
 - ◇ Auto Scaling
 - ◇ Elastic Load Balancing (ELB)
 - ◇ Virtual Private Cloud (VPC)
 - ◇ Amazon Route53
 - ◇ Elastic Beanstalk

4.14 Elastic Compute Cloud (EC2)

- Amazon EC2 provides web-scale compute capability
- It is an example of an unmanaged service
- EC2 instances (deployed as virtual machines) are bootable from an Amazon Machine Image (AMI) of the operating system of your choice, which may optionally have pre-installed software and associated configuration files
 - ◇ Currently, EC2 offers a selection of 500+ AMI's of popular Open Source and commercial software, available from the Amazon's marketplace
 - ◇ You can also create your own AMI
- You have a wide range of supported OS images to choose from:
 - ◇ Various Linux distros
 - ◇ Windows
- Security is managed through security groups (virtual firewalls)

4.15 AWS Lambda

- AWS Lambda lets you run your code (called the *Lambda function*) written in a number of supported languages in Serverless Computing (a.k.a. NoOps) environments
- You pay only for the actual time your Lambda function runs
- Lambda functions are used
 - ◇ As triggers responding to changes (events) in other AWS services (sources of events), such as an S3 update
 - ◇ As code invoked directly from your apps in the form of an in-app activity, e.g. fetching data from Amazon DynamoDB

Notes:

As of 2017, Lambda gives you a choice of the following languages:

- Node.js (JavaScript)

- Python 2.7
- Java 8
 - Lambda provides the Amazon Linux build of openjdk 1.8
- C#
 - Distributed as a NuGet package with the “dotnetcore1.0” runtime parameter

4.16 Auto Scaling

- Auto Scaling allows automatic elastic scaling of your capacity up and down by configuring resource utilization thresholds and usage conditions
- It can increase the number of your Amazon EC2 instances during demand peaks and decrease capacity during demand valleys to optimize your EC2 fleet utilization costs
- Metrics used by Auto Scaling conditions that trigger scaling activities are collected by the CloudWatch service

Notes:

Auto Scaling uses a number of plans to control how auto scaling happens:

- Constant pool of instances
 - ✓ If an instance fails or pulled out from the pool, a new instance is automatically provisioned
- Manual scaling
 - ✓ Defined in your Auto Scaling group
- Scheduled scaling
 - ✓ Great for anticipated or known in advance load spikes

Auto Scaling has the following components:

- A Launch configuration
 - ✓ Based on an EC2 profile template (AMI, instance type, security group, keys, etc.)
- An Auto Scaling group, which has the following configuration parameters:
 - ✓ The minimum capacity
 - ✓ The desired capacity
 - ✓ The maximum capacity

- A Scaling policy (optional)

4.17 Elastic Load Balancing (ELB)

- Elastic Load Balancing (ELB) automatically distributes incoming network traffic across a cluster of your Amazon EC2 instances
- ELB helps you achieve fault tolerance and high availability (HA) in your app
 - ◇ EC2 instances may run in multiple AZ's for HA
- Tightly integrated with the Auto Scaling service
- There are two types of ELB:
 - ◇ The Classic ELB (the simple type) that routes traffic based on either application- or network-level information
 - ◇ The Application ELB (the advance type) that can route traffic based on the content of the inbound request
- ELB is a highly available AWS-managed service that supports both Internet-facing and internal application-facing deployments
 - ◇ Internal application-facing deployments help create scalable multi-tier applications

Notes:

ELB supports routing and load balancing for the following protocols:

- HTTP
- HTTPS
- TCP
- SSL

ELB offers a number of configuration settings, including:

- Idle connection timeout
 - ✓ Default is one minute
 - ✓ If a client request cannot be completed within this period, the connection is forced to close
- Connection draining (quiescing) to deregistering instances

- Proxy protocol for TCP or SSL
 - ✓ Adds the original request's IP address and some other information as a header to the forwarded request
- Cookie-based sticky sessions (aka session affinity)
 - ✓ ELB offers this capability via the AWSELB cookie
 - ✓ You can also use your own application session cookie
- EC2 instances health-check
 - ✓ Based on a ping or a web page hit

AWS recommends accessing a LB by its DNS name, not by its IP address.

ELB deployed in VPC support only IPv4 addresses.

4.18 Virtual Private Cloud (VPC)

- Allows you to define a private virtual network inside the AWS Cloud with your own IP address range, subnets, route tables, and network gateways
- VPCs have one-to-one relationship with regions
- Enables you to build hybrid clouds by extending your corporate network topology into a private cloud inside the Amazon Cloud
- A VPC can include resources from different AZ's within the region
- You can have more than one VPC in a region
- AWS VPC uses IPSec tunnel mode for point-to-point secure channel between your data center and the AWS Cloud

4.19 Route53 Domain Name System

- A highly scalable DNS service which you can use to manage your DNS records for all your domains to help resolve domain names to AWS-hosted IP addresses of your applications
- It also provides ways to buy new domains and transfer in existing domains
- Route 53 can also act as the registrar for your existing domains that you bought from other registrars

- This service can also be used to monitor the health and performance of your web applications and redirect traffic accordingly

4.20 Elastic Beanstalk

- Amazon Elastic Beanstalk service enables fast development cycles for your projects by offering on-demand provisioning of the needed servers and automatic deployment of your applications written in Java, PHP, Ruby, Node.js, .NET, Python, and Go
- Can be viewed as a PaaS capability
- The Beanstalk environment gives you an option to leverage the load balancing and auto-scaling services for your applications

Notes:

For example, you can set up the Tomcat web server Beanstalk environment, then upload a Java Web ARchive (WAR) file and start using it as a Java Web application publicly available to your clients.

4.21 Security and Identity Services

- Identity and Access Management (IAM)
- AWS Directory Service
- AWS Certificate Manager
- AWS Key Management Service (KMS)

4.22 Identity and Access Management (IAM)

- IAM is the AWS user management, authentication and authorization service
- IAM is natively integrated into all the AWS Services
- It helps managing users and their permissions within your AWS Account
- IAM uses AWS Key Management Service (KMS) to create and control the user encryption keys

4.23 AWS Directory Service

- Allows organization to set up and run Microsoft Active Directory (AD) on the AWS Cloud
- Facilitates integration with on-premise AD's
- Provides single sign-on capabilities

4.24 AWS Certificate Manager

- Supports SSL/TLS certificate provisioning and renewal
- Offers a streamlined process of obtaining and installing certificates
 - ◇ e.g. on ELB

4.25 AWS Key Management Service (KMS)

- Deals with creation and management of client encryption keys
- Uses Hardware Security Modules (HSMs) for safekeeping the keys
- Tightly integrates with other AWS services that require data encryption

4.26 Storage and Content Delivery

- Elastic Block Storage (EBS)
- Simple Storage Service (S3)
- Glacier
- CloudFront

4.27 Elastic Block Storage (EBS)

- The Elastic Block Store (EBS) is a network-attached persistent storage volume that you can attach to EC2 instances
- An EBS volume is seen by your EC2 instance as a local hard drive
- **Note:** You will pay for this storage even if your instance to which the

volume is attached is stopped

- Launched EC2 instances also receive free local instance store of limited capacity which is ephemeral and gets recycled (totally erased) when the server is stopped
 - ◇ You should use the local instance store only for temporary data that don't require durable persistence

4.28 Simple Storage Service (S3)

- Amazon Simple Storage Service (S3) is a secure, durable, highly-scalable object (file) storage
- S3 was Amazon's first publicly available web service launched in the United States in March 2006 and in Europe in November 2007
- Amazon reported that in April 2013, S3 stored over 2 trillion objects
- You only pay for the actual storage you use
- S3 can be used as a stand-alone service or integrated with other AWS services
- For redundancy, client's data is stored across a number of locations and multiple devices in each location
- S3 supports the RESTful end-point as a programmatic interface for uploading and downloading objects

4.29 Glacier

- Amazon Glacier is a an archival service
- Closely integrated with S3 to provide a low-cost durable solution for data and back-ups
- Glacier is optimized for infrequent user access patterns that can tolerate high latencies in data retrieval

4.30 CloudFront Content Delivery Service

- A global accelerated content delivery service (CDS) for static and/or streaming content, e.g. S3 objects
- CloudFront organizes your content into distributions based on the location(s) of the original version of your files
 - ◇ Each distribution is assigned a unique domain name (e.g. *QXYZ.cloudfront.net*) that is referenced through the global network of AWS edge locations
 - ◇ When your clients request an object using this domain name, they are automatically routed to the nearest geographical locations (called edges)
- S3 objects (images, documents, etc.) can be replicated and cached using CloudFront in a number of different geographical locations

4.31 Database Services

- Relational Database Service (RDS)
- DynamoDB
- ElastiCache
- Redshift

4.32 Relational Database Service (RDS)

- Backed-up by full-featured instances of a database engine of your choice
- Databases currently supported are:
 - ◇ PostgreSQL,
 - ◇ Oracle,
 - ◇ Microsoft SQL Server (various editions, ranging from Express to Enterprise),
 - ◇ MariaDB,
 - ◇ MySQL Community Edition,

- ◊ Amazon Aurora (MySQL-compatible, cost-effective enterprise-class database)
- RDS is a managed service, where database administration tasks such as backups, patch management, etc., are taken care of by AWS support
- Suitable for complex transactions or SQL queries with up to 30K IOPS (15K reads/second + 15K writes/second) on a single node/shard

Notes:

Amazon RDS supports Multi-AZ deployments, which enables architects to create highly available and fault-tolerant database applications. In the Multi-AZ architecture, you deploy your database in two AZ's -- you place the Master database in one AZ and a secondary database instance in another. AWS automatically replicates the data from the Master to the secondary node. AWS generates a unique DNS name that is resolved to a specific IP address. The database clients use that DNS name when connecting to the database.

4.33 DynamoDB

- DynamoDB is Amazon's flagship NoSQL database service
- Provides consistent, single-digit millisecond latency at any scale
- Offers flexible key-value and document data store models
- Uses SSDs
- Suitable for applications with ~100K IOPS with sharding to support high throughput

4.34 Amazon ElastiCache

- ElastiCache is a scalable (fully managed) in-memory cache in the AWS Cloud
 - ◊ Supports up to 3.55 TiB of in-memory data
- Provides sub-millisecond latency for object retrieval
- As of 2017, ElastiCache is compatible with Redis and Memcached

4.35 Redshift

- Amazon Redshift is a fully managed, petabyte-scale data warehouse service in the AWS cloud
- Provides a standard SQL-based interface to your data and access to a number of BI tools
- Employs columnar storage
- Supports parallelized query execution across a cluster of nodes

4.36 Messaging Services

- Simple Queue Service (SQS)
- Simple Notifications Service (SNS)
- Simple Email Service (SES)
- Amazon MQ
- **Note:** These are PaaS-type of capabilities

4.37 Simple Queue Service (SQS)

- A fully managed cloud-grade distributed queue-based messaging platform
- Using SQS, you can send, receive, and retain messages in a fully decoupled asynchronous fashion without requiring message consumers to be always up and running
- You have a choice between two message delivery options:
 - ◇ **SQS standard queues**
 - Use it for fire-hose throughput, best-effort (not guaranteed) message ordering, and at-least-once delivery guarantee
 - ◇ **SQS FIFO queues**
 - Guarantee that messages are processed exactly once, in the exact order that they are sent,
 - Less overall throughput compared with SQS standard queues

Notes:

The Amazon SQS Messaging API for Java supports sending and receiving messages to a queue (the JMS point-to-point model), as specified in the JMS 1.1 specification. The SQS Java Messaging Library uses a JMS interface to Amazon SQS that you can use in your JMS-enabled Java applications with minimum changes.

SQS offers clients Delay Queues for delayed message delivery. The delay (during which a message is invisible for consumers) may be configured for as long as 15 minutes. Hiding messages on a queue may help with synchronization of application sub-systems.

A single-threaded SQS client can fetch within a second no more than about 50 messages. To increase the throughput of your application, add multiple clients, or create a multi-threaded SQS client.

4.38 Simple Notifications Service (SNS)

- Simple Notifications Service (SNS) is a fully managed cloud-grade push messaging platform
- SNS allows you build applications based on the pub-sub (topic-based) protocol
- You can push individual or fan-out messages to the interested subscribers, such as mobile devices, email recipients, SMS clients, etc.
- Integrates with HTTP end-points, AWS Lambda functions, and SQS

4.39 Simple Email Service (SES)

- Amazon Simple Email Service enables you to send and receive email using a scalable email platform
- Tightly integrates with S3, SNS, and Lambda
- Verifies email addresses and domains
- Maintains email metrics, such as the number of emails sent and the email bounce rates

4.40 AWS Monitoring with CloudWatch

- Amazon CloudWatch is a monitoring service for AWS-deployed cloud resources

- CloudWatch is capable of monitoring and collecting some predefined resource metrics, or custom-defined resource artifacts (e.g. log files and alarms)
 - ◇ For example, CloudWatch will provide monitoring of EC2 instances with respect of operational performance, including such metrics as CPU utilization, I/O operations, network traffic (in/out), and response latencies. It can also be configured to monitor HTTP status codes in Apache logs, etc.
- For collecting custom log data, you need to install and configure CloudWatch agents that will be sending your logs to the CloudWatch Logs service and then create your specific metric filter there

4.41 Other Services Example

- Elastic MapReduce (EMR) - an analytics service
 - ◇ A hosted Hadoop framework for running MapReduce jobs
 - ◇ You have a choice of Hadoop-related applications that can be installed alongside with your Hadoop cluster:
 - ◇ Hive, Pig, Hue, HBase, Impala, etc.
 - ◇ EMR is tightly integrated with EC2 and S3 services
- Amazon Kinesis – real-time streaming service
- Machine Learning services
- Internet of Things

4.42 Summary

- The AWS Cloud offers a variety of services that you can leverage when building your solutions in the cloud