

**Cloud Native
Microservices with
Spring and Kubernetes**

*Design and Build Modern Cloud
Native Applications Using Spring and Kubernetes*

Rajiv Srivastava



www.bpbonline.com

FIRST EDITION 2021

Copyright © BPB Publications, India

ISBN: 978-93-90684-311

All Rights Reserved. No part of this publication may be reproduced, distributed or transmitted in any form or by any means or stored in a database or retrieval system, without the prior written permission of the publisher with the exception to the program listings which may be entered, stored and executed in a computer system, but they can not be reproduced by the means of publication, photocopy, recording, or by any electronic and mechanical means.

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The information contained in this book is true to correct and the best of author's and publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but publisher cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners but BPB Publications cannot guarantee the accuracy of this information.

Distributors:

BPB PUBLICATIONS

20, Ansari Road, Darya Ganj

New Delhi-110002

Ph: 23254990/23254991

MICRO MEDIA

Shop No. 5, Mahendra Chambers,

150 DN Rd. Next to Capital Cinema,

V.T. (C.S.T.) Station, MUMBAI-400 001

Ph: 22078296/22078297

DECCAN AGENCIES

4-3-329, Bank Street,

Hyderabad-500195

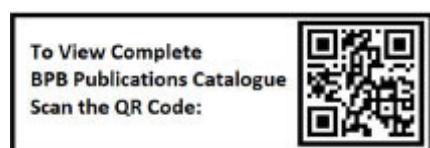
Ph: 24756967/24756400

BPB BOOK CENTRE

376 Old Lajpat Rai Market,

Delhi-110006

Ph: 23861747



Published by Manish Jain for BPB Publications, 20 Ansari Road, Darya Ganj, New Delhi-110002 and Printed by him at Repro India Ltd, Mumbai

www.bpbonline.com

Dedicated to

Mr K P Srivastava, my father, who has been my inspiration and an author of his autobiography, Mrs Durgawati, my mother, Mrs Meera Saxena, my teacher and mentor, Mrs Shweta, my spouse, who has supported me to write this book, and my loving daughter and little teacher Shanvi (Manya), who has helped me to edit this book!

About the Author

Rajiv Srivastava is a cloud native solution architect with application and cloud infra specialist with 17+ years of work experience in software development and architectural design.

He is the founder of cloudificationzone.com, which is a cloud native modern application tech blog site.

He is also a blogger, author, and passionate technologist. He holds strong insights into the latest technology stack, including modern cloud architectures, modern applications, microservices, and open sources.

He is an expert in enterprise, solution and cloud architecture, application modernization/migration, architecture design, Java language, Spring framework, cloud migration, Kubernetes platform, VMware Tanzu platform, agile development, event-driven design, reactive, NoSQL modern databases, search engine, API Gateway, Service Mesh, CI/CD, containerization, GCP, AWS, VMware, serverless, etc.

He is a AWS certified Solution Architect (Associate), Certified Kubernetes Application Developer (CKAD) and Sun (Oracle) Certified Java Professional (SCJP).

About the Reviewer

Rohit Kelapure is an app modernization scale specialist at overlaying the direct enterprise sales organization and working closely with the largest Google Cloud customers to help them shape their cloud strategy and enable their digital transformation. Rohit has led the VMware App Modernization Practice in engineering, delivery, training, tooling, scoping, and marketing. Rohit has delivered multiple enterprise engagements, including ones featured in the Wall Street Journal driving digital transformation, streaming and microservices architectures, modern application design and legacy-to-modern software transformation techniques and processes. Rohit is an expert in migrating applications to the cloud and decomposing monoliths. Rohit actively blogs on cloud foundry, kubernetes, decomposing monoliths, and app modernization strategies. His webinars include topics as diverse as middleware migration, mainframe migration, and tools and recipes for replatforming monolithic apps.

His LinkedIn link: <https://www.linkedin.com/in/rohitkelapure/>

Acknowledgement

There are a few important people I want to thank for the continued and ongoing support they have given me during the writing of this book. First and foremost, I would like to thank my wife, Shweta and daughter, Shanvi, for putting up with me while I was spending many weekends and evenings writing this book. I could never have completed this book without their moral support.

I would like to thank my ex-colleague and friend Rohit Kelapure, who has reviewed my book and shared his valuable technical suggestions. My gratitude goes to him for providing valuable insights into some of the new modern cloud native features and practices. He works with Google. He is a blogger and founder of cloud.rohitkelapure.com.

I would like to thank Nrip Jain, Head Business Development Group, BPB Publications for giving me this opportunity to write my first book for them!

Last but not the least, I would like to thank editors and reviewers at BPB Publications for their continuous support while writing and editing this book.

I am really thankful to all my friends, family, acquaintances, colleagues, and ex-colleagues for their continuous inspiration and support.

Preface

The Microservice architecture (MSA) is a variant of the service-oriented architecture (SOA) structural style which arranges an application as a collection of loosely coupled services. In the microservices architecture, services are fine-grained and the protocols are lightweight. Microservice is an independent module which can be deployed separately and easily maintained with no dependency on other microservice. This book will cover detail microservices architecture overview, batch microservices, reactive and event-driven microservices and various microservices design patterns to build modern cloud native applications. It will also cover microservices related business use cases and best practices.

The Spring framework is most popular and best suited for building and managing microservices. This book will cover fundamentals and detailed overview of cloud native microservices architecture, hands on microservices development with the Spring framework, and Kubernetes using source code examples.

Kubernetes is a well-established open-source container-orchestration platform for automating computer application deployment, scaling, and container management. It provides a container centric platform for microservices to automate auto-scaling, resilience, and high availability. This book will cover how to build Docker images run on Kubernetes containers.

This book is for the Spring developers, Microservice developers, Kubernetes developers, Cloud engineers, DevOps consultants, Application, Solution, enterprise and cloud architects, who are familiar with application development, Docker, containers and Kubernetes concepts.

This book will cover all fundamental and advanced level concepts to make use of cloud native modern microservices applications on Kubernetes containers.

The primary goal of this book is to provide information and skills that are necessary to build, design microservices apps with the Spring framework, and deploy on Kubernetes containers with real use cases examples. There are 12 chapters in this book. You will learn the following topics:

Chapter 1: Overview of Cloud Native microservices, introduces cloud native modern applications, cloud first overview, benefits, types of clouds, classification, and the need for cloud native modern applications. It will cover detailed microservices (MSA) overview, characteristics, motivations, benefits, best practices, architecture principles, challenges and solutions, application modernization spectrum, twelve-factor apps, and beyond twelve-factor apps.

Chapter 2: Microservice design patterns, introduces various microservices design patterns with use cases, advantages, and disadvantages.

[Chapter 3:](#) API first approach, discusses fundamentals of the API first approach. It discusses details of the REST overview, API model, best practices, design principles, components, security, communication protocols, and how to document dynamically with OpenAPI Swagger. It discusses API design planning, specifications, API management tools, and testing API with SwaggerHub inspector and PostMan REST client.

[Chapter 4:](#) Build microservices using the Spring Framework, is a key chapter of Spring Boot and Spring Cloud components with hands-on lab exercises. It will cover steps to build microservice using the REST API framework. It covers the Spring Cloud config server and resiliency of microservices practical aspects.

[Chapter 5:](#) Batch microservices, introduces batch microservices, use cases, Spring Cloud Task, and Spring Batch. It discusses hands-on lab exercises using Spring Cloud Data Flow (SCDF) and Kafka. It also discusses a few Spring batch practices, auto-scaling techniques, batch orchestration, and compositions methods for sequential or parallel batch processing. Last but not the least; it talks about alerts and monitoring of Spring Cloud Task and Spring Batch.

[Chapter 6:](#) Build reactive and event-driven microservices, describes building of reactive microservices, non-blocking synchronous APIs, and event-driven asynchronous microservices. It covers steps to develop sample reactive microservices with Spring's project Reactor, Spring WebFlux, and event-driven asynchronous microservices. It discusses Spring Cloud Stream, Zookeeper, SpringBoot, and overview of Kafka. It also covers hands-on lab exercises of event-

driven asynchronous microservices using Spring Cloud Stream and Kafka.

Chapter 7: API gateway, security, and distributed caching with Redis, introduces the API Gateway overview, features, advantages, and best practices. It covers hands-on lab exercises to expose REST APIs of microservices externally with the Spring Cloud Gateway. It covers distributed caching overview and hands-on lab exercises using Redis. It discusses API gateway rate limiting and Implementation of API gateway rate limiting with Redis and Spring Boot. Last but not the least, it covers best practices of API Security. Implementation of SSO using Spring Cloud Gateway, Spring Security, Oauth2, Keycloak, OpenId, and JWT tokens.

Chapter 8: Microservices testing and API mocking, describes important aspects of microservices testing practices, challenges, benefits, testing strategy, testing pyramid, and different types of microservices testing. It covers implementation of the integration testing framework using Behavioral Driven Development (BDD) with hands-on code examples. It also discusses microservices testing tools and best practices of microservices testing. It covers the role of testing in the microservices CI/CD pipeline. Last but not the least; it talks about API mocking and hands-on lab implementation with the WireMock framework.

Chapter 9: Microservices observability, covers detail observability and monitoring overview and techniques of microservices with the Spring actuator, micrometer health APIs, and Wavefront APM. It covers application logging overview, best practices, simple logging, and log aggregation of distributed microservices with

implementation using Elasticsearch, Fluentd, and Kibana (EFK) on the Kubernetes container. It discusses the need of APM performance and telemetry monitoring tools for distributed microservices and how to trace multiple microservices in a distributed environment. It also covers hands-on lab implementation of monitoring microservices with Prometheus and Grafana.

Chapter 10: Containers and Kubernetes overview and architecture, is a key chapter which introduces containers, docker, docker engine containerization, Buildpacks, components of docker files, build docker files, run docker files, and inspect docker images. It covers docker image registry and how to persist docker images in image container registries. It covers an overview of Kubernetes, need, and architecture. Last but not the least; it covers a detailed introduction of Kubernetes resources.

Chapter 11: Run microservices on Kubernetes, discusses practical aspects of Kubernetes, installation, and configuration with monitoring and visualization tools with Octant and Proxy. It discusses how to create and manage Kubernetes clusters in detail. It discusses hands-on exercises of creating docker images of Java microservices, pushing it to the Docker hub container image registry, and deploying to Kubernetes clusters. It covers hands-on lab examples of exposing API endpoints of microservices outside the Kubernetes cluster by using the Nginx ingress controller. Last but not the least; it covers various popular and useful Kubernetes application deployment and configuration of management tools.

[Chapter 12:](#) Service Mesh and Kubernetes alternatives of Spring Cloud, covers a detailed overview and benefits of GitOps and Service Mesh. It covers the Istio Service Mesh architecture and deployment of microservices on Kubernetes with Argo CD. Last but not the least; it discusses various Kubernetes alternatives of Spring Cloud projects and popular cloud buzzwords!

**Downloading the code
bundle and coloured images:**

Please follow the link to download the ***Code Bundle*** and the ***Coloured Images*** of the book:

<https://rebrand.ly/bf2965>

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Did you know that BPB offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.bpbonline.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at business@bpbonline.com for more details.

At you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.

BPB is searching for authors like you

If you're interested in becoming an author for BPB, please visit www.bpbonline.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

The code bundle for the book is also hosted on GitHub at In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at Check them out!

PIRACY

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at business@bpbonline.com with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit

REVIEWS

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit

Table of Contents

1. Overview of Cloud Native Microservices

Structure

Objective

Fundamentals of cloud computing

Introducing cloud

Cloud principles

Business benefits of cloud

Types of cloud

Private/on-prem cloud

Public cloud

Hybrid clouds

Cloud classifications

Infrastructure as a Service (IaaS).

Container as a Service (CaaS).

Platform as a Service (PaaS).

Function as a Service (FaaS) or serverless architecture

Software as a Service (SaaS).

Database as a Service (DaaS).

Need for cloud native modern applications

Overview of cloud native modern application

Introduction to modern application

Cloud native overview

Microservices architecture (MSA).

Microservices characteristics

Microservices motivations and benefits

Microservices architecture principles

Microservices challenges and solutions

Beyond the twelve-factor apps

Application modernization spectrum

0-1 factors cloud journey

1-4 factors

5-9 factors

10-15 factors

Conclusion

Points to Remember

Key terms

References

2. Microservice Design Patterns

Structure

Objectives

Introduction to software design patterns

Overview of Microservice design patterns

Application architecture patterns

Monolithic architecture

Microservices architecture

Microservices communication

Synchronous communication

Asynchronous communication

Decomposition of microservices

Domain Driven Design (DDD).

Introducing bounded context

Strangler pattern

Integration patterns

Chain of responsibility

Aggregator and branch design patterns

API gateway design pattern

Micro frontends (UI composition) pattern

Database patterns
Database per service
Shared database
Command Query Responsibility Segregation (CQRS).
SAGA design patterns
Orchestrating microservices
Choreography microservices
Hybrid communication
Final conclusions
Event sourcing and an event-driven architecture
Introducing an event
Event sourcing with SAGA
Observability and monitoring patterns
Distributed tracing
Log aggregation
Health check API
Application metrics
Service discovery design pattern
Client-side discovery
Server-side discovery
Circuit breaker design pattern
External configuration
API token security
Integration testing
Conclusion
Points to Remember
Key terms
References

3. API First Approach

Structure

Objective

Introducing API

API first approach overview

Current challenges with API design

Introducing API first approach

Need of API first approach

REST overview

Introducing REST and RESTful API

REST and HATEOAS support

Hypertext Application Language (HAL).

API design principles

Benefits of API first approach

API components

API best practices

API security

Exposing API on communication protocols

API documentation and specifications

API documentation

API specifications

Planning API design

API specifications

OpenAPI Specifications (OAS).

RESTful API Modelling Language (RAML).

OAS versus RAML

API management tools

SwaggerHub

Other API management tools

Designing and generating API docs

Code First with OpenAPI 3.0 and Spring REST API

Prerequisite

Setting up Spring doc OpenAPIwith Swagger UI

[Generating API doc with JSR-303 validation](#)

[Generating API doc with @Operation and @ApiResponses](#)

[API First with SwaggerHub](#)

[Introducing Swagger](#)

[SwaggerHub UI overview](#)

[Testing API with SwaggerHub inspector](#)

[Conclusion](#)

[Points to Remember](#)

[Key terms](#)

[References](#)

[4. Build Microservices with Spring Framework](#)

[Structure](#)

[Objective](#)

[Introduction to Spring Boot](#)

[Spring framework use cases](#)

[Introduction to Spring Cloud](#)

[Spring Cloud libraries](#)

[Creating a Spring Boot microservice project using Spring.io](#)

[Introduction to Spring Cloud config server](#)

[Prerequisites](#)

[Building Spring Cloud config server](#)

[Building Spring Cloud config client](#)

[Fault tolerance using the Spring Cloud Circuit Breaker](#)

[Introduction to Resilience4j](#)

[Implementing Resilience4j circuit breaker](#)

[Prerequisites](#)

[Conclusion](#)

[Points to Remember](#)

[Key terms](#)

[References](#)

5. Batch Microservices

Structure

Objective

Introduction to batch microservices

Introduction to Spring Cloud Task

Spring Cloud Task features

Getting started with Spring Cloud Task

Prerequisite

Introduction to Spring Batch

Getting started with the Spring Batch application

Prerequisites

Introducing to Spring Cloud Data Flow (SCDF).

SCDF objectives

SCDF architecture

SCDF modules

Data flow server

Skipper server

Database

Security

Stream processing

Batch processing

Monitoring

CLI and API library

Installing SCDF

Getting started with SCDF

Prerequisites

Introduction to Spring Cloud Stream

Comparing Spring Cloud Task and Spring Batch

Best practices for Spring Batch

Spring Batch auto-scaling

[Spring Batch orchestration and composition](#)

[Active-active operation for high availability \(HA\) between data centers](#)

[Alerts and monitoring of Spring Cloud Task and Spring Batch Conclusion](#)

[Points to Remember](#)

[Key terms](#)

[References](#)

[6. Build Reactive and Event Driven Microservices](#)

[Structure](#)

[Objective](#)

[Reactive programming overview](#)

[Advantages of a reactive programming model](#)

[Spring WebFlux and Spring MVC comparison](#)

[Non-blocking synchronous API](#)

[Challenges with traditional REST API](#)

[Reactive Relational Database Connectivity \(R2DBC\)](#)

[Developing reactive microservices using Spring Webflux](#)

[Prerequisite](#)

[Event-driven asynchronous microservices](#)

[Developing event-driven asynchronous microservices using Spring Cloud Stream and Kafka](#)

[Prerequisite](#)

[Creating producer microservice](#)

[Creating Consumer Microservice](#)

[Test event driven microservice](#)

[Conclusion](#)

[Points to Remember](#)

[Key terms](#)

[References](#)

7. The API Gateway, Security, and Distributed Caching with Redis

Structure

Objectives

Overview of the API gateway.

API gateway features and advantages

Best practices of the API gateway design

Overview of the Spring Cloud Gateway.

Implementing Spring API Gateway.

Prerequisite

Distributed caching overview

API caching with Redis distributed caching

Redis cluster architecture for high availability (HA).

Implementing Redis distributed caching with the Spring Cloud Gateway.

Prerequisite

API gateway rate limiting

Implementing the API gateway rate limiting with Redis and Spring Boot

Prerequisite

Best practices of API security.

API security with SSO using Spring Security, OAuth2, Keycloak, OpenID, and JWT tokens

Prerequisite

Conclusion

Points to Remember

Key terms

References

8. Microservices Testing and API Mocking

Structure

Objectives

Microservice testing challenges

Microservice testing benefits

Black box versus white box testing

Manual versus automation testing

Microservice testing strategy and testing pyramid

Unit testing

Component testing

Contract testing

Integration testing

End-to-End (E2E) testing

Microservice CI/CD DevOps pipeline on Kubernetes

Best practices of microservices testing

Microservices testing types – a quick reference

Functional testing

Non-functional testing

Need for microservices integration testing

Testing microservices integration using Behavioral Driven Development (BDD).

Given-When-Then test strategy.

Given

When

Then

Additional And, But keywords

Advantages of BDD

BDD with Cucumber: pros and cons

Implementing integration testing with BDD and Cucumber

Prerequisite

Unit testing versus TDD versus BDD

Testing microservices tools

REST API mocking with WireMock

Need of WireMock

WireMock scope

Mocking existing active APIs (through WireMock recording and playback for new API).

Mocking the new API when the API is not available

Assumptions and limitations

Conclusion

Points to Remember

Key terms

References

9. Microservices Observability

Structure

Objectives

Application logging overview

Logging levels

Best practices of logging and tracing

Simple logging using Spring Boot

Log aggregation of microservices

Log aggregation using EFK on the Kubernetes container

Prerequisite

Need for monitoring distributed microservices

API health check using the Spring Micrometer

Monitoring apps and infra with the Wavefront APM tool

Microservices debugging using tracing and telemetry.

Distributed tracing with Spring Cloud Sleuth

Prerequisite

Microservices monitoring

Monitoring microservices with Prometheus and Grafana

Prerequisite

Conclusion

Points to Remember

Key terms

References

10. Containers and Kubernetes Overview and Architecture

Structure

Objective

Container overview

Comparison of container and virtual machines

Containerization benefits

Container image registry

Docker container

Docker Engine

OCI container image using Buildpacks

Pack

kpack

Difference between Pack and kpack

Install and configure Docker Desktop

Components of Dockerfile

Build Dockerfile

Inspect Docker image

Run Dockerfile

Build and run Docker image for Java app

Prerequisite

Public repositories using Docker Hub

Private repositories using Harbor

Install Harbor image registry

Introduction of Kubernetes

Kubernetes architecture

Master control plane/management cluster

Worker cluster

Need of Kubernetes

Kubernetes resources

Namespace

Service

Ingress

Ingress controller

Deployment

ConfigMap

Secret

Taints and tolerations

RBAC

Network policies

Storage

CronJob

ReplicaSet

StatefulSets

DaemonSets

Conclusion

Points to Remember

Key terms

References

11. Run Microservices on Kubernetes

Structures

Objective

Installing and running kubectl CLI commands

kubectl command modes

Kubernetes installation

kind

Minikube

Kubeadm

Enterprise Kubernetes

Kubernetes management UI tools Octant and Proxy.

Setup Octant K8s dashboard UI

Setup K8s proxy dashboard UI

Kubernetes application deployment and configuration management tools

Helm

Developer loves Helm

Difference between Helm 2 and Helm 3

Install Helm

Kubeapps

Install Kubeapps

Kubernetes operator

Skaffold

Tekton

Kustomize

Jenkins X

Spinnaker

Knative

Microservice deployment on Kubernetes container with NGINX

Ingress controller

Prerequisite

Conclusion

Points to Remember

Key terms

References

12. Service Mesh and Kubernetes Alternatives of Spring Cloud

Structure

Objective

GitOps

GitOps principles

GitOps benefits

Deployment of microservices on K8s with Argo CD

Benefits of Argo CD

Install Argo CD on Kubernetes cluster

Service mesh overview

Need of service mesh

Microservices challenges without service mesh

Service mesh features and benefits

Service mesh implementation tools

Istio service mesh architecture

Data plane

The Control plane

Istio service mesh best practices

Kubernetes alternatives of Spring Cloud

Kubernetes alternatives

Cloud buzzwords

Conclusion

Points to Remember

Key terms

References

Index

CHAPTER 1

Overview of Cloud Native Microservices

At present, cloud native technology is driving the modern revolution of build, run, manage, monitor, and secure modern applications on the cloud. Cloud native microservice is part of modern applications. There are a ton of benefits to deploy modern applications on the cloud platform. Every organization is building a new design or migrating from the legacy monolithic design to modern microservice applications using cloud native technology. **Microservices** and **containers** are new buzzwords for application and infrastructure, respectively.

So, the next questions are: *What is cloud computing? What are their benefits, principles, classifications, and so on? Why organizations need cloud native modern applications? What are microservices, principles, benefits, challenges cloud computing software and hardware modules? What are the best practices of microservices?* You will find the answer to these questions of cloud computing and microservices internals in this chapter.

Let's get started!

Structure

In this chapter, we will cover the following topics:

Fundamentals of cloud computing

Introducing cloud

Cloud principles

Business benefits of cloud

Types of cloud

Private/on-prem cloud

Public cloud

Hybrid clouds

Cloud classifications

Infrastructure as a Service

Container as a Service

Platform as a Service

Function as a Service or serverless architecture

Software as a Service

Database as a Service

Need for cloud native modern applications

Overview of cloud native modern application

Introduction to modern application

Cloud native overview

Microservices Architecture

Microservices characteristics

Microservices motivations and benefits

Microservices architecture principles

Microservices challenges and solutions

Beyond the twelve-factor apps

Application modernization spectrum

0-1 factors

1-4 factors

5-9 factors

10-15 factors

Objective

After studying this chapter, you should be able to understand the fundamentals of cloud computing, business benefits of cloud, and the need for cloud native microservices for modern applications. This chapter will help you to understand the cloud native microservices pattern, microservice architecture, challenges, and solutions. You will learn beyond twelve-factor principles and best practices of design, and development of microservices.

Fundamentals of cloud computing

We will cover basics of cloud for those who are beginners or have a little knowledge on all current cloud computing evolution.

Introducing cloud

Cloud computing is the on-demand availability of computer system resources, especially memory, data/cloud storage and CPU computing power, without manual management by the user. The cloud term is generally used to describe private and public data centers (DC) available to many users over the internet. Large clouds are distributed over multiple locations from central servers. If the connection to the user is relatively close, it may be designated an edge server.

Clouds may be limited to a single organization (enterprise clouds) on their on-prem or be available to many organizations (public cloud) on the internet. Cloud provides smooth experience on their platform with faster response, no outage, and highly available environment.

The following diagram gives a glimpse of different cloud computing software and hardware modules:

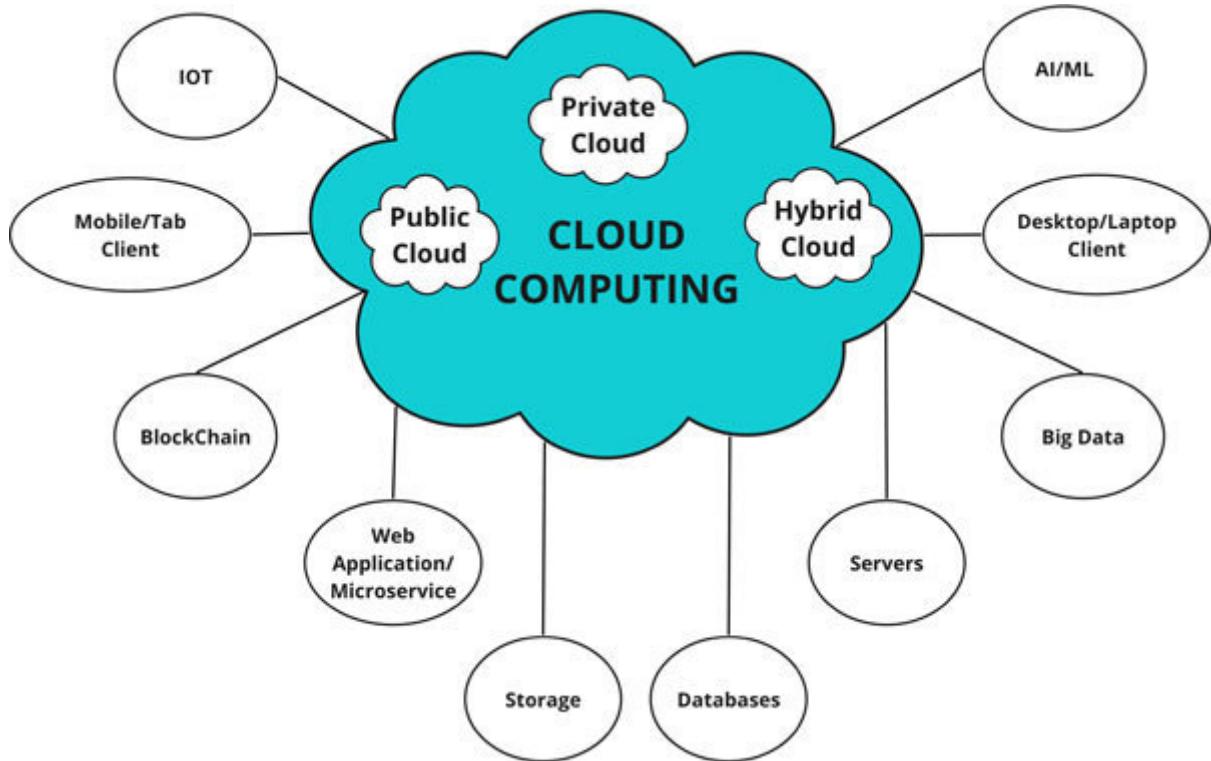


Figure 1.1: Cloud computing software and hardware modules

Cloud principles

Cloud has built on these main principles as follows:



Figure 1.2: Cloud principles

Let's discuss some major cloud principles:

Cloud provides auto-scalability of the applications and infrastructure.

Critical business apps can't afford outage. Cloud provides **high availability** where applications are deployed on multiple distributed environments such as **data centers** and **availability zones**

Cloud computing offers high-performance servers and super-fast SSD drives.

Cloud environment is self-healing with its advanced monitoring and recovery mechanism. Microservices apps and environments

easily recover and restore automatically.

Low Cloud services manage and automate all cloud admins and management tasks. That's how it provides low maintenance.

Cloud gives agility to build, run and manage applications on the cloud and launch in the market in no time. It provides auto scaling of environment by pay per usage costing model, which makes agility to organization.

It increases developer and DevOps operator's productivity. It provides various cloud services for the same.

Cost Cloud provides lower cost infrastructure. Organizations don't have to predict the infra environment and buy expensive hardware/software in advance, which may not be optimally utilized if usage is low. Cloud provides pay per usage infrastructure.

Multi Cloud provides multitenancy to deploy the same app on multiple servers and balance the load. Multiple customers and applications can share the same computing resources. Despite sharing same resources their data is kept separately with added security.

Advance Cloud provides advanced security to protect an organization's applications, infrastructure and data. Cloud environment provides multi-layer security at network, data and application layers such as **Distributed-Denial-of-Service** and **Open**

Web Application Security Project Cloud continuously takes backups; it's an ideal solution to ensure **business continuity**

The **National Institute of Standards and Technology** definition of cloud computing identifies *five essential*

On-demand A consumer can unilaterally provision computing capabilities, such as server time and network storage, as needed automatically without requiring human interaction with each service provider.

Broad network Capabilities are available over the network and accessed through standard mechanisms that promote the use of client platforms such as mobile phones, tablets, laptops, and workstations).

Resource The provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand.

Rapid Capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly outward and inward commensurate with demand. To the consumer, the capabilities available for provisioning often appear unlimited and can be appropriated in any quantity at any time.

Measured Cloud systems automatically control and optimize resource usage by leveraging a metering capability at some level

of abstraction appropriate to the type of service (for example, storage, processing, bandwidth, and active user accounts). Resource usage can be monitored, controlled, and reported, providing transparency for both the provider and consumer of the utilized service.

Courtesy: National Institute of Standards and Technology (NIST).

Business benefits of cloud

It's important to understand the benefits for the business who is planning to migrate to the cloud and invest time and money. These are some generic and common benefits after adopting cloud technology using modern application approach:

Smoother and faster app user Cloud provides faster, highly available app interfaces that improve rich user experience. For example, AWS stores static web pages and images at nearby edge servers on **Content Delivery Network** servers physically on cloud, which provides faster and smooth application response.

On demand scaling for Cloud provides on-demand compute, memory and storage horizontal/vertical scaling. Organizations/customers should not bother about infra prediction for higher load and they also save money to use only required infrastructure resources.

24*7 high Cloud provides high availability. Therefore, whenever any app server is down, the client load will be diverted to another app server, or a new app server will be created. User and client sessions will also be managed automatically using internal load balancers.

Less operational cost Cloud manages most of the infra management operations automatically or by cloud providers. For

example, PaaS automates entire platform automation with a smaller number of DevOps resources, which saves a lot operational cost.

Easy to Cloud providers and PaaS platforms provide a very easy and intuitive web, CLI and API-based console or interface, which can be easily integrated with the CI/CD tools, **Infrastructure as a Code (IaaC)** and scripts. They can also be integrated with apps, and so on.

Release app features quickly to compete in the Cloud provides a lot of ready-to use services on cloud like SAAS, which takes lesser time to build and deploy apps to cloud quickly using microservices agile like development methodologies. It supports container-based orchestration services such as Kubernetes, where smaller microservices can be deployed in quick time, which enables organizations to release new features quickly.

Increased Cloud solutions provide out-of-the-box intrinsic security features at various levels of application, network, data, and infra level. For example, AWS provides DDOS and OWASP security features with firewall, and so on.

Increase developer Cloud provides various tools and services to improve developer productivity like PaaS, Tanzu Build Service, Spring framework, AWS Beanstalk, GCP, OpenShift developer tools, and so on.

Modular Cloud motivates to follow modern applications microservice framework for dev and test teams to work in Agile on independent small and modular microservices independently.

Public cloud's pay as you go usage Customer has to *pay for pay as you go* usage of infra, so that no extra infra resources wasted. These public service providers pricing model saves a lot of cost.

Easy disaster recovery Cloud deployed on multiple DCs or AZs for **disaster recovery** so that if any site (DC/AZ) is down, then client or application load will be automatically routed to another site using server side load balancers.

Business continuity (BC): It provides all necessary processes and tools to manage BC for smooth and resilient business operations. They provide faster site recovery in case of disaster and data backup. Cloud also provides enterprise level compliances for various industries such as **Health Insurance Portability and Accountability Act** for health insurance.

Types of cloud

The following diagram gives various cloud hosting types in terms of various hosting methods:

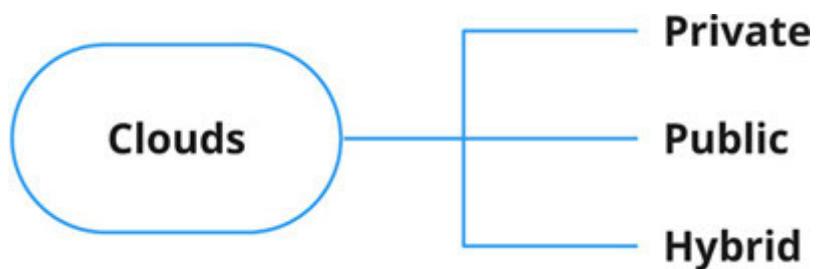


Figure 1.3: Types of cloud

Private/on-prem cloud

As the name says, it provides privacy and security to the cloud environment. It is sometimes hosted on an organization's site called **on-prem** or **private workspace** on the public cloud service providers. Private cloud platforms are typically built **on-premise** at customer's site. They can also be configured in a third-party DC and still provide the advanced level of privacy and security by hosting apps on physically dedicated isolated servers.

Public cloud

It's completely hosted on a public cloud service provider. Service providers manage all infra environments of their customers on the internet. Cloud provider provides various compute applications, databases, caching, messaging queue, analytics, IOT, AIML, blockchain, and so on.

Hybrid clouds

It's a combination of private and public cloud. Some of the organizations have both use cases requirements to keep part of the apps to their on-prem/private cloud for privacy and for internal users and remaining on public cloud.

Multi-cloud: **Some** organizations don't want lock in to a single cloud provider; they want to save cost or use services, which are not available at one service provider; that's why they distribute app and data among multiple cloud providers.

Cloud classifications

These are some major classifications of cloud computing:

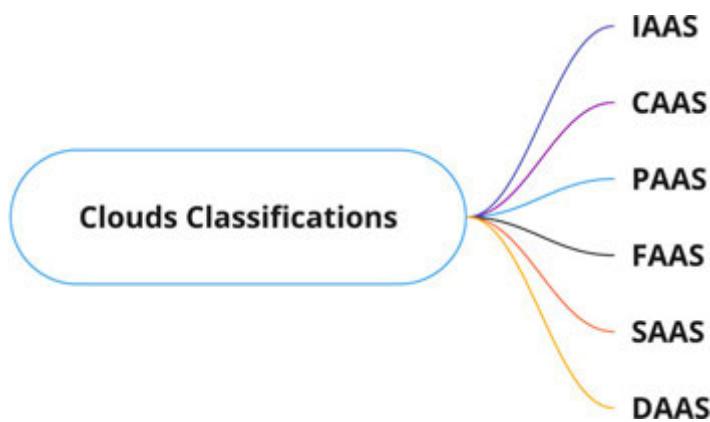


Figure 1.4: Cloud classifications

Here are cloud classifications with some related reference technologies and services of service providers:

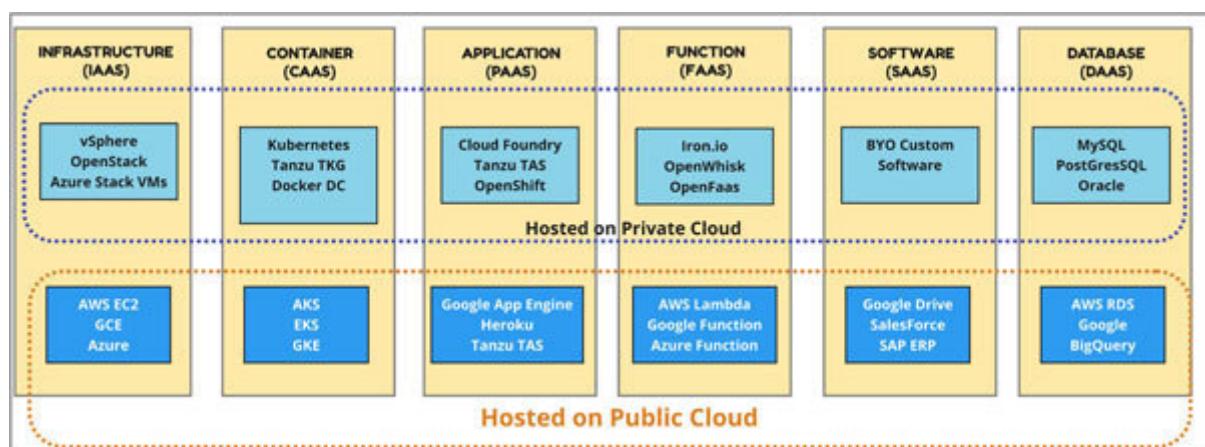


Figure 1.5: Cloud classifications reference architecture

Infrastructure as a Service (IaaS)

Infrastructure is the infra layer that provides a platform of **operating system** compute services, memory and persistence storage. It provides IP-based connectivity using TCP/IP, UDP and REST API services. It provides a **bare-metal** platform where applications can be deployed. For example, AWS, GCP, Azure, VMware vSphere/vCenter, and so on.

Use cases are as

An organization wants to manage their infrastructure on their own from the hardware layer to OS and runtime.

It's an economical option, however, needs a lot of manual operations.

Provide more control and flexible customization from hardware to software OS and application layers based on the business use cases of the organization.

Container as a Service (CaaS)

Container is a new IaaS for cloud native modern applications. Container is an executable unit/bundle of software in which application binary code is packaged with its dependent libraries and OS-related configuration dependencies. CaaS provides a platform to deploy, run, and manage software applications as containers.

Applications can be deployed and run on the containers, which are on top of the IaaS layer of virtualization hypervisor layer or BM (Bare-Metal) using container-based platform Kubernetes or similar. Containers and clusters are used as a service and deployed in the cloud or on-prem data centers.

Use cases are as

Run apps on smaller containers on top of bare-metal/hypervisor infrastructure.

When business needs smooth customer experience by providing faster, highly available, resilient, and scalable apps.

It needs lesser infra resources because it shares compute, memory and storage of OS based on the usage.

Platform as a Service (PaaS)

It provides an automated easy platform allowing developers to develop, run, and manage applications without the complexity of building and maintaining the infrastructure typically associated with developing and launching an application. It increases developer productivity, application development, and deployment, where developers only focus on business logic and rest of configuration code will be automated by PaaS platform.

Use cases are as

Faster releases to complete in market.

No vendor lock-in for backend and middleware software on cloud, because same PaaS software platforms manage your apps, databases, and infra on multi-clouds on private, hybrid, and public clouds such as Amazon AWS, Google GCP, and Microsoft's Azure.

Function as a Service (FaaS) or serverless architecture

It provides a platform where service providers manage all the server's infrastructure automatically for organizations by using an event driven model. It's an on-demand service with managed services that allows execution of microservices through API. In this approach, microservices are exposed as a function.

Public cloud charges *pay as you go* model; FaaS saves infrastructure cost by only spinning servers for a shorter duration on demand, for example executing a batch job at a scheduled time or short-lived services. It hides all infra-abstractions.

Use cases are as follows:

Good choice for short-lived microservices.

Good for event-driven microservices architecture model.

Cost-effective choice for apps that run rarely on-demand like batch jobs, analytical weekly/monthly reports, and so on.

Software as a Service (SaaS)

It provides ready-to-use software on cloud on the internet based on subscription. Many organizations are doing business by deploying these software on cloud and exposing them to end users on paid subscription. Popular examples are Google's services like Google search, Google Drive, Microsoft Office 360, WebEx, Zoom, and so on.

Use cases are as

Subscription-based cloud services.

Readymade services like e-mail service using MailChimp, AWS SMS, **Single Sign-On** OAuth services like Okta, file servers and office tool suite like Microsoft office 360 and Google G Suite, and so on.

Database as a Service (DaaS)

It's a subset of SaaS that provides SQL/NoSQL databases as a service on cloud through the internet. It provides strong security, performance backup and easily auto scaling and database upgrades, and patching. It's also called **managed** which are deployed on cloud, which can be accessed with simple REST APIs.

AWS RDS is a popular database as a service.

Use cases are as

Outsource database to avoid operational and scaling cost.

No overhead of upgrading and patching latest updates.

In-memory analytics.

Quick disaster recovery.

Data consolidation.

Mission critical applications that need high availability of data, with no outage.

Need for cloud native modern applications

We will try to understand business needs for adopting cloud-based modern applications by the organizations who had already adopted or planning to adopt cloud native applications.

Based on my experience, these are top 15 motivations of cloud native modern applications adoption:

High availability with auto scaling.

Resilient platform/automatic failover on the cloud.

High performance.

Zero downtime.

Reduce capital and operational cost.

Flexibility to achieve modern polyglot applications development technologies.

Managing heterogeneous workloads.

Improve developer productivity.

Enhance operator/DevOps productivity.

Deliver applications and features faster in market to compete.

Automate urgent hardware capacity needs with infrastructure automation.

Software and hardware quick updates and patching without outage.

Advance security to avoid any threats on application and infrastructure.

Reduce hardware and software cost.

Distribute workload on multi and hybrid clusters and manage through a single pane of glass.

Overview of cloud native modern application

Modern applications and cloud native application terms are being used interchangeably. These both terminologies are part of cloud native modern applications design, and development using modern technologies.

Only application modernization is not enough for cloud migration. We need to migrate infra, databases, development practices, automation, and so on.

Based on our experience of application development and design, this formula for cloud native modern applications is derived:

**Cloud-native modern applications = 15-factor principals +
microservices + modern database + API Driven Management +
Service Management + Cloud platforms + Containers orchestration
+ Serverless + DevOps (CI/ CD) + Open Sources + Web
assembly.**

We will try to cover all these cloud native modern applications ingredients in this book.

Introduction to modern application

Modern application is also called cloud native application, which follows **twelve-factor/ fifteen-factor** principals, microservice architecture patterns, best practices, CI/CD pipeline, modern databases, and containerized on containers.

Cloud native overview

Cloud-native a little more narrowly, to mean using open source software stack to be containerized, where each part of the app is packaged in its own container, dynamically orchestrated so that each part is actively scheduled and managed to optimize resource utilization, and microservices-oriented to increase the overall agility and maintainability of applications, the CNCF defines

Microservices architecture (MSA)

Microservice is a de facto architectural standard of developing cloud native modern applications. It's a group of autonomous services for a complete software system, which are separated by business model within bounded-context, it means every microservice will have isolated and independent responsibility. It's easy for them to build, test, deploy, run, secure, manage, and monitor.

MSA is a variant of the **service-oriented architecture** that provides a collection of loosely coupled modular services. It provides fine-grained and light-weighted services. A microservice is an independent module that can be built, deployed, and managed separately with no dependency on other microservice.

The following diagram has three microservices connected with their own databases independently. They may share a single database also based on the use cases. Technically they are isolated and their dev teams also work independently.

Example In an online eCommerce application, customer, order and product catalogue are categorized as three independent microservices.

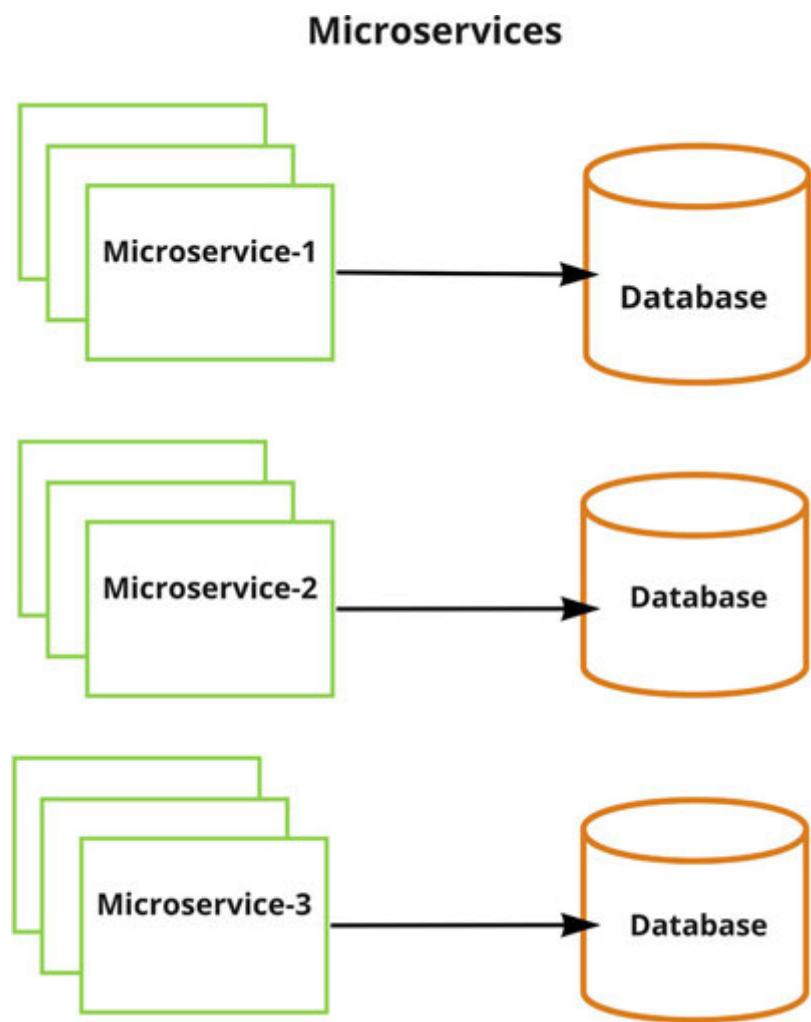


Figure 1.6: Microservices architecture (MSA)

Microservices characteristics

These are some important characteristics of microservices:

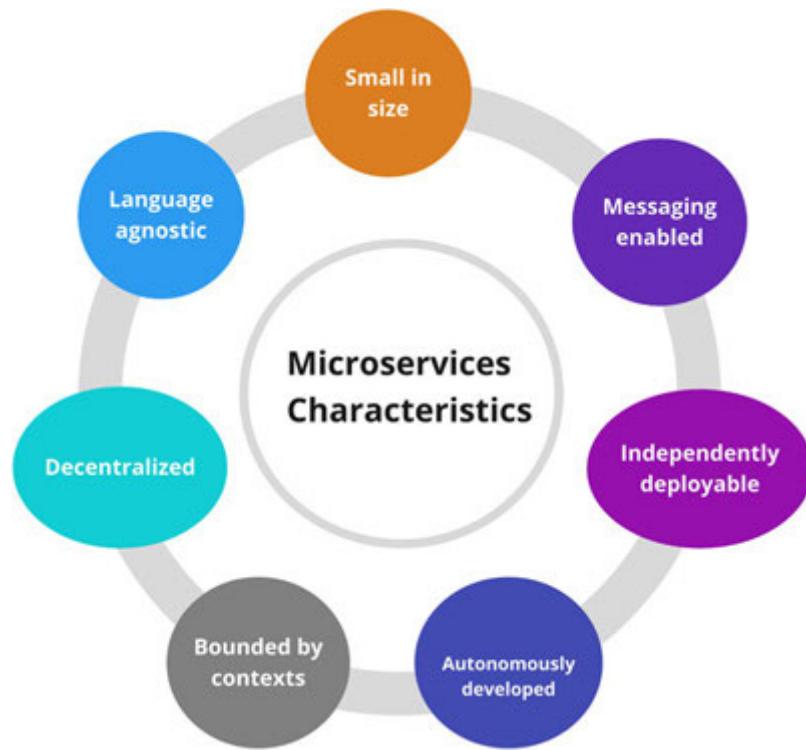


Figure 1.7: Microservices characteristics

Let us understand them one by one:

Small in Microservices should be in small size, modular, and decoupled to easily maintained, modified, tested, and deployed on a small size of container.

Microservices should follow well-established event-driven architecture design patterns practices.

Bounded by Every microservice should be bounded within their own context and should be independent within their business context. They should all follow a single responsibility principle.

Autonomously Every microservice should have their own small development team who works on that module separately without any dependency on other development teams.

Independently Since microservices are smaller deployable units, which can be deployed independently through CI/CD pipeline, they can be deployed on any container.

Microservices are deployed and run separately, it makes it easy to develop, test and run apps. They can be tested together using integration testing.

Microservices can be coded in any programming language and deployed as independent modules.

While you may see microservice and MSA used interchangeably, they're not quite the same. A microservice architecture is a style of engineering highly-automated based on twelve/fifteen factors and modern technologies, evolvable software systems, while microservices are lightweight applications, which follows MSA.

Microservices motivations and benefits

These are a few motivation and benefits of adopting microservices:

Migrate legacy large monolithic applications to cloud native modern applications.

Separation of business concerns.

Best fitted for distributed high usage environments.

Leverage container advantages.

Faster development and increase developer productivity.

Faster deployment.

Faster testing and release cycle.

Support Agile (Kanban/Scrum) development approach.

Faster cloud migration.

Save revenue, low operational cost.

Small development teams for independent microservices to follow the same architectural design and development standards.

Real-time easy monitoring and self-healing.

Isolation from failures.

Tracing and debugging in distributed environment.

Centralized apps health monitoring.

Monolithic is a single-tiered software application in which the user interface and backend data access code are combined into a single program.

Microservices architecture principles

These are a few important microservice architecture principles:

Microservices are scalable. They can be scaled-in or scaled-out on-demand based on usage.

Cloud native microservices are highly available. There are multiple application instances running on different servers, so in any case one server is down, another server will serve client requests.

They are easily maintainable with smaller and modular services with CI/CD pipeline and small development team. Kubernetes container platform provides these features out of the box for the microservices.

Auto Cloud providers and PaaS services take care of auto provisioning.

Fault tolerance and resilient Microservices should be fault tolerant and resilient, so that it can be easily recovered and provide backup services seamlessly.

Independent and autonomous lightweight modular These services should be isolated and modular for easy development, maintenance, and deployment.

Decentralized Microservices are deployed in decentralized environments on different servers, VMs or containers on multiple distributed DCs and AZs.

Dev and infra It also provides flexibility and agility of using agile iterative development approach, small and modular code. Also, it provides infrastructure agility to automate and change it easily using various CI/CD, IaaC tools, and so on.

Polyglot It provides polyglot persistence on different data storage and databases of file systems like S3 object storage, cloud storage, SQL NoSQL on cloud, and so on.

Real-time load balancing at client and server Microservices provide client and server-side load balancing real time on any cloud at application layer or network layer.

Compliment distributed systems Microservices is built with a distributed architecture where all services, databases and infra are all deployed on a multi-tenant infrastructure. They are well connected with each other and perform business requirements.

CI/CD It provides cloud native best practices of **Continuous Integration** and **Continuous Delivery** support to deploy and manage microservices apps on any cloud.

Microservices challenges and solutions

As mentioned, microservices are de facto standard for all modern applications and all cloud migration projects from monolithic to microservices. In spite of ton of benefits, there are some downside practical challenges too. Here are a few practical microservices architecture challenges and possible solutions:

solutions: solutions:

solutions: solutions: solutions: solutions: solutions: solutions:
solutions: solutions: solutions: solutions: solutions: solutions:
solutions: solutions: solutions: solutions: solutions: solutions:

solutions: solutions: solutions: solutions: solutions: solutions:
solutions: solutions: solutions: solutions: solutions: solutions:
solutions: solutions: solutions: solutions: solutions: solutions:
solutions: solutions: solutions: solutions: solutions: solutions:

solutions: solutions: solutions: solutions: solutions: solutions:
solutions: solutions: solutions: solutions: solutions: solutions:
solutions: solutions: solutions: solutions: solutions: solutions:

Table 1.1: Microservices challenges and solutions

Co-relation ID can be passed by the client in the header to REST APIs to track all the relevant logs across all the pods/Docker containers on all clusters.

Beyond the twelve-factor apps

Here we will discuss quick overview of twelve-factor applications standard and newly added three factors for modern cloud native application. It provides best practices of cloud native modern microservice applications.

Twelve-factor principal is de facto standard for a cloud native modern application, which is widely used. There are three more factors added that were identified by VMware, which makes it fifteen-factor apps:

One codebase tracked in revision control; many deploys.

Explicitly declare and isolate dependencies.

Store config in the environment.

Backing Treat backing services as attached resources.

Build, release, Strictly separate build and run stages.

Execute the app as one or more stateless processes.

Port Export services via port binding.

Scale out via the process model.

Maximize robustness with fast start-up and graceful shutdown.

Dev/prod Keep development, staging, and production as similar as possible.

Treat logs as event streams.

Admin Run admin/management tasks as one-off processes.

API API first suggests starting the API designing and exposing all cloud native microservices through REST API endpoints. Try to follow best REST API practices, set API contracts like request, response payload, API name, security, and arguments. In modern applications, cloud native apps are exposed and integrated through APIs. Use API gateways to follow these standards. Personally, Spring Cloud Gateway being an open-source to implement code centric API standards is the choice as it's faster, lightweight and easily configurable by the developers.

Add advanced features of monitoring hundreds and thousands of microservices apps, containers, and environment. It's very important to monitor logging, disk space usage, memory consumption, performance, and so on. Based on these telemetry data platforms can scale, self-heal and manage alerts for end users and platform operators. Analytics can be done using machine learning and based on that any organization can derive to future business strategy.

Security, Authentication and Authorization Security of microservice cloud native application is super important. Security is a major concern in the modern era, where hackers are really smart to steal confidential and critical information. Make sure almost all security policies are in place at hardware, network, and software.

Application modernization spectrum

The following diagram is depicting journey of *not-cloud ready* to *cloud native applications* using Fifteen-Factor principles and how it improves developer productivity and operational benefits:

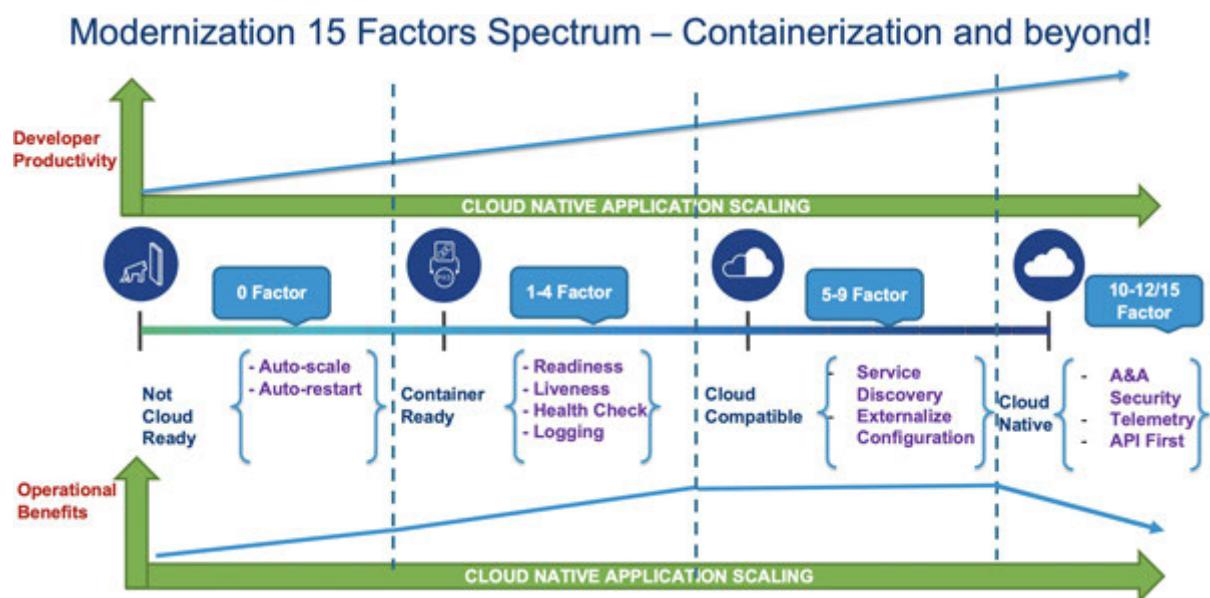


Figure 1.8: Fifteen-Factors modernization spectrum

Let's discuss about these cloud native journey from monolithic to microservices.

0-1 factors cloud journey

In this phase, we start with legacy/no cloud ready apps and apply minimal changes to migrate to cloud. Mainly, we should make configuration changes for auto-scaling, auto-healing and restart. Objective of this phase is to lift-and-shift apps on the Cloud with minimal code changes.

1-4 factors

In this phase, we continue from containerized app and try to convert to container ready by applying more cloud native factors practices like container supported readiness, liveness, health check and logging.

5:9 factors

In this phase, we continue from container-ready app and make it cloud compatible by applying container supported service discovery and externalize configuration for all microservices apps.

10-15 factors

In this phase, we continue from cloud compatible apps and make it fully cloud native by applying advanced cloud capabilities by applying IAM authentication, authorization, security, monitoring, observability, API first approach, and so on.

Conclusion

In this chapter, we have discussed the fundamentals of cloud computing, benefits, principles, classifications, and related technologies. We have covered introduction to microservices and their challenges, how to solve those challenges with different modern technologies.

We have tried our best to simplify the cloud and microservices concept in simple language which can be easily understandable to beginners to pro. Hope, now you have a basic understanding of cloud and microservices technologies benefits, use cases and *why it's gaining popularity!*

In the next chapter, we will discuss the most popular microservices design pattern, their advantages, limitations, and some valid use cases.

Points to Remember

Microservice is not a technology paradigm; it's a design pattern emerged from **Service Oriented Architecture** which is part of cloud native modern applications.

Generally, cloud term is referred to as public internet; however, cloud computing technology also works the same with an organization's own internal private network or intranet with some security restrictions.

Cloud can be set up on-prem at an organization's physical location or with third-party vendor's DCs with all cloud advantages of public cloud like IaaS, PaaS, CaaS, SaaS, DaaS, and so on.

Docker is a technology, a runtime container image which contains source code, dependent libraries and OS base layer configurations. It provides portable containers, which can be deployed and run on any container platform like Kubernetes.

There are two kind of private cloud:

On-prem at the organization's physical location or hosted by a third party.

Private cloud on public cloud infra where a separate server, databases, and infrastructure allocated to the organization to keep

their app and data with more privacy and noisy neighbors.

Modern applications, cloud native modern applications are similar terms.

The microservice architecture is a way to implement modern cloud native applications. However, monolithic/legacy applications can also be deployed on cloud infrastructure on containers without using real advantages of containers!

Microservices can be deployed on VM and containers on both, however container is better choice for high availability, scaling, security, and other infra-automation.

Key terms

Domain Driven Design is used for microservices design based on closed context of independent business modules.

A **Denial-of-Service** is a cyber-attack to shut down a machine or network, making it inaccessible to its intended users. DoS attacks accomplish this by flooding the target with traffic, or sending it information that triggers a crash.

It's an extension DoS. The incoming traffic flooding the victim originates from many different sources. This effectively makes it impossible to stop the attack simply by blocking a single source.

Application Program It's a computing interface which defines interactions between multiple software applications and can be interacted using REST.

Representational State Transfer is a software architectural style that defines a set of constraints to be used for creating web services.

The **Open Web Application Security Project** is an online community that produces freely-available articles, methodologies, documentation, tools, and technologies in the field of web application security.

Data It's a physical facility that organizations use to host their critical applications and data.

Availability Zones are isolated virtual locations within data center regions from which public cloud services originate and operate for HA, DR, and backups.

High availability.

Disaster recovery.

Authentication & Authorization.

Bare metal (on plain OS).

Operating system.

Structured Query Language for RDBMS databases like MySQL, Oracle.

No Structured Query Language for non-RDBMS databases like document-based MongoDB, or column-based HBase, Greenplum, and so on.

Bring Your Own. For example, bring your own software.

Do It It's used for mainly self-service management/operations.

Transmission Control Protocol and the **Internet Protocol** The IP suite is the conceptual model and set of communications protocols used in the internet and similar computer networks. It works on handshaking dialogues.

User Datagram Protocol is one of the core members of the IP suit. It has no handshaking dialogues.

A set of practices that combines software **development** and **IT operations** It aims to shorten the systems development life cycle and provide continuous delivery and continuous integration with high software quality. DevOps is complementary with Agile software development.

It's the philosophy of integrating security practices within the DevOps process. DevSecOps involves creating a *Security as a Code* culture with ongoing, flexible collaboration between release engineers and security teams.

Continuous Delivery and Continuous Integration.

Agile software Iterative sprint based rapid development model.

Hardware.

Software.

Command query responsibility segregation is a microservice design pattern to separate read and write command responsibilities of data.

Microservices architectural pattern to implement a consistent transaction that spans across multiple microservices.

Hypertext Transfer Protocol is an application-layer protocol for transmitting web-content.

HTTP with SSL security.

SSL and Secure Sockets Layer and its successor, **Transport Layer Security** are protocols for establishing authenticated and encrypted links between networked computers or cloud applications/services.

Greenfield: New scratch application.

Legacy or old application.

Multi-factor authentication which involved more than one device to authenticate like login credentials with mobile based OTP confirmation.

One-time password.

Role-Based Access Control is a method of restricting access based on the roles of individual users within an enterprise.

Single Sign-On for multiple applications from external authentication and authorization servers like Okta and RSA. Public Federated services also provide SSO such as Google, FaceBook, LinkedIn and so on.

IaaC: Infrastructure as a Code.

Amazon Web A leading public cloud provider.

Google Cloud A leading public cloud provider.

Content Delivery Network.

References

<https://12factor.net/>

API <https://apiacademy.co/>

Cloudification (my official blog) <https://cloudificationzone.com/>

Beyond the twelve-factor:

<https://tanzu.vmware.com/content/blog/beyond-the-twelve-factor-app>

CHAPTER 2

Microservice Design Patterns

Microservice applications are intended to deploy on distributed environments on multiple containers on multi-clouds. It's more likely that the same microservice can be deployed as multiple instances on multiple containers in this distributed environment.

Ideally, every microservice should have its own database and they work in isolation; however, during migration of brownfield legacy apps, monolithic and microservices might share the same database. It makes it very complex to communicate these microservices with each other in a distributed environment and solve complex business problems. Many moving parts in the **microservice architecture** creates challenges such as managing multiple instances of microservices, communication between microservices, transactional databases, monitoring, debugging, tracing and testing challenges, and so on. Microservice design patterns help to solve these real-problem statements and use cases.

In this chapter, we will cover a few important design patterns and principles of the modern cloud native microservices design.

Let's get started!

Structure

In this chapter, we will cover the following design principles:

Introduction to software design patterns

An overview of microservice design patterns

Application architecture patterns

Monolithic architecture

Microservices architecture

Microservices communication

Synchronous

Asynchronous

Decomposition of microservices

Domain Driven Design and bounded context

Strangler pattern

Integration patterns

Chain of responsibility

Aggregator and branch design patterns

API gateway design pattern

Micro frontends (UI composition) pattern

Database patterns

Database per service

Shared database

Command Query Responsibility Segregation

SAGA design patterns

Orchestrating microservices

Choreography microservices

Hybrid communication

Event sourcing and event-driven architecture

Observability and monitoring patterns

Distributed tracing

Log aggregation

Health check API

Application metrics

Service discovery design patterns

Client-side discovery

Server-side discovery

Circuit breaker design patterns

External configuration

API token security

Integration testing

Objectives

After studying this chapter, you should be able to learn about the various microservices design patterns and why they are needed. This chapter will help you understand the advantages and limitations of microservices design patterns and a couple of real-business use cases.

Introduction to software design patterns

A **software design pattern** is a general blueprint, set of rules, and reusable solution option for generic use cases or problem statements. So, in a nutshell, software design patterns are a set of techniques and algorithms to solve real-life business challenges in a systematic and standard way.

Overview of Microservice design patterns

Microservice is a well-known *de-facto standard architecture* to develop cloud native modern applications. Microservice applications are intended to deploy in a distributed environment on multiple containers. It means that the same microservice application can be deployed as multiple instances on multiple containers.

Ideally, every microservice should have its own database and they work in isolation; however, during the migration of brownfield legacy apps, monolithic and microservices might share the same database. It makes it very complex to communicate these microservices with each other in a distributed environment and solve business problems. There are many moving pieces which create many challenges like managing these microservice applications, database transactional issues, monitoring, debugging, testing, and so on. We need microservice design patterns to solve these real problem statements in a distributed environment.

Microservice design patterns are generic to any programming languages.

Application architecture patterns

We will discuss the following two very important design patterns of application design:

Monolithic architecture

Microservices architecture

Monolithic architecture

This is based on the legacy *N-tier application architecture* where all the business services are bundled in a single big application. So, all business logic, configuration, and integration are composed in a single application which uses a single database for all business services. It's good for small applications. It's easy to implement, integrate, and debug. The development team can use the same source code, test, and debug end-to-end functionalities of internal services easily. It creates a single deployable file, which can easily be deployed.

This is a typical architecture of monolithic applications:

Monolithic Architecture

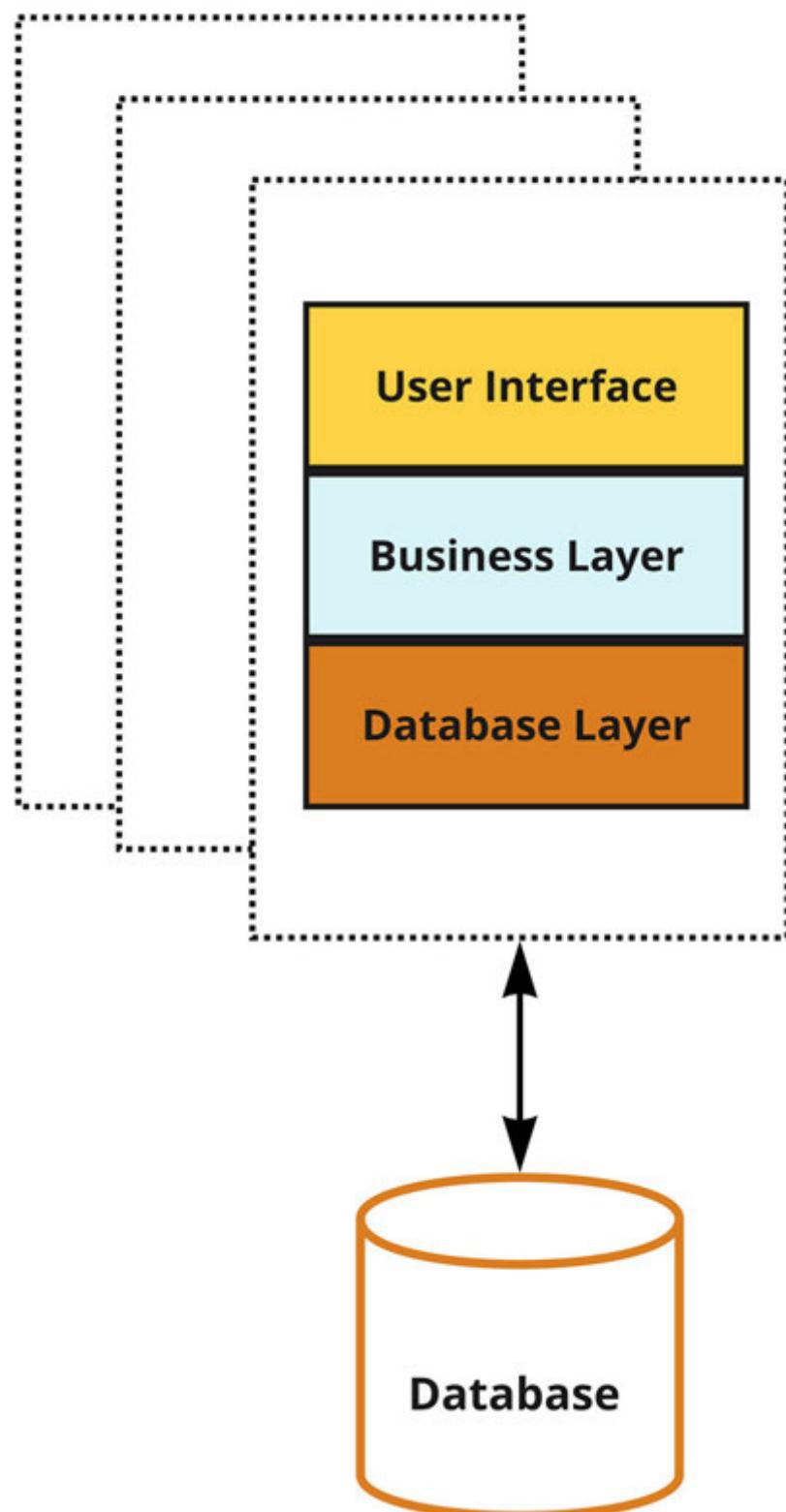


Figure 2.1: Monolithic architecture

These are some of the challenges of the monolithic architecture, which motivated the microservice architecture:

Hard to change

Integration issues

Long testing and release cycle

Hard to scale infrastructure

Complex DevOps CI/CD cycle

Not reliable

Not highly available

Not resilient

Hard to adopt new technologies trends

Difficult to achieve operational agility

Big ball of a black box, hard to understand, and debug the code

There are workarounds and solutions to these challenges; however, this model doesn't comply with the **twelve-factor/fifteen-factor** cloud native modern applications for containers.

Monolithic applications can be deployed on Kubernetes containers as a big application with large compute resources and hardware as a lift and shift model by making minor configuration changes to make it cloud native; however, these applications can't achieve the real value of containerization like scalability, high availability, and so on.

Microservices architecture

Microservice is a de-facto architectural standard of developing cloud native modern applications. It's a group of autonomous services which are separated by the business model within the bounded-context; it means every microservice will have an isolated and independent responsibility, as they don't depend on other microservices. It's easy for them to build, test, deploy, run, secure, manage, and monitor.

The microservice architecture is one of the variants of the **service-oriented architecture** which provides a collection of loosely coupled modular services. In a microservices architecture, services are fine-grained and light weighted. Microservice is an independent module which can be built, deployed, and managed separately with no dependency on any microservice.

The following diagram displays three microservices that are connected with their own databases independently. Technically, they are isolated and their development teams are also working independently:

Microservices

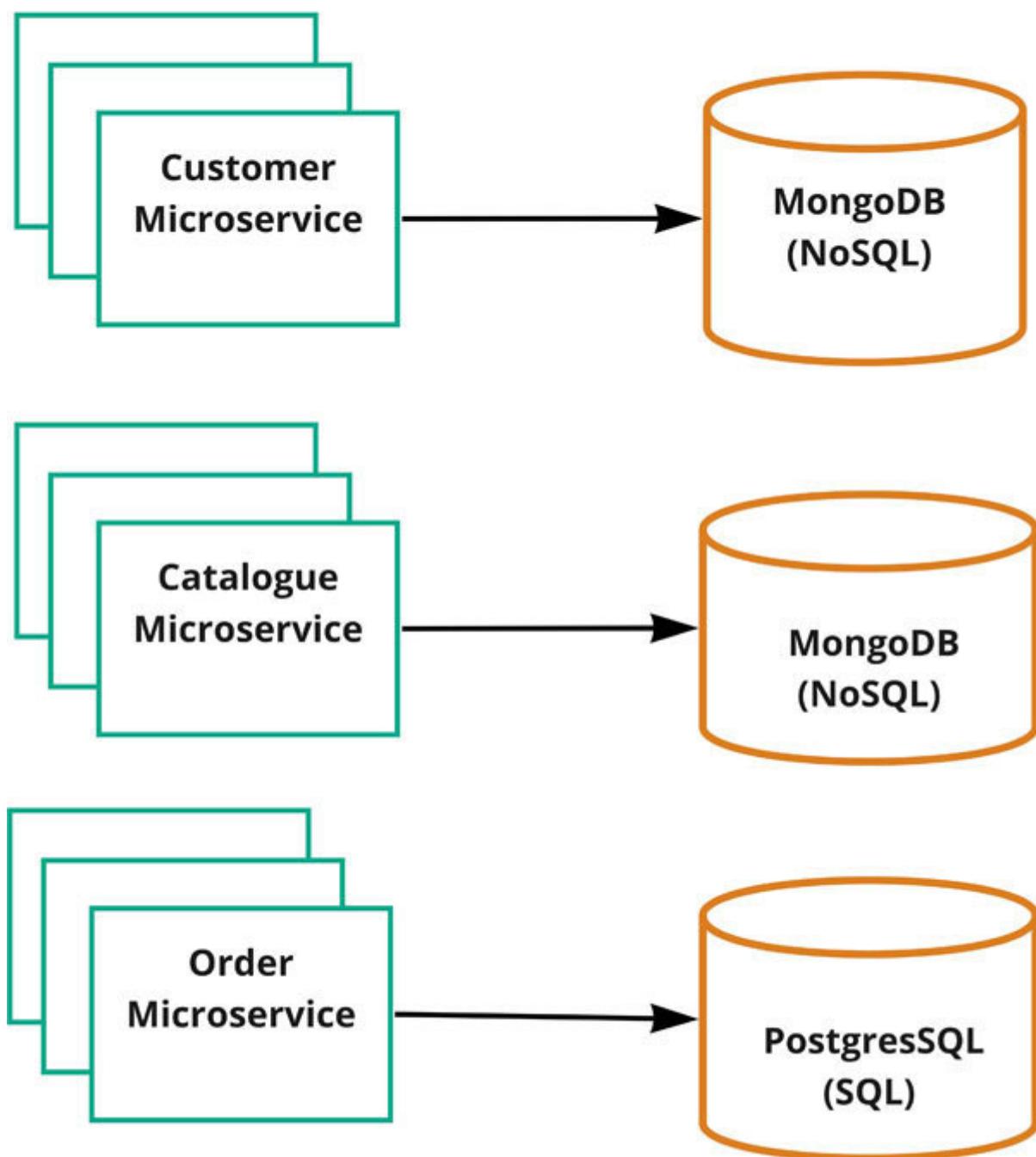


Figure 2.2: Microservices architecture

Microservices communication

There are mainly two types of microservices communication methods in the cloud native landscape:

Synchronous

Asynchronous

Synchronous communication

It provides direct communication between microservices. In this communication channel, one microservice consumer can send requests and get responses immediately from the producer microservice. It's good for use cases where a synchronous response is required without any delay like authentication-based services.

Order microservice calls the catalogue and customer microservice through the **point-to-point request-response** call.

This following diagram depicts the synchronous communication:

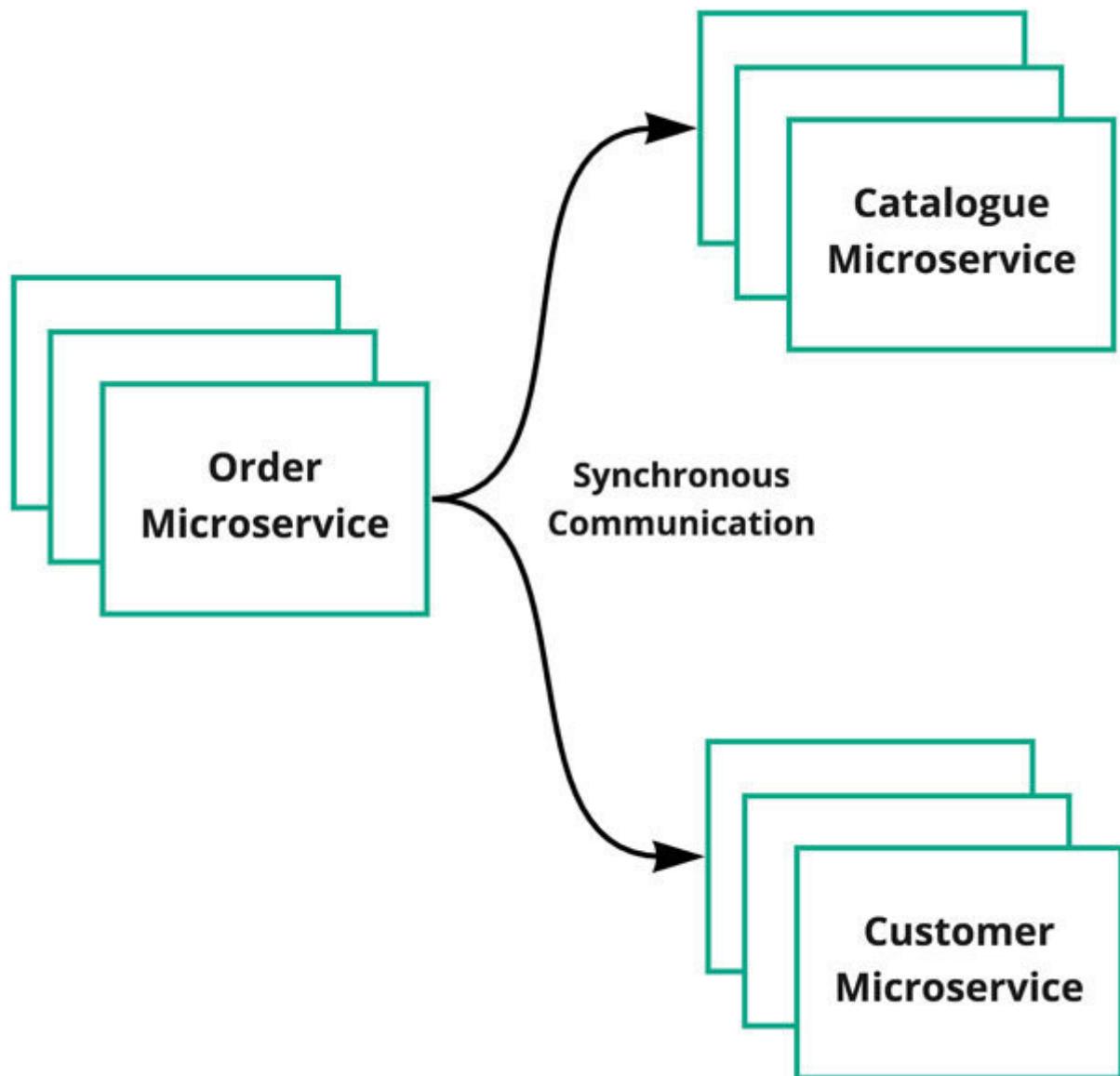


Figure 2.3: Synchronous communication

Asynchronous communication

It's an indirect communication between microservices, where microservices run independently and communicate with other microservices through the **async messaging**. Participant services listen and communicate with each other through messaging broker topics.

This following diagram depicts the asynchronous communication:

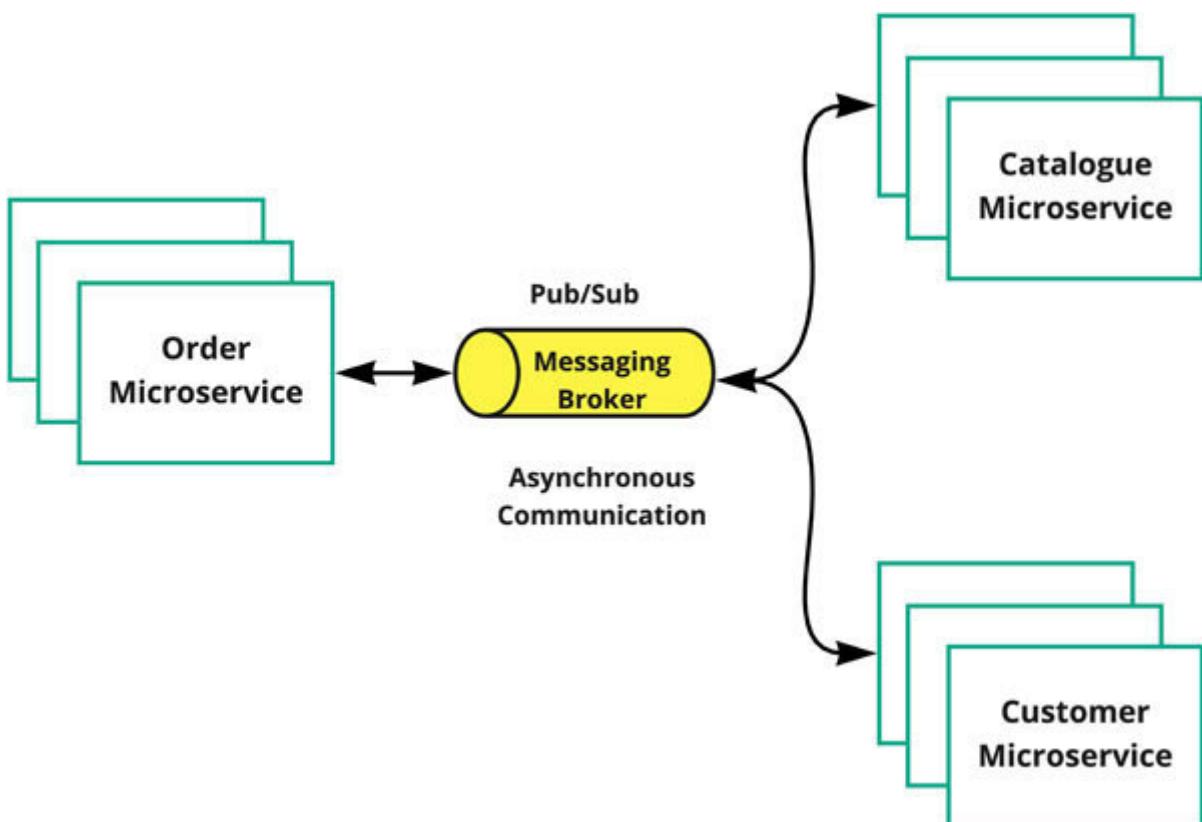


Figure 2.4: Asynchronous communication

Topic: A logical pub/sub queue where events can be published and consumed. This is called **async eventing**.

Decomposition of microservices

It's a very important design pattern aspect when we initially plan to decide decomposition for legacy monolithic applications to microservices, or create new microservices from scratch. We need a way to decompose microservices based on some base principles which will be easy to understand and maintain independently.

Problem statement

Decompose business and legacy monolithic services into smaller modular microservices. *How to design decoupled microservices for greenfield projects from scratch?*

These are some ways of decomposing microservices:

Domain Driven Design

Strangler pattern

Domain Driven Design (DDD)

Before we discuss DDD, let's understand how a large business system can be decomposed to the bounded context. Then, we will discuss how we can apply domain-driven design patterns and identify domains and subdomains. Finally, these domains/subdomains can be converted into single responsibility microservices.

Introducing bounded context

Bounded context is a group of subdomains. It's a logical boundary for domains and subdomains. DDD divides a large business system to small bounded contexts; each bounded context has its own model. A microservice is a bounded context; there could be multiple microservices inside the same bounded context but not vice versa. The same data model may have different meanings for different microservices.

In the following diagram, there are two bounded contexts: **Sales** and **Sales Bounded Context** has two subdomains such as **Order** and **Payment** and **Inventory Bounded Context** has the **Inventory** domain. The **Catalogue** sub-domain represents different things in each bounded context. It's like the fruit apple is *Apple* in English and *Manzana* in Spanish. The sub-domain takes on different behaviour and characteristics depending on the bounded context such as the **Catalogue** domain that is common with **Sales** and **Inventory Bounded**

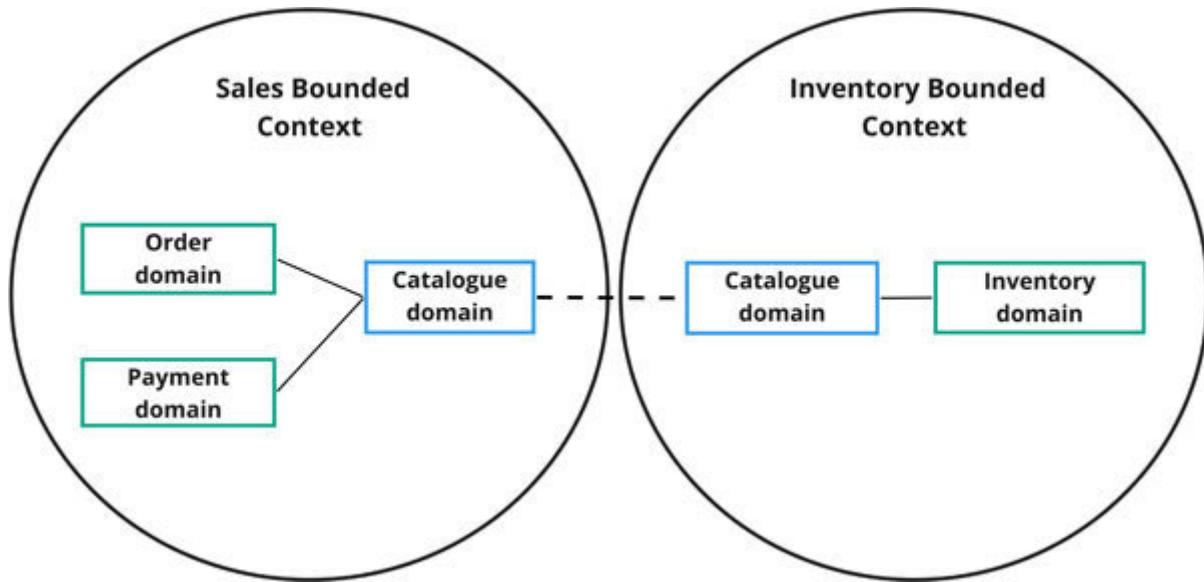


Figure 2.5: Bounded context

Now, let's discuss the insight of DDD. As per the modern cloud native microservice design, all microservices should be loosely decoupled. Every microservice module should work on a **single responsibility**. We will need a logical and meaningful way to split an application into smaller microservices. These services will work in a bounded context model in isolation.

These are two standard ways to decompose applications into smaller domains:

Based on business

Applications can be decomposed based on the business processes of the organization like customer service, catalogue service, order service, and so on. Sometimes, microservices can be divided into business domains/departments like finance, HR, inventory, delivery, sales, and so on.

Based on business function or

Some application modules are common and spread across multiple business use cases like an order service can be divided into multiple modules like order management, order packing, delivery, and so on. These related subdomains can be combined within a single order service.

Now, in the following diagram, we will understand how domains and subdomains can be transformed to microservices:

Domain => Microservice

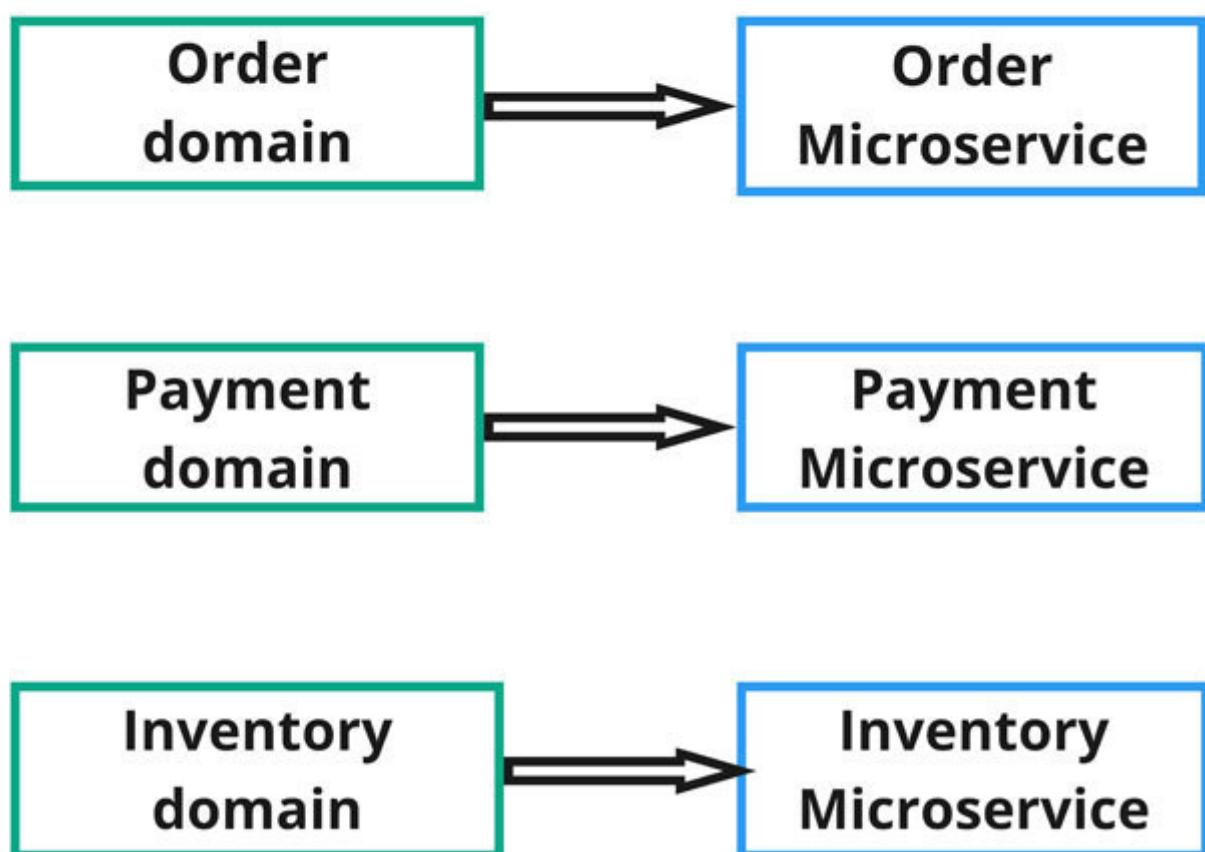


Figure 2.6: Business domain to microservice transformation

Advantages

The advantages are as follows:

Business centric

Loosely coupled

Easy to modify, upgrade, and unit test

Easy to understand and debug

Easy to deploy with CI/CD pipeline

Limitations

The limitations are as follows:

Difficult to do integration testing for end-to-end business use cases.

Identifying and understanding subdomains is a little complex exercise.

Use cases

The use cases are as follows:

Decompose a big monolithic application to smaller microservices.

Design new microservices for a greenfield project.

Strangler pattern

Majority of the current applications are legacy monolithic applications, which can't be decomposed in microservices in a single attempt. It takes a long time to migrate all monolithic applications into microservices, which can be done in phase-wise.

Modern cloud native applications follow the Agile development approach, where they are deployed in phase-wise sprints in non-production and production.

In this design pattern, a monolithic app can be broken into smaller business domains in phase-wise. In this **phase-wise** first an independent low business critical service can be extracted from a monolithic app as a microservice and deployed on production, where client applications access the newly extracted microservice and old monolithic apps both. Other business services of a monolithic app can be extracted to microservices to cloud after successfully testing phase-wise. In this approach, the monolithic app and extracted microservices both are deployed on production at the same time. The client call goes to both until all services of the monolithic application can be extracted to microservices and deployed on production. When all monolithic applications are extracted to microservices, then the monolithic app will be retired at the end.

The strangler design pattern is best suited for migration from legacy monolithic applications to microservices.

This following diagram depicts the stagewise transition of monolithic apps to microservices:

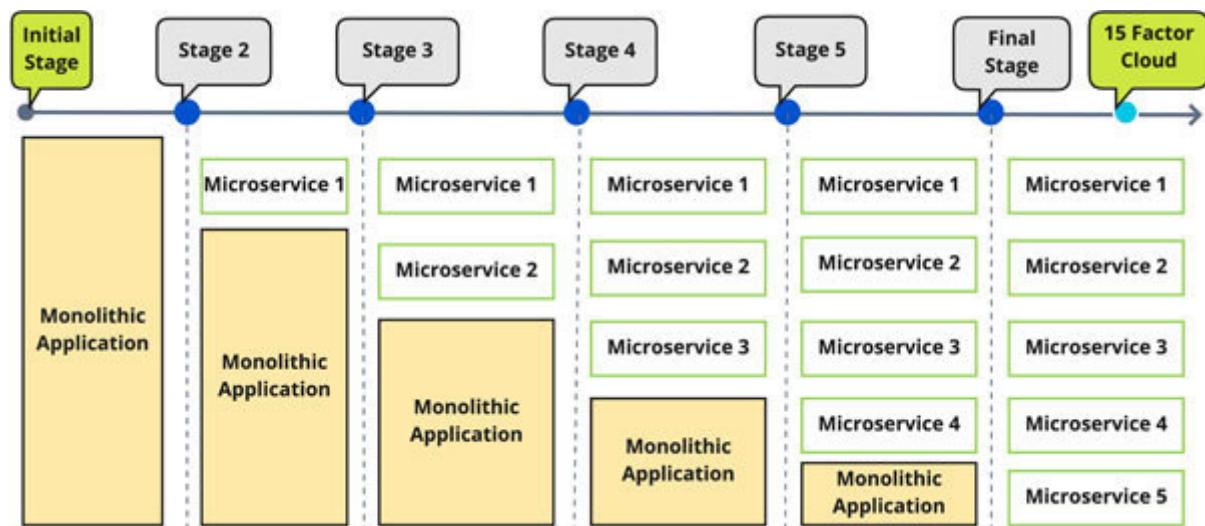


Figure 2.7: Strangler pattern

Advantages

The advantages are as follows:

It's good for web applications.

It's a safe and easy way to migrate monolithic apps to modern microservices in phase-wise.

Limitations

The limitations are as follows:

This process takes time to fully migrate monolithic apps to microservices. Sometimes, it takes a couple of months to years.

This is complex to manage, test, and debug because business logics are spread across both microservices and monolithic applications.

Use cases

Migrate a big legacy app to microservices which contains many business logics.

If an organization wants to move to the cloud phase-wise to avoid any kind of risks.

Integration patterns

In the microservices architecture, all services are distributed and deployed on different servers and containers. They all communicate with each other. We will discuss a few important integration design patterns:

Chain of responsibility

Aggregator and branch design patterns

API gateway design pattern

Micro frontends (UI composition) pattern

Chain of responsibility

This is a **legacy design pattern** where all microservices call each other in a chain or a given sequence to complete a use case end to end and produce a single response. For example, if three microservices **C** are connected with each other, then **A** will be called first from the client, then service **A** will call service and then service **B** will call service In this case, service **C** will send the response to then service **B** will compose the response from and then combine it with its own response and send it to consumer **A** as a final response. This is not recommended when we have more than two or three services. The downside of this design pattern is that if any service is down in the chain, then the entire chain will be broken. For example, if service **B** fails, then **A** will not get a response and will not return a final response to the client.

This following diagram depicts the chain of responsibility integration of three microservices:

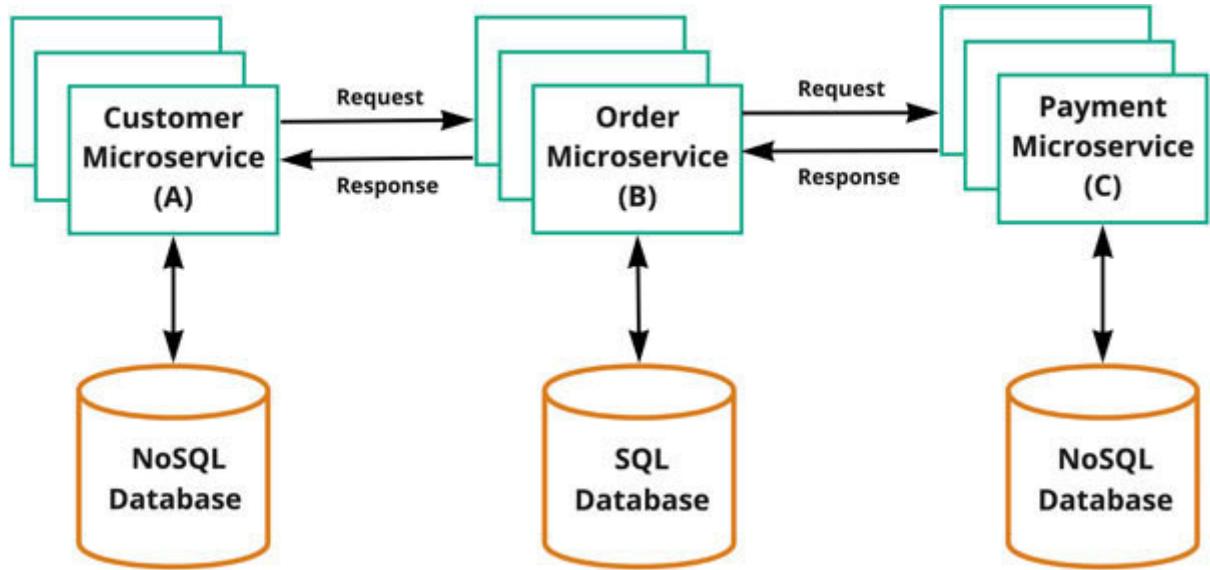


Figure 2.8: Chain of responsibility

Advantages

The advantages are as follows:

Easy to use and easy to implement.

Requests will be different between services **A** and **B** and

Request-response can be customized.

Limitations

The limitations are as follows:

Tight coupling between microservices.

If one service is down, other dependent services won't work and the chain will be broken.

Use cases

The use cases are as follows:

When two or three small number of services is connected with each other like login services.

Good for small enterprise systems.

Aggregator and branch design patterns

In this design pattern, two or more services work in parallel and combine their requests and responses at the end. So, unlike chain of responsibility design patterns this is not sequential and works in a sequence/chain. Parallel processing is possible, which creates advantages on top of the chain of responsibility design pattern.

The following diagram depicts branch design patterns of two microservices:

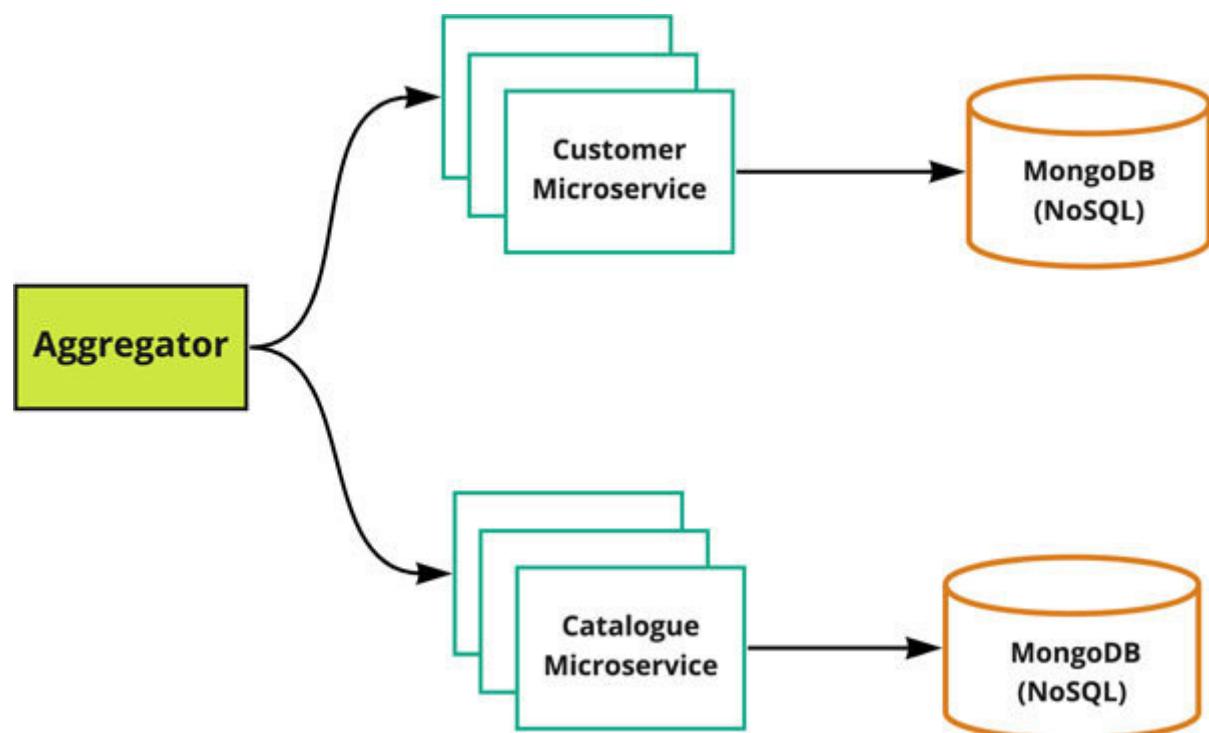


Figure 2.9: Aggregator design pattern

In this design pattern, one service can parallelly call more than one service and aggregate the response and return it to client apps. We can get responses from different sources and then combine the results based on the client requirement.

Advantages

The advantages are as follows:

Parallel processing makes processing faster.

Easy to understand.

Easy to implement and debug.

If any service is broken, it won't break the link like a chain of responsibility.

Limitations

The limitations are as follows:

Internally, it also has dependencies like the chain of responsibility. If one service is broken, other dependent services will also be impacted. Even though it's parallel; however, if your client **A** is requesting a combined response of service **B** and your service **B** is down, then this entire business transaction will not be completed.

The main consumer service will have extra responsibility to call other services and combine their responses by filtering the logic. It's overhead also, and sometimes it gets stuck when producer services are down and do not respond to the main service.

Single point of failure: If the main consumer service is down, there will be no response to client requests.

Use cases

The use cases are as follows:

When more services are connected with each other in the branching pattern one-to-many.

Parallel processing by using different microservices.

Good for small number of microservices intercommunication.

API gateway design pattern

This is a very important microservice pattern which works as a single point of entry for all kinds of backend microservices communication through the REST API. It provides a façade-based controller which provides an abstraction layer which decides which API will serve which client request. The API gateway seems to be just a specialized form of aggregator with some more advanced features.

It's deployed separately as a separate orchestrator application. Clients and other microservices first interact with the API gateway; it has all routing and filtering business logic which routes the client request to the appropriate microservice REST API for further processing. It also manages multiple workloads from different kinds of clients like web, mobile, IoT, and so on.

This following diagram depicts the API gateway which is a single-entry point between web/mobile client and microservices:

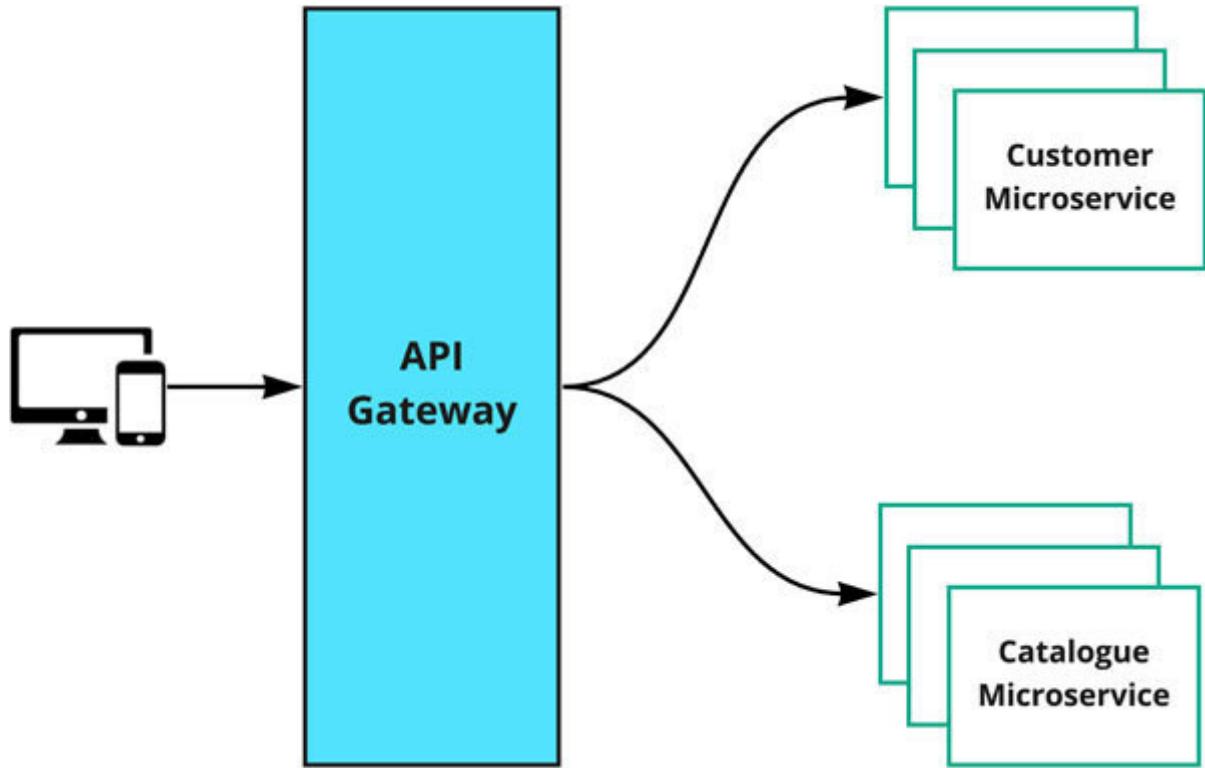


Figure 2.10: API gateway aggregator

Advantages

The advantages are as follows:

A single point of controlling and monitoring traffic of all APIs.

Easy to configure and add new routing of APIs with simple configuration.

Improve performance of APIs.

It provides cross cutting concerns of caching, service discovery, rate limiting, tracing, and other features.

It can be scaled also, by deploying multiple API gateways to provide **high availability**

Limitations

The limitations are as follows:

Sometimes, it increases extra latency because even internal microservices APIs can communicate to other APIs by the API gateway.

It makes it complex to implement.

Single point of failure, which can be solved by adding other gateway instances in parallel for HA.

Use cases

The use cases are as follows:

The UI/REST client interacts with other microservices.

Paid subscription of public APIs.

Improve performance by caching response of the REST API.

Must have the container-based microservice architecture.

Micro frontends (UI composition) pattern

This design pattern is designed for front end/UI applications to display the API response data from different microservices. In this design approach, a **single page application** application is split into smaller tiles/sections which shows data from different backend microservices. In this scenario, multiple tasks can be executed simultaneously and the entire page won't be broken if any tile or microservice is down or has some data issues. It extends the same concept of microservices in the frontend development methodology. In this pattern, the entire single UI page is considered as a composition of different services with their different business responsibilities.

The biggest advantage of this UI development pattern is that multiple UI development teams can work independently on their UI section/tile. They can test independently their new features and release quickly to the production environment.

The micro front-ends itself has a number of patterns. Refer to the following link:

<https://cloud.rohitkelapure.com/2018/11/decomposing-ui.html> and
<https://micro-frontends.org/>

In the following diagram, **Amazon** eCommerce applications split into multiple sections/tiles:

Catalogue Microservice

Customer Microservice

Order Microservice

Cart Microservice

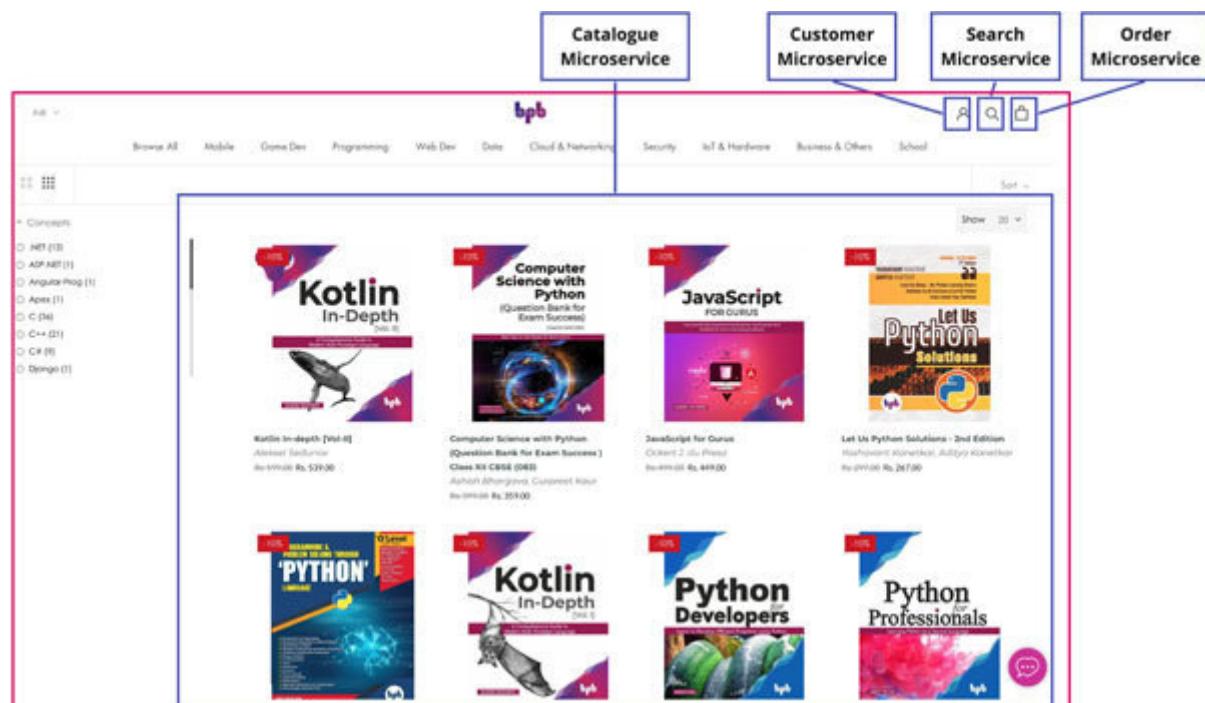


Figure 2.11: Micro-frontends example page

They all are connected to their own microservices REST APIs, which works on the UI client request and response model. They load data on the page individually; if one microservice is down, then it won't impact other services.

Advantages

The advantages are as follows:

Easy to implement and change.

Easy to deploy using CI/CD pipeline.

Easy debugging.

Multiple UI development teams can work independently on their UI section/tile.

Developed as isolated microservices.

Easy to test and maintain.

Databases can be also isolated with a single section of web page.

Build a resilient web portal.

Limitations

The limitations are as follows:

If any service is down, then showing partial data to the end customer/user will create a bad user experience.

Hard to integrate and test real end-to-end business use cases, when the page data is distributed.

Sometimes, it's also possible to fetch data from one section from different backend services at the backend. If any dependent microservice is down, then the expected data can't be shown on that given page section.

High dependency on underlying services and databases.

Use cases

The use cases are as follows:

Display multiple sections/tiles with their own set of data. Every section has connected with the same domain microservices.

Split the UI team for different sections of the page.

Faster UI development with quick production releases.

Parent-child kind of page relations.

Display different app tiles on a single page.

Monitoring and analytics dashboard

SPA: A Single Page application is considered to be the best design for a front-end UI application and it is very popular at present.

Database patterns

When we plan about cloud native applications, it doesn't only mean modernize applications. It's also meant to modernize legacy databases. At present, there are multiple SQL and NoSQL-based applications, which can be chosen based on the business use cases and type of structured and unstructured data.

Earlier, we had only VM supported databases. Now, databases can be also installed on containers which are scalable and **highly available**

There are multiple database patterns which are suitable for the microservice architecture:

Database per service

Shared database

Command Query Responsibility Segregation

SAGA design patterns:

Orchestrating microservices

Choreography microservices

Hybrid communication

Database per service

In the microservice architecture, every microservice should have its own database and they work in isolation; however, during migration of brownfield legacy apps, monolithic and microservices might share the same database. Every database has a separate requirement of memory, CPU speed, avoid duplication of data, and so on. It's advisable to have a separate database per microservice to only store relevant data for that service, which makes it easy to manage, deploy, and upgrade in a container environment for the lightweight microservices.

The following diagram depicts that **Customer** and **Order** microservices have their own databases:

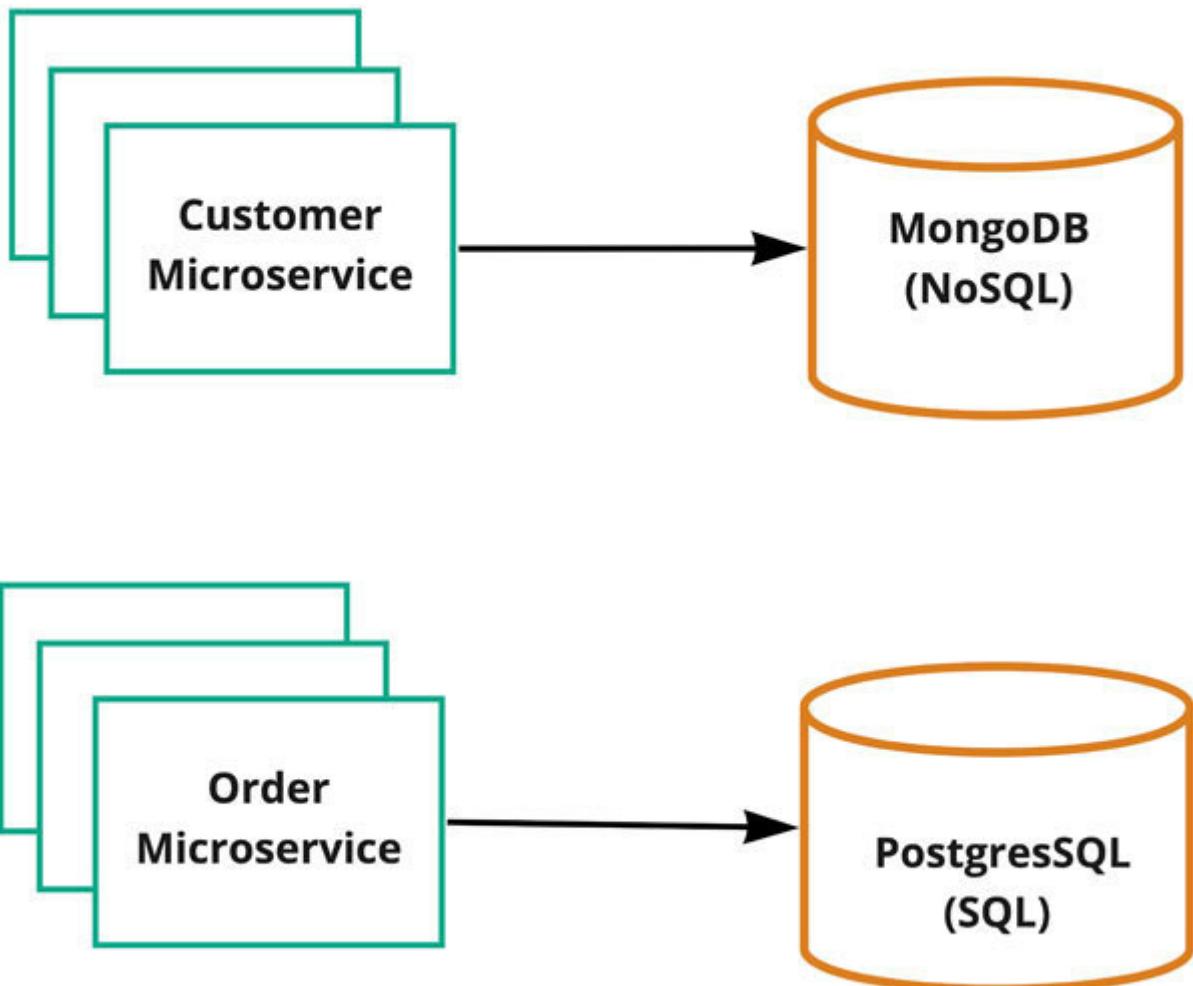


Figure 2.12: Database per service

For example, order microservice can use the transactional structured data for which SQL-based databases are a good choice. If it's catalogue data, a NoSQL database is a good choice. If it's a search intensive service, then choose search engine-based databases.

There are multiple ways to database mapping with services. The table for per data model is an approach for easy mapping and data response customizations using different data modelling APIs.

These separate databases work independently and don't make any direct database connection with each other. All database communication happens with the help of microservices applications.

Advantages

The advantages are as follows:

It helps microservices loosely coupled and scalable.

Databases can be easily scalable on the container environment.

Easy to debug and test microservices with their separate databases.

No strong dependency on other databases.

Simple to update and upgrade the database.

Lightweight to deploy on the container and highly available.

Provide SQL and NoSQL support.

They can be integrated with CI/CD DevOps pipeline, which can be updated using GitOps and by automating DB scripts.

Limitations

The limitations are as follows:

It doesn't support transactions, where commit happens in two-phases or where multiple cross tables are interconnected.

Doesn't fully support the CAP theorem for ACID transactions.

Use cases

The use cases are as follows:

Choose document-based database MongoDB for the catalogue microservice, where write one and read many use cases. Here, updates are very rare.

Choose SQL-based MySQL/PostgreSQL for transaction structured data in tables.

Deploy the database as a service on the Kubernetes container for high availability.

Scaling databases on demand on the container environment.

Choose use cases-based databases for different services like Elasticsearch search engine for intensive search use cases, where

it needs high memory and CPU speed. However, MongoDB document-based databases are good for storing product catalogue, which works like static data and rarely changes.

Shared database

In some common use cases, where the data type is also the same, multiple microservices share the same database. This design pattern is good for a smaller number of microservices from two to four, where they share common database tables from a single database with similar structural data type.

The following diagram depicts **Customer** and **Order** microservices sharing the same database:

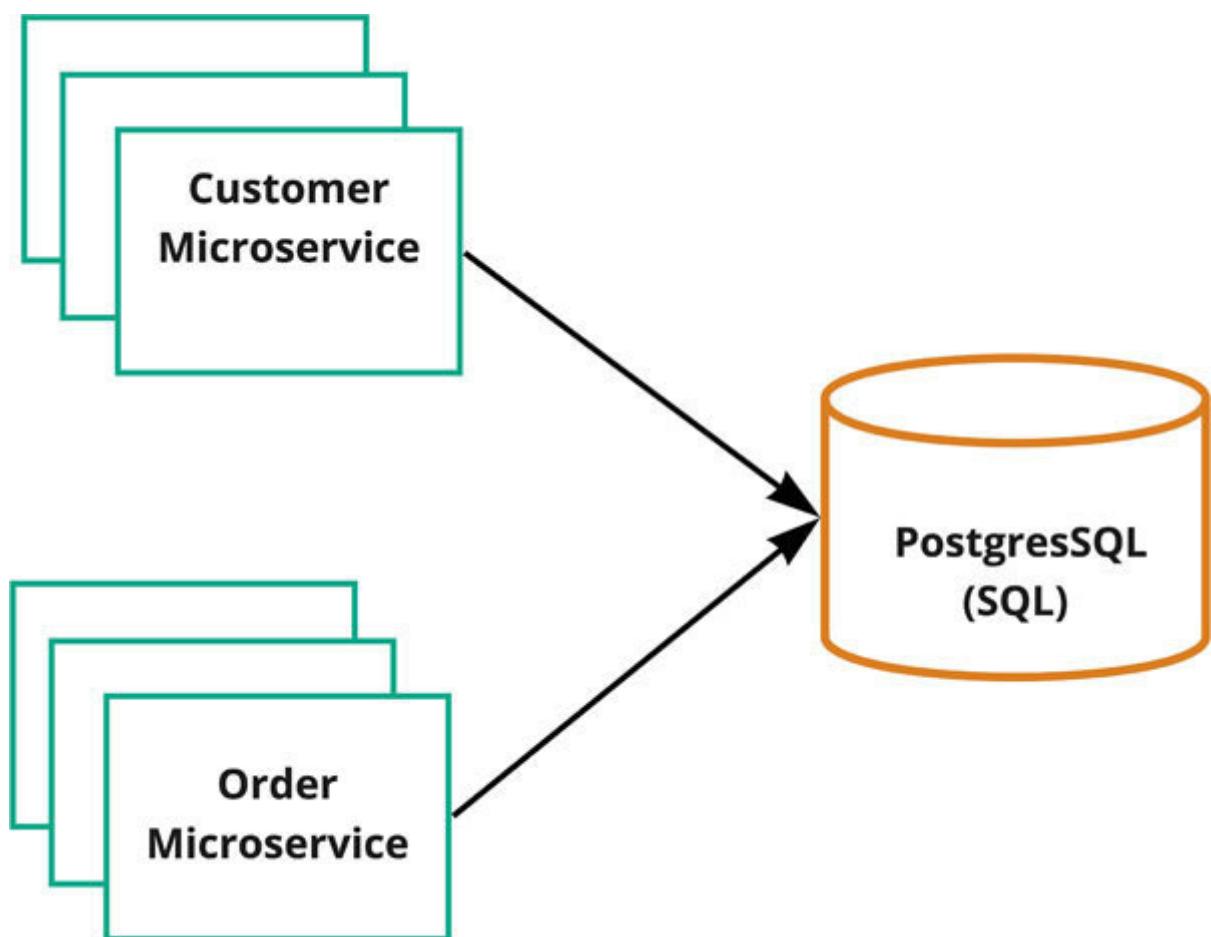


Figure 2.13: Shared database

Advantages

The advantages are as follows:

Multiple microservices can access same databases.

Small number of microservices can be used for ACID transactions.

Easy to integrate with services and map with their data model.

Easy to normalize database tables.

Limitations

The limitations are as follows:

Every microservice has different kind of data processing needs based on type and operations on that data.

Tight coupling with other microservices for data dependency and updates.

Developers also will be dependent to other microservice development teams for common functional changes.

Use cases

The use cases are as follows:

To maintain the ACID statement among a few similar microservices.

Some monolithic applications can't directly split their database per service during the migration phase. Initially, monolithic apps migrated to microservices, but they share the same database. In later phases, shared databases will be broken to a single database per service.

Command Query Responsibility Segregation (CQRS)

It's a database pattern where the command (write) and query (read) have their separate databases and separate responsibilities. CQRS is divided into two parts:

delete

Read query

The following diagram depicts **Customer Microservice** has its own write data model for all and **delete** operations. **Order Microservice** has the read data model exclusive for read queries purpose:

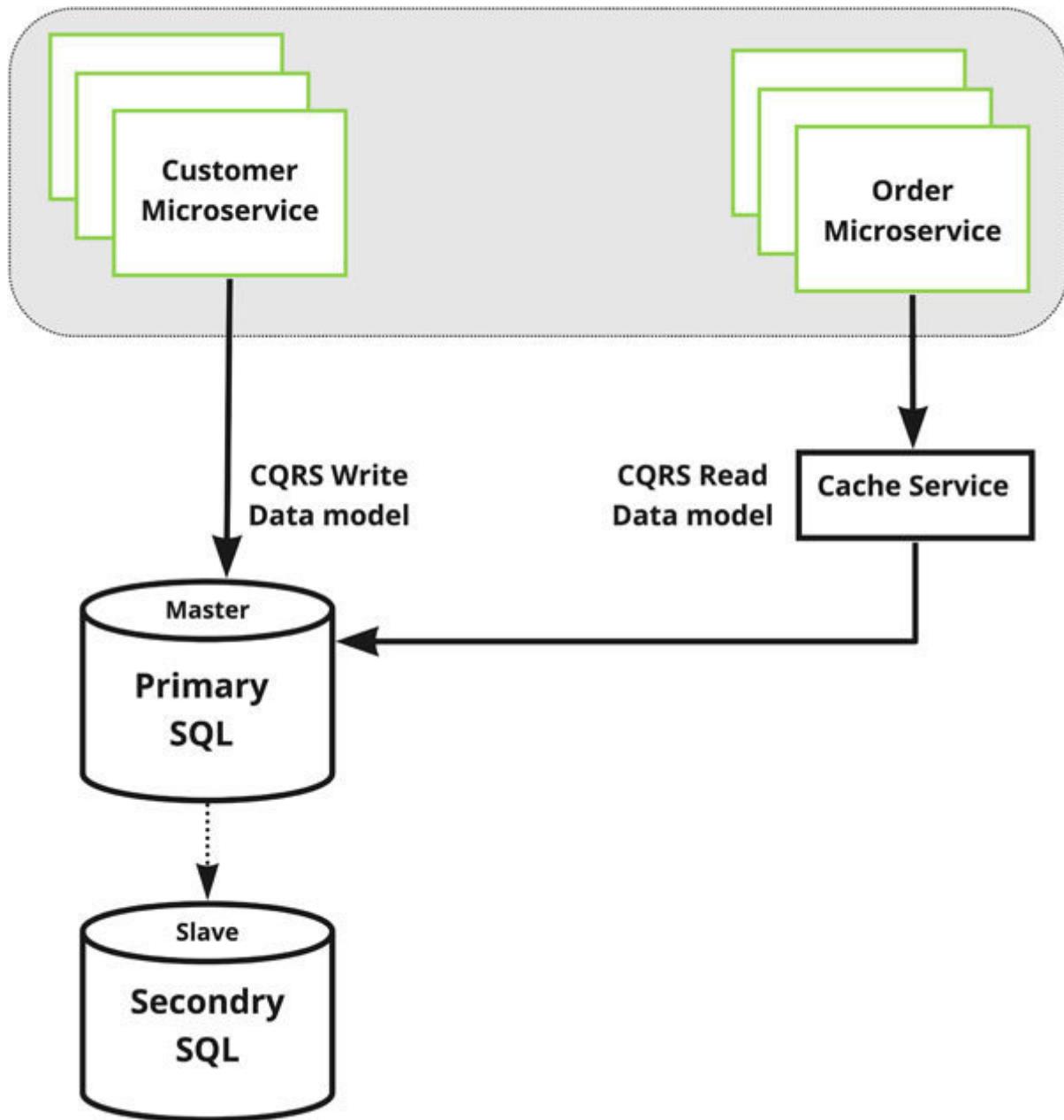


Figure 2.14: CQRS design pattern

It makes separation of read and write for complex business use cases. In the cloud native microservice architecture, database tables are mapped with services based on one database table per data domain model like order service has order database tables. There could be many one-to-one database tables per service. In

most of the business use cases, query can be built by joining many database tables.

In the CQRS pattern, the model will have the responsibility of inserting, updating, and deleting data from the database. Command has the sole responsibility of the database **Create**, **Update**, **Delete** action and queries are methods that are able to read and return data without any modification. CQRS defines a new standard for microservices to have separate read and write data models. These two different models are managed independently. It solves many complex business use cases where intensive query on multiple database tables is required.

For example, inventory synchronization happens between regional databases and connected stores by using the summary event aggregation which aggregates all inventory updates from different stores and then groups by store. This will calculate the inventory of that store accurately, which can be persisted to read databases and is available for specific query purposes. The CQRS pattern helps to process and make consistent data of inventory management among all stores and their regional database/data center.

Advantages

The advantages are as follows:

Faster reads on already processed/readymade data from different tables.

Easy to manage and update separate databases.

Can read data from multiple microservices and store in read data models for specific use cases asynchronously.

It's very similar to a persistent view.

CQRS database patterns take full advantage of modular and independent cloud native microservice.

Simplify complex business logic and data validation logic for separate business models.

Databases can be optimized and normalized in a better way.

Databases can be easily scaled for read and write separately.

Better granular level of security on database read and write

Limitations

The limitations are as follows:

It's not Git for simple and small-scale applications because it makes it more complex to implement and manage.

Data consistency issue: CQRS does not handle eventual data consistency at various read and write databases. Event sourcing manages data consistency using **System for Automated Geoscientific Analysis** design patterns. That's why both are used together.

Use cases

The use cases are as follows:

High-end applications of e-commerce like inventory management and price synchronization among stores and regional/central databases.

Use event sourcing with the SAGA design pattern for faster, scalable, and secure databases for microservices. Both patterns complement each other.

If there is a big difference between reads and writes. Based on usage read and write databases can be scaled individually.

For complex high performant apps where performance and user experience are critical.

SAGA design patterns

This pattern guarantees data consistency across multiple microservices in a distributed environment. Traditional monolithic applications supported **two-phase commit** of RDBMS databases. 2PC doesn't support and guarantee data consistency across multiple databases for multiple microservices.

The SAGA design pattern is about implementing transaction rollback and compensation controls to guarantee consistency in a series of atomic transactions that constitute a business process. Refer to the following link:

<https://microservices.io/patterns/data/saga.html>

These are some of the popular coordinating techniques with SAGA design patterns:

Orchestrating microservices

Choreography microservices

Hybrid communication

Orchestrating microservices

Microservices are a lightweight set of related services for a particular domain with bounded contexts. In real-life business scenarios to complete a single use case, multiple microservices interact and depend on each other in a distributed environment. We need a systematic mechanism to communicate and orchestrate these distributed microservices.

In orchestrating microservices, a **central controller** works as an orchestrator and handles all interactions between microservices. It works on a **request-response model** because responses of one microservice could be requested for other microservices because they may be dependent on each other. Each microservice has the business logic to call other dependent microservices. In other ways, it's hard coupling with each other.

Orchestrating microservices handle all routing and maintaining state of services to complete a single use case like online transfer of funds to another bank account.

In this use case, multiple microservices will interact for these high levels of sub-tasks which will be performed by individual microservices:

Log in to internet banking

Authorize logged in user

Get bank balance and other user details

Send fund transfer request

Perform validations

Bank balance of accounts updated

Confirmation of payment

The following diagram depicts the orchestrator design pattern where a single orchestrator orchestrates/manages three microservices: and The client interacts with this orchestrator by messaging the broker using the command and reply channels topics:

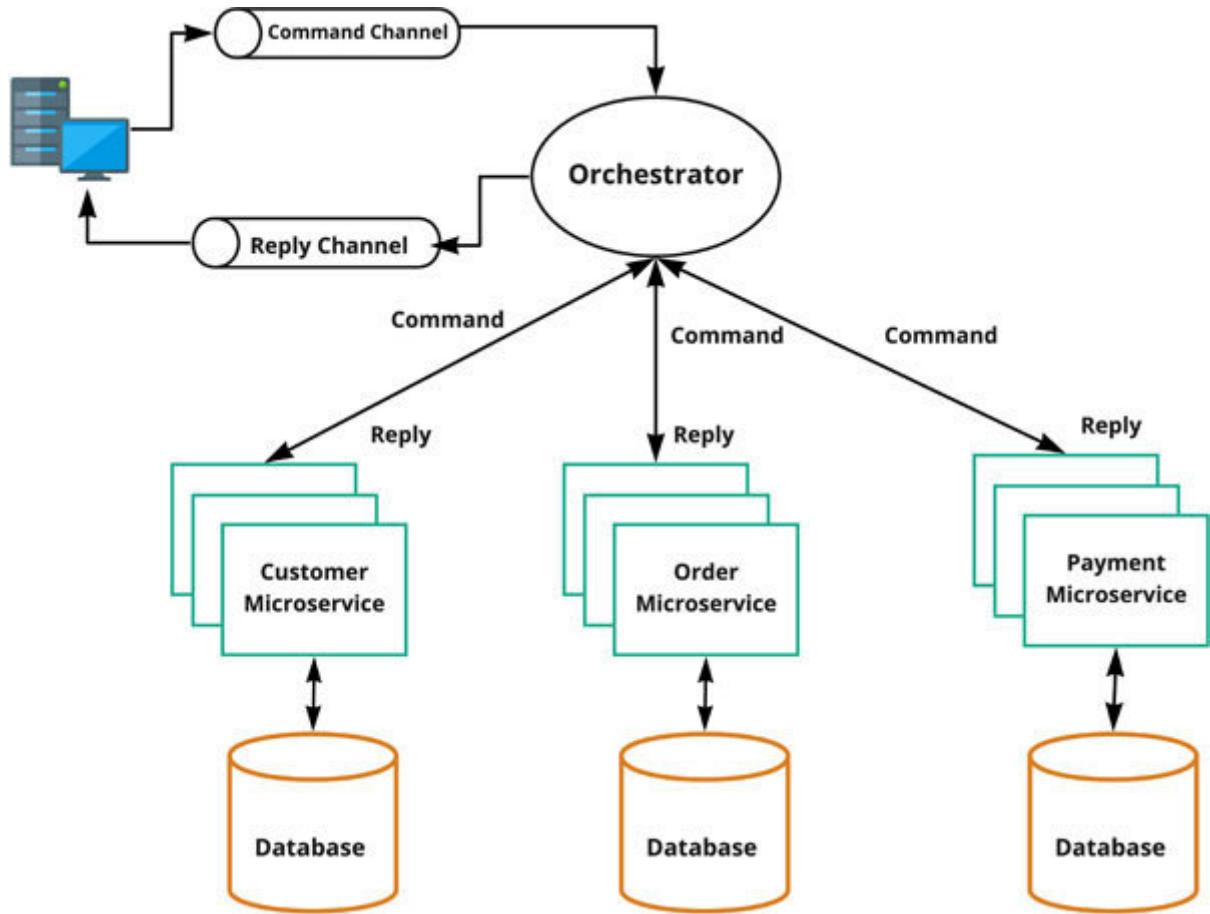


Figure 2.15: Orchestrator design pattern

Messaging topic is recommended between the clients and orchestrator layer to provide high availability. There are two topics needed in this design; the *command* channel to send requests to the orchestrator and *reply* channel for the outcome.

Advantages

The advantages are as follows:

Easy to implement, manage, and debug because all the business logic is handled at orchestrator (front controller).

It works on the front controller design pattern where a single controller manages all microservices communications.

It is based on popular request/response models. It works well for synchronous use cases coordination.

Limitations

The limitations are as follows:

Single point of failure. No service communication will happen when the orchestrator will be down with some reason.

If one microservice will have some issues, then dependent services will not work.

Low visibility because all the business logic will be controlled inside the front controller.

Choreography microservices

It's another way of inter-communication method of microservices, where all microservices will run independently and interact with each other by a set of events. It works on event-driven **reactive design patterns** where events will be generated after completing a task and other microservices which are interested in this service will use this generated event and take action. Here, services will be decoupled and work as a source of events or an event sourcing model. All microservices will interact with each other asynchronously.

This method has better advantages over orchestrating microservices which are dependent on each other and has a single point of failure limitation. The following diagram depicts communication between the customer, order, and payment microservices using event-based send and listen messaging topics. There is a final confirmation topic, which is being used by the last service when the transaction ends. It notifies to start microservices and confirm when data commit happens at this point of time:

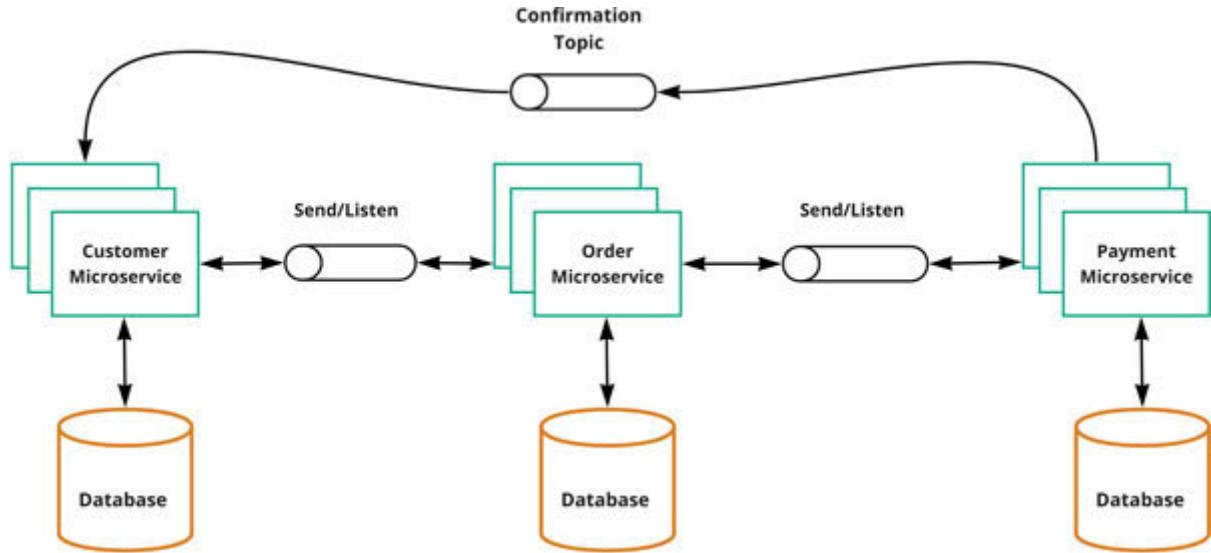


Figure 2.16: Choreography microservices

An event is a completed independent task.

Example: Like an order item has been added to the cart, the order has been created, payment has been processed, and so on.

Advantages

The advantages are as follows:

Decoupling of services—microservices.

Easy to maintain, test, and debug.

No single point failover because microservices are not dependent on a central controller like an orchestrator.

Easy add and update microservices. They can be easily added/removed from the event pipeline.

Faster response because no direct dependency on other microservices

Events can be stored, which is called **event sourcing** and supports SAGA.

CQRS and patterns can be applied for separate read and write operations.

Limitations

The limitations are as follows:

Cyclic dependency of microservices.

Complex to understand the flow and maintain business logic because it works on reactive models using publisher and subscriber models.

It's challenging to debug for errors, maintain, and rollback transactions.

Dev team and architects still prefer the traditional request-response model over the async event-driven model.

Hybrid communication

Practically in real-business use case scenarios, sync, and async both kinds of communication have been used based on their use cases. Hybrid communication when orchestration and choreography both have been used together for sync and async use cases.

Validation-related services like login authentication and authorization services are synchronous in nature; they can't wait from the client; however, order creation and payment microservices are async business processes. They are event-driven using messaging systems and can be communicated through event topics.

Advantages

The advantages are as follows:

No-cyclic dependency of microservices.

Controller orchestrator services maintain the flow of business processes synchronously. Easy to manage centralize business processes.

Microservices communications happens through event-based messaging brokers on the send and listen model asynchronously.

Other benefits of orchestration and choreography.

Limitations

The limitations are as follows:

Complex to design and develop. It has more overheads.

Centralized orchestrator microservice is like a single point of failure and considered as monolithic which can't be scaled.

Complex to debug and understand because monitoring is a little complex. Need additional tool to trace microservices communication logs

Use cases

The use cases are as follows:

Data consistency across multiple microservices and databases.

ACID transaction management in a distributed environment.

Final conclusions

Both microservices communication models have pros and cons. There is enough detail to choose based on the business use cases. You can consider less complexity, easy to manage, and maintain principles while providing solutions.

Most of the organizations are adopting the event-driven architecture with high-end faster environments to avoid any latency between transactions of a single business process, like end-to-end order of any product on the e-commerce online portal.

Event sourcing and an event-driven architecture

This pattern complements the microservice architecture because this is becoming the de-facto standard for developing microservices for most of the use cases. This pattern is based on **async event**

Introducing an event

An event is an occurrence or change or a short-lived task in a microservice application/system. It's like push notification services, or alerts, and so on.

Some of the examples of events are as follows:

Order is added in cart

Order is submitted

Payment is confirmed

order is delivered

cab is booked

The following diagram shows the event-driven architecture using messaging event-based brokers. Here, the Order microservice publishes the event for **Messaging** which will be consumed by the **Catalogue and Customer event consumer**.

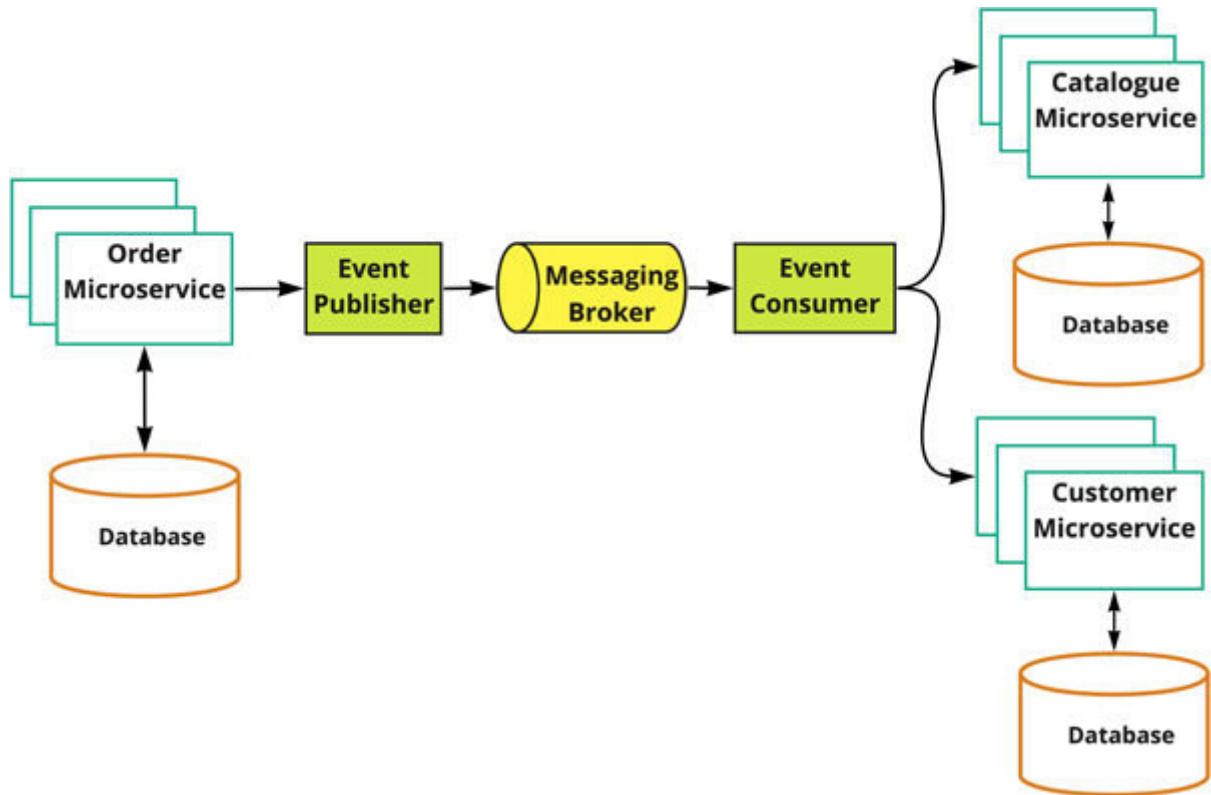


Figure 2.17: Event-driven architecture

Event sourcing generates events for any changes in the microservices applications, which is listened to by other microservices and performs read/write actions accordingly. Events are the smallest key trigger points for any system changes in any microservices apps. These events can be persisted, queried, filtered, and processed individually.

Event sourcing persists and maintains the state of business use case-related changes. In this pattern, an entire use case can be built using a set of events. This works on the publisher and subscriber model, where one microservice generates events to producer microservice(s), other consumer service(s) listen to producer services, and performs business CRUD operations.

Sometimes, they can be replayed when any transaction fails during inter-microservices communication. These events are persisted into **event store** database or messaging service. It also maintains a sequence for the business flow-related use cases. An event store works as a service broker, which is being used to listen to other microservices in the pub/sub model.

Event sourcing with SAGA

In the event-driven architecture, there are multiple services that work together asynchronously with the pub/sub model. Also, each microservice follows bounded context with its single responsibility and separate database tables. This makes it very complex to maintain data/transaction consistency. That's the reason why the event-driven design works better with SAGA design pattern to maintain data consistency.

Advantages

The advantages are as follows:

Easily scalable on demand and manageable.

Resilient and fail independently without affecting other services.

High performance.

Producers and consumers are decoupled.

Support streaming use cases where consumers will not subscribe to producers. In this scenario, all events store in the event log and any service can consume from any portion of this event log, it can also replay, and so on.

Limitations

The limitations are as follows:

API response may be delayed.

Complex to design and implement in a distributed system.

Complex to track, monitor, and manage changes in a distributed system.

Skill set challenges due to use of heterogeneous technologies.

Doesn't maintain data eventual consistency between various microservices and databases independently.

Use cases

The use cases are as follows:

Parallel processing in multi-tasking highly available in a distributed environment.

Async use cases for high-end system like an e-commerce system in peak time.

Streaming data for IoT, fraud detection, payment processing, website monitoring, stock market, analytics, and real-time marketing.

Taxi booking system like Uber is the best to find out the nearest cab/taxi to the customer and message alerts to the driver/customer after successful booking.

Push notification, alerting, and so on.

Observability and monitoring patterns

In a cloud native microservices environment, it's very tough and tedious to monitor hundreds and thousands of microservices and their scaled containers. There are so many moving parts, so we need observability and monitoring patterns to manage these humongous microservices traffic and their inter-communications.

Let's discuss some of these important patterns in detail:

Distributed tracing

Log aggregation

Health check API

Application metrics

Distributed tracing

When multiple microservices interact with each other for various business use cases, there may be a possibility of failure of services during this inter-communication, which can break the business flow and make it complex and tedious to identify and debug the issue. Logging is just not enough; it's a very tedious process to read logs and identify issues when we have thousands of lines of logs of multiple microservices on different containers in a multi-cloud environment. In some cases, the same microservice is deployed on multiple clusters and data centers.

The following diagram has two microservices such as **Customer** and **Order** which create and persist tracing logs and push to the centralized tracing service:

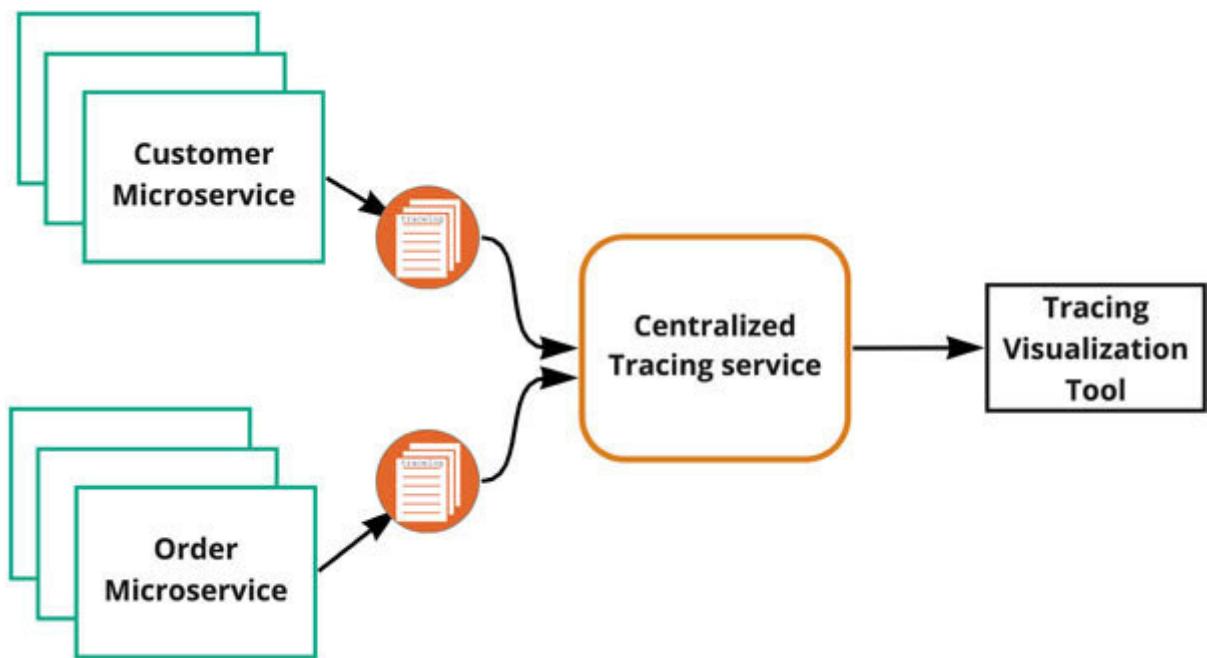


Figure 2.18: Distributed tracing

We need a mechanism to manage, monitor, and debug the production issues. This pattern is based on the REST API tracing to quickly track the issue of any culprit and buggy services. In this pattern, client request and response logs for the microservices REST API are recorded and monitored asynchronously.

This tracing can be done by various techniques; common **correlation ID** is one the popular techniques where end-to-end API calls from the client/UI to backend services are done by a unique correlation ID. Every external client request assigns a unique request/correlation ID which can be passed to all subsequent services. In this way, tracking can be done easily by persisting all request-response payload.

Advantages

The advantages are as follows:

Customize and visualize a centralized web UI dashboard to track and observe all microservices, environment and databases, and so on.

It's also does profiling of the application and checks performance.

It filters requests using many business logic and data points.

It shows all tracing data from different clusters and multi-cloud on a single dashboard. It's easy to manage and monitor from a single pane of the glass.

Customizes query on these API tracing logs by the dashboard and APIs.

It also tracks the request and response time and peak usage duration.

These traces are being collected automatically without any code changes.

Use cases

The use cases are as follows:

Microservices tracing for debugging production issues.

Application performance monitoring using APM tools..

Log tracing for apps, databases, message broker, Kubernetes container solutions, and other infrastructure systems.

Log aggregation

This is a pattern to aggregate logs at a centralized location. It's a technique to collect logs from various microservices and other apps and persist at another location for query and visualization on log monitoring UI dashboards.

To complete business use cases, multiple microservices interact with each other on different containers and servers. During this interaction, they also write thousands of lines of log messages on different containers and servers. It's difficult to analyse humongous end-to-end logs of business use cases on different servers for developers and DevOps operators.

Sometimes, it takes a couple of tedious days and nights to analyze logs and identify production issues, which may cause loss of revenue and customer's trust, which is *MOST* important. Log aggregation and analysis are very important for any organization. Outage of applications can drop company's shares down.

The following diagram has two microservices, which writes logs locally/externally and finally all logs aggregated and forwarded to the **Centralized Log Aggregation** service:

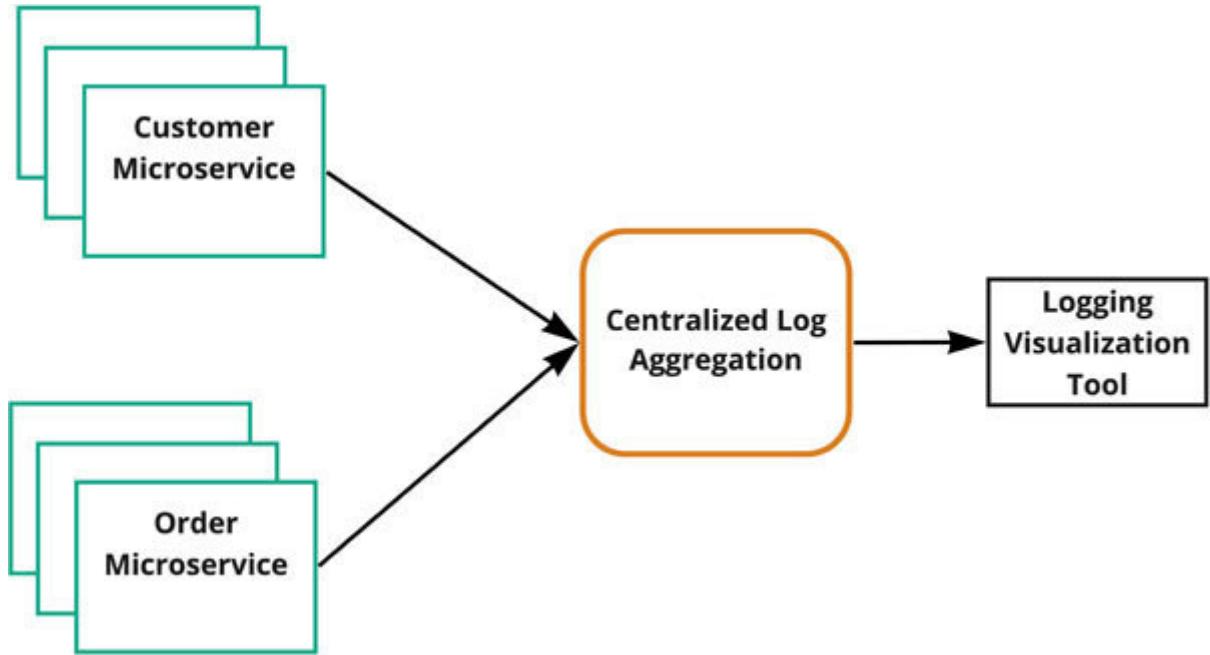


Figure 2.19: Log aggregation

There should be a mechanism to aggregate end-to-end logs for given use cases sequentially for faster analysis and debugging of logs. There are a lot of **open source** and enterprise tools which aggregate these logs from many sources and persist at the centralized location asynchronously at the external location which is dedicated for central logging and analysis. It can be on the same cluster or on the cloud.

It's important to write logs asynchronously to improve application performance of the application because in this scenario, the actual API/application response time won't be impacted.

There are multiple log aggregation solutions like ELF, EFK, Splunk, APM, and some other enterprise APM tools.

Structured logging can be done using various tools, which filter and format the unstructured log to the required format with various log filter tools. Fluentd/Fluent Bit and Logstash open-source tools are useful for making data structured.

Advantages

The advantages are as follows:

Easy to debug and identify issues with a unique correlation ID quickly.

Stores structured and organized logs for complex queries for a given filter condition.

External centralized log's location to avoid overhead of extra storage and compute.

Central logging and analyzing asynchronously.

It makes application faster to log async logging.

Use cases

The use cases are as follows:

Good integration with distributed microservices.

Async logging on an external server using the file system or messaging topic.

Log analytics, dashboarding, visualization graphs, and so on.

Health check API

When multiple microservices run on different clusters and server nodes, there may be use cases where the server is up and running; however, the application is down. The challenge is that there is no easy and automated way to identify this. In this scenario, apps show healthy status to infrastructure monitoring tools; however, they might be down. Either someone will report manually or an auto-bug created by reading application logs.

It keeps on checking the health status of running of microservices applications and accordingly acts for remedy. This is very important when an app is running on bare-metal and VMs where the app can't be restarted if it's down due to some app issues like memory leakage, heap issues, database connections issues, and so on. Load balancers will not know which server does not respond to client responses because it pings only the server and gets a positive response. It keeps on checking the running microservice/monolithic application status by load balancer and other infra tools to make the right decision at the right time and re-start the app or re-route requests to other servers.

There are various methods to check the health status of microservices apps, which will keep on checking the status by hitting the health check REST APIs, which are exposed from applications.

Advantages

The advantages are as follows:

It forwards client requests to other servers if the application is down.

Service registry can reshuffle and re-route for only healthy apps APIs.

Use cases

The use cases are as follows:

Identify app issues in running microservices by a load balancer.

Service registry management for healthy services.

Integration with an app monitoring dashboard.

Generate alerts for any app issues.

Send an email/push message notification to stakeholders for any app issues during odd hours.

Application metrics

This is a way to check application metrics, their performance, audit logs, and so on. It's a measure of microservices applications/REST APIs characteristics which are quantifiable or countable. It helps to check the performance of the REST API like how many requests an API is handling per second and what's the response time. It helps to scale the application and provide faster applications to web/mobile clients. It also checks **Transaction Per Second** and other metrics of applications.

There are many tools available to check matrices of applications like Spring Boot Micrometer, and APM tools like and so on. They work either with push or pull models using REST APIs.

For example, Grafana pulls metrics of the applications by using the integrated Prometheus REST API handler and visualizes on the Grafana dashboard.

Advantages

The advantages are as follows:

Helps in scaling hardware resources for future.

Identifies the performance of REST APIs.

Identifies Return of Investment

Monitors and analyzes application behaviors during peak and odd hours.

Limitations

The limitations are as follows:

It needs extra hardware like compute, memory, and storage.

It slows down the performance of applications because it runs with applications and also consumes the same memory and computes with the speed of a server.

Service discovery design pattern

It's easy to inter-communicate with monolithic application services because they are all bundled in the same application and deployed as a bundled service on the same server. So, discovery of services is easy in monolithic applications.

In the microservice architecture, when many microservices are running in a distributed environment on multiple containers/servers. It's a complex process to discover and communicate with each other because they may have different dynamic IP addresses, when they scale and create multiple containers for the same app. Sometimes, the same microservice application is running at multiple places in multiple containers, clusters, or VMs servers. There should be a way to register all services at one centralized place, which helps client requests to discover and redirect to REST API services in a distributed environment. Another challenge what we see here in container platforms is that when any container is down, then it creates another container and deploys the same application as a separate instance.

This pattern is divided into two parts:

Client-side discovery

Server-side discovery

Client-side discovery

This discovery happens at the client side. In this pattern, all microservices are registered first with the **discovery server**, where all services are listed with server details. In this pattern, there will be two API calls. First, the client request will connect to the discovery server, get server details to the server which it wants to be connected to, and then call the actual API service directly in the second call.

In the following diagram, all three microservices have their own registry clients. They all are registered to the discovery server. The order microservice will check the **Catalogue** microservice server details with the first API call to the discovery server and then make a second API call to the **Catalogue** service. This discovery is at client side:

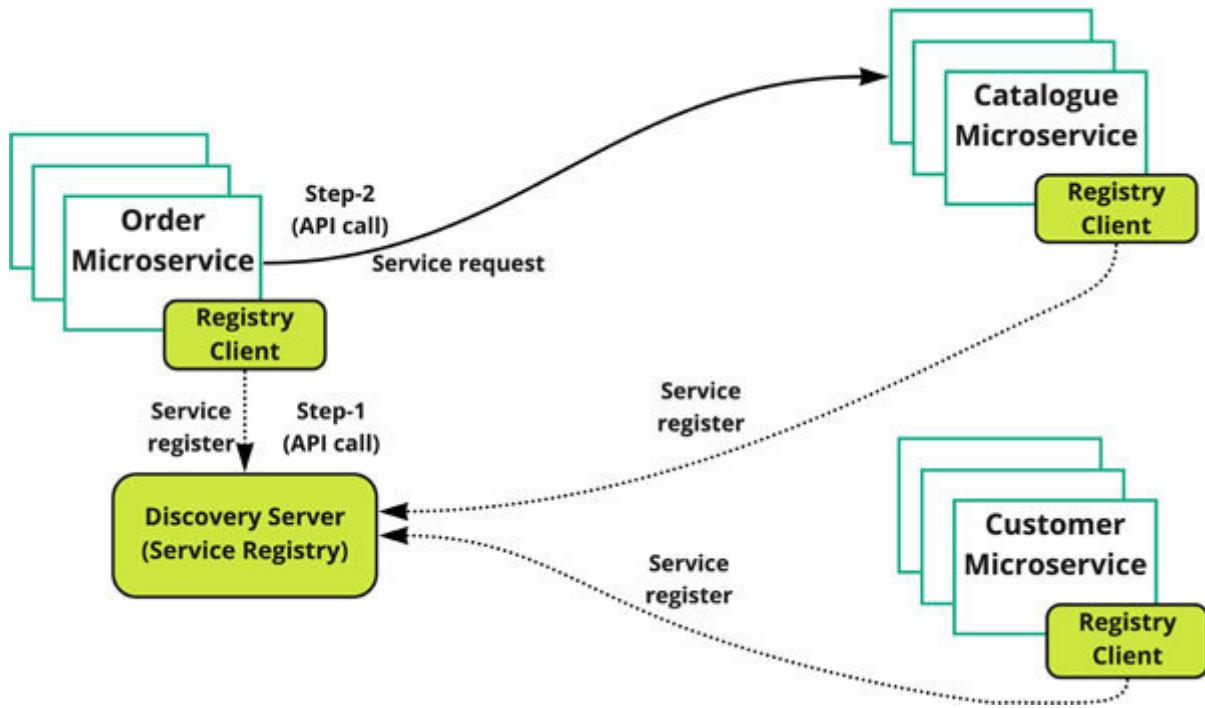


Figure 2.20: Client-side discovery

There are multiple open source and enterprise solutions for the client-side discovery. **Spring Eureka** and **Ribbon** open sources can be used for implementing the client-side discovery. You can also use the enterprise **Consul**

Advantages

The advantages are as follows:

No load balancer support for services.

It's simple to implement programmatically by developers.

Easy to maintain and update by the UI dashboard.

Limitations

The limitations are as follows:

Two round of service API calls for services, which makes it a little slow.

Dependency on the discovery server which could be a single point of failure if it doesn't scale.

Tight dependency on the application source code; it's hard to manage complex configuration code with code. It makes the code messy.

Code has to be tested for any changes for discovery.

Use cases

The use cases are as follows:

When client services want to connect backend REST API services.

Backend microservices want to interconnect with other microservices.

Server-side discovery

This server-side discovery happens at the backend infrastructure. In this case, all microservices are registered at the server-side registry. The load balancer is a good example which forwards client's requests to register services with it. Microservices can be easily connected with each other by using a single API call.

Let's assume service **A** wants to be connected to service Client **A**. Client **A** will not get the server details of service **A**. In this use case, client **A** will send a request to the load balancer at the server side; the load balancer will have its own service registry, where all services are listed with their addresses. The load balancer redirects requests to instances of microservice **B** using various routing algorithms like **round robin** or **weight** and so on. The discovery server plays an important role in this server-side discovery.

In the following diagram, the service discovery takes place at the server side using the load balancer and with its own service registry. The order microservice client requests the **load balancer** to connect to the **Catalogue** service. LB gets its address from the service registry and redirects the request to the **Catalogue** service in a single call:

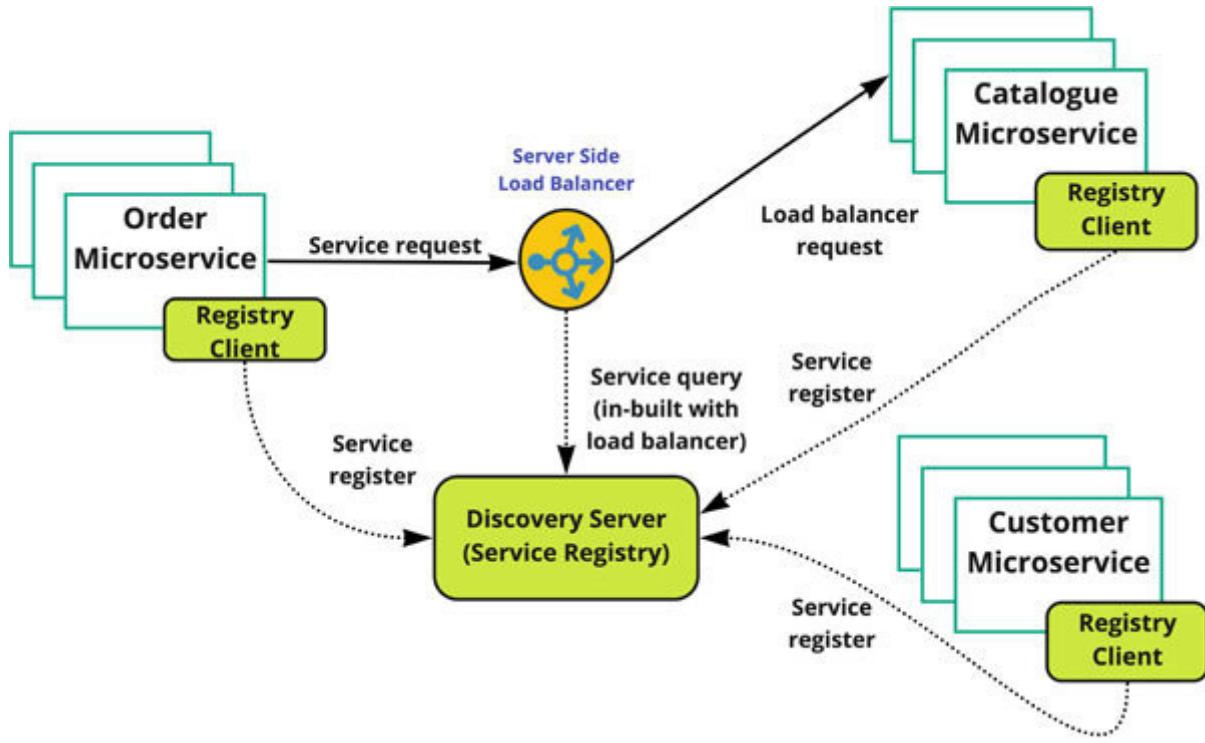


Figure 2.21: Server-side discovery

It can be implemented by using various load balancer tools/APIs like AWS ELB, Google Load Balancer, and so on.

There are other server-side container orchestration tools like Kubernetes, which manage this with their own load balancers at the containers end.

Advantages

The advantages are as follows:

This intercommunication between microservices happens at a single API call which provides better performance.

Faster performance because it takes lesser time to discover with a single API call.

Limitations

The limitations are as follows:

Dependency on the load balancer side.

Still have the server-side responsibility to update the service-side registry at the load balancer side.

Use cases

The use cases are as follows:

When client services want to connect to backed services.

Backend microservices want to interconnect with other microservices.

Circuit breaker design pattern

The major challenge of having many moving parts of microservices is that if one service is down, then the client apps keep on hitting the same service without knowing that the backend microservice has some issues. Other challenges are when microservices **A** and **B** will try to connect to microservice there may be a possibility that service **C** has some app/database side glitches. In this case, service **A** and **B** will keep on hitting service which can create cascading failure issues for other services **A** and **B** which depend on service These challenges create an overhead of occupying extra memory heap/thread and CPU compute speed, which slows down other consequent client requests too. This issue creates traffic jams for all the incoming requests. Sometimes, producer services are down due to the continuous hit by producer services, which could create a bottleneck between services. Sometimes, it may be a network failure where microservices are hosted. These service failures are intermittent and need some extra time to recover from the services.

The following diagram depicts all the states of the circuit design pattern and connectivity between **Order** and **Catalogue** microservices:

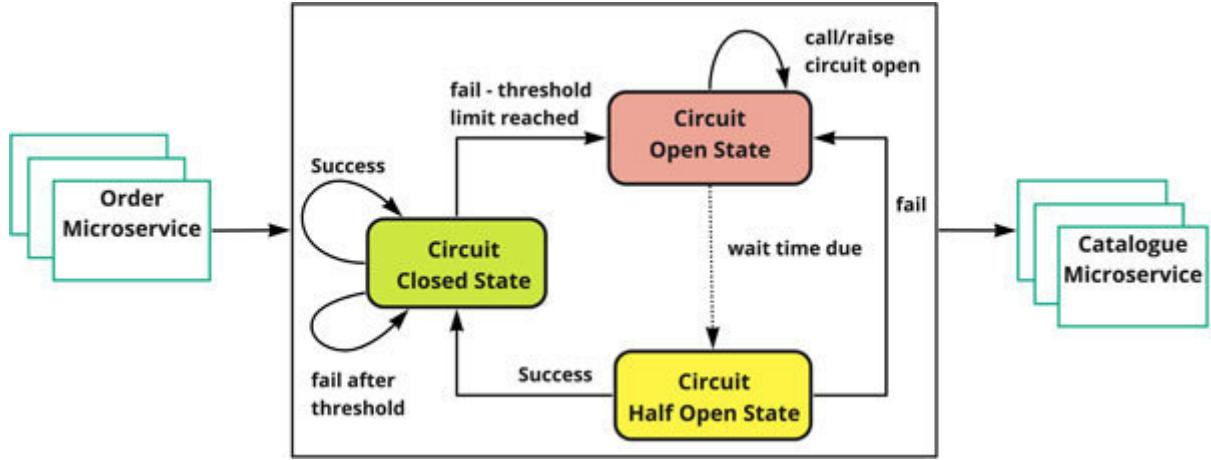


Figure 2.22: Circuit breaker design pattern

This design pattern is also called fault tolerant service, which avoids services faults during runtime on production servers and makes it more resilient.

Now, we will try to understand how it works internally:

Most microservices fail or tend to fail at some point of time due to many issues like application, databases, network, and other environments and servers. Circuit breaker patterns handle this service failure gracefully with a backup strategy. They never terminate client requests abruptly, even when the producer service is down.

It also keeps on counting the number of client requests from the consumer service and takes an alternate approach. The circuit breaker has a dedicated service, which monitors and counts the number of service calls to the faulty service and breaks the circuit if it reaches the threshold/maximum number of attempts. Timeout

can also be set for consumer/client services to break the flow if any anomaly at the producer service ends.

For example, in this use case, client service **A** will keep on trying service **B** for 10 times on one second interval. If it still doesn't get any response from service then service flow from **A** will be disconnected after reaching the threshold. It also has a mechanism to take backup and preventive measures to restart the service and give a nice error message to clients or redirect requests to other servers.

Circuit breaker design pattern has the following three states:

It's for happy use cases. If there is no issue in microservices, the circuit breaker maintains this closed state, which allows traffic between microservices and clients.

It's for negative use cases when the producer service is down. If the number of service calls from consumer to producer services crosses the threshold limit, then the state of circuit breaker changes to open and blocks all traffic from consumers to producer microservices. It allows some time to restate producer services; sometimes, these issues are intermittent which don't require restart, changes, or re-deployment of services.

The circuit breaker has a mechanism to half open communication between services after a timeout period to check whether an issue

still persisted. It switches to the half open state when any service fails, and resets back if it succeeds.

Advantages

The advantages are as follows:

It makes REST API microservices sustainable, reliable, and resilient after failure. Sometimes, services have intermittent issues, which can be fixed quickly. It handles this kind of scenario.

Gracefully handles failures with proper workaround/alternate action and gives some time to microservices to be ready. If the backend service is down, it redirects to the caching service to fetch cached responses.

Helps to monitor service intercommunication, transactional logs, and status.

Changes its status based on response from producer service(s).

It handles downtime and provides resilient service to recover fast.

Limitations

The limitations are as follows:

It's a little difficult to presume the timeout period.

It adds extra latency in the response time of the API.

It is tightly coupled with the source code. The service has to be re-tested after any config changes.

Use cases

The use cases are as follows:

Inter-communication of microservices APIs on different clusters, on-premise, and multi-cloud.

Handle fault tolerance of microservices.

Integration with various external cloud or third-party APIs.

External configuration

In the monolithic world, all services are bundled in a big fat application, which uses the common configuration. However, in the microservices world, there are multiple small microservices across different servers and containers. It's challenging to maintain configuration within the microservices code because many services use common configuration. If you make any common configuration changes in one service, other microservices should be also changed. For example, if the database credentials have been changed, it has to be updated to all dependent microservices. Also, it's hard to manage and change the configuration within the source code because in this case, configurations are tightly coupled with microservices source code.

The following diagram depicts the centralized configuration server where all configurations are stored. All these three microservices access these centralized configurations. Any change at the **Config Server** will be reflected in other services:

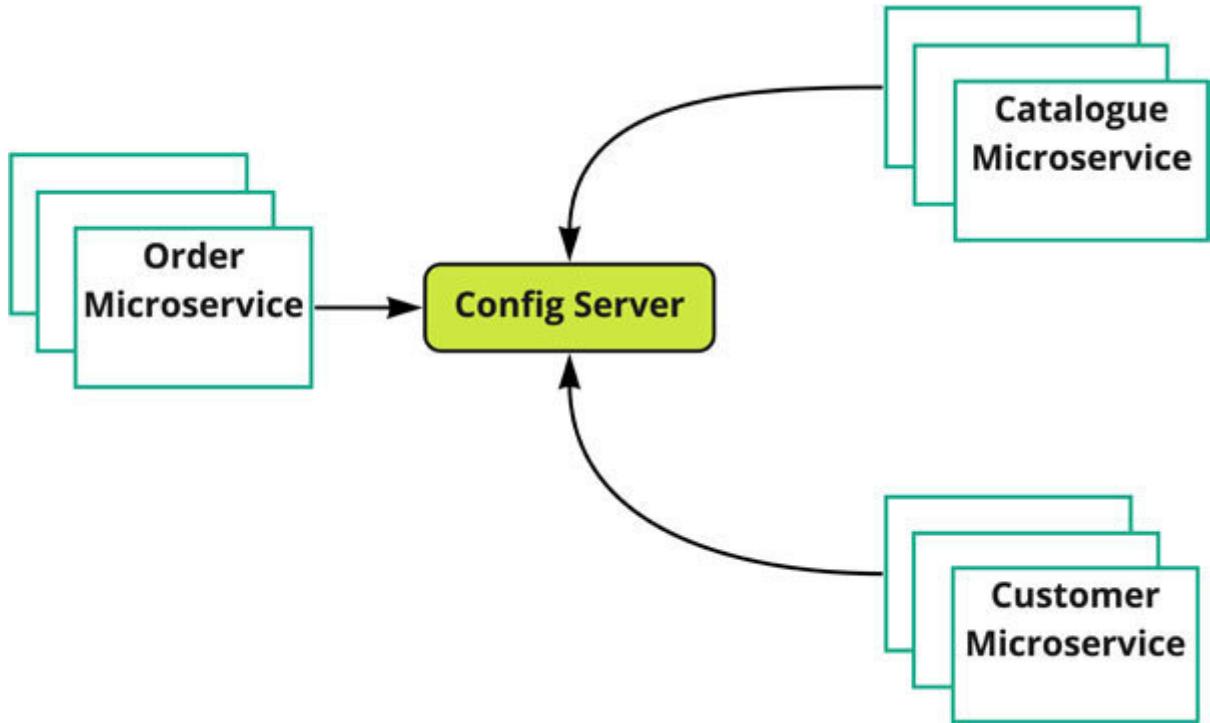


Figure 2.23: External configuration

This external configuration pattern solves these challenges. In a microservice environment, this pattern provides a centralized dynamic configuration service outside the source code of the microservices applications, which can be accessed from any microservices as a global configuration and can be easily maintainable.

For example, database connection details, messaging, dynamic parameter values, and external APIs stored in external servers. It also externalizes secrets like passwords and certificates, and so on.

Advantages

The advantages are as follows:

Extracts cross cutting concerns from the source code to the config server.

Externalizes configuration from apps to a separate server and source code repository.

Apps can run separately without any code changes if the config server has any change.

Config changes can be done anytime without any impact on any app and the same config changes will be reflected dynamically to all dependent microservices.

Limitations

The limitations are as follows:

A little challenging to map apps and config servers.

Dependent apps should be aware of any changes in the centralized config service to adopt.

Use cases

The use cases are as follows:

Externalize secrets and database connections details.

Externalize cross cutting concerns from the source code to another centralized server.

API token security

It's very critical to provide authentication and authorization security to the microservices apps, when they communicate with each other. Mostly, microservices are exposed as REST APIs in distributed environments. REST APIs are stateless. It's challenging to maintain client sessions with backend REST APIs.

A strong API security mechanism is required to communicate between consumer and producer microservices. This API token security pattern provides a better **Authentication & Authorization** service and maintains a session between client and backend REST APIs. For example, web/mobile clients connect to the API gateway first, which is a single point of entry of backend REST API microservices.

The following diagram depicts how to get the API token by passing the access token to the authorization server which could be JWT/OAuth and get the API token to access backend APIs. Web/mobile clients will add this API token to the REST request call in the form of an HTTP request header and pass it to other backend services after A&A:

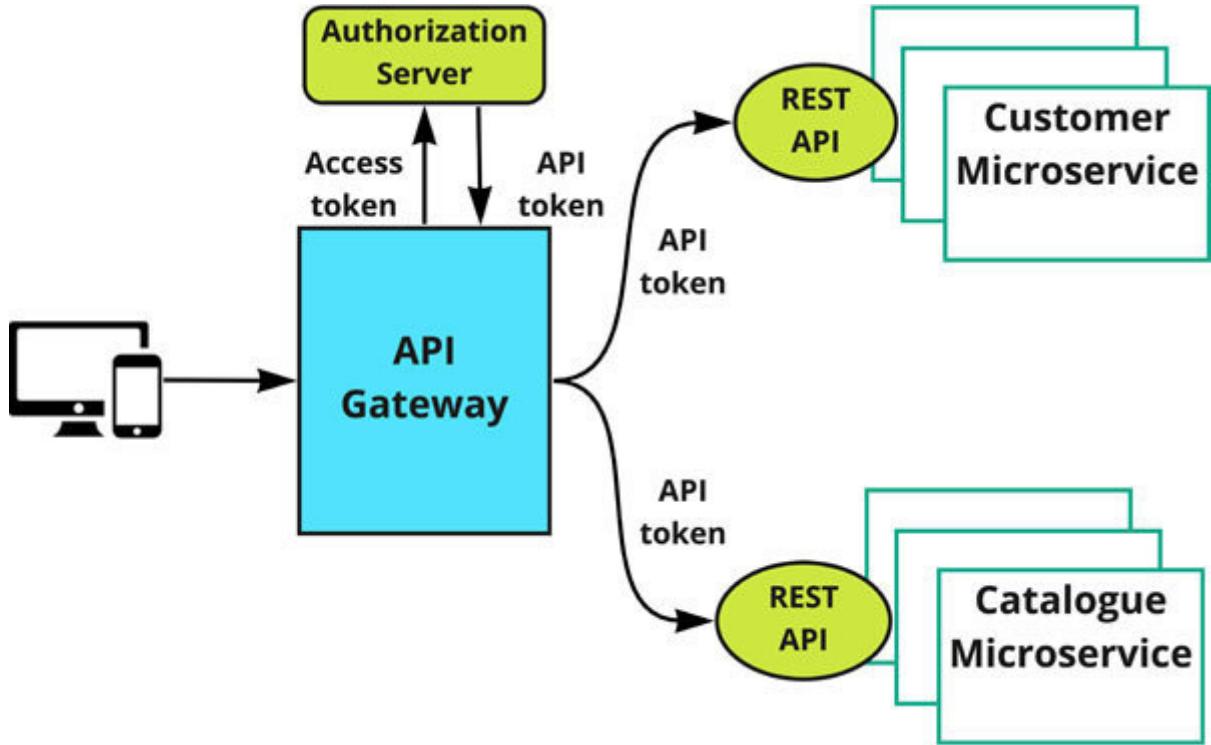


Figure 2.24: API token security

There are multiple open source and enterprise solutions to implement the API token security. OAuth and **JavaScript Web Token** are popular, which create encrypted tokens and encrypt other information like user details, roles, and other key-value pairs. There is a provision to also add the expiry time of the API token. This token is generated by the server-side A&A service on demand. It can be added to a client's service REST API header during the inter-microservice communication. This token is authenticated and validated for the expiry time on every request and then authorized to connect to backend business services and databases.

So, with a design pattern, only authorized services can be granted access to microservices APIs.

The API gateway is a recommended way to expose all backend microservices to client services like mobile, web, and other clients. It provides a single point of entry to the backend APIs. It uses this API access token internally to get access. It internally integrates with the access token API.

Advantages

The advantages are as follows:

Secures inter-communication between microservices.

Provides secure and strong encrypted authorization.

Also provides user roles, which can be checked and accessed at the backend authorization service.

Tokens can be encrypted with a strong encryption algorithm and expiry time.

Access token can be also created to access a new security token, if it's expired.

Use cases

The use cases are as follows:

User login authentication.

User role authorization like admin, read only access, and so on.

Use the API gateway with microservices.

Integration testing

In the monolithic architecture, it's easy to test end-to-end business use cases easily, because all services inside the same application source code can call each other easily without any communication issues.

In the microservices architecture, when multiple services are deployed on different servers, it's very challenging to test end-to-end business use cases because every microservice is built on a single responsibility in a bounded context.

Integration testing pattern allows us to build a separate test suite service which tests end-to-end business use cases, like when a customer buys a product and the customer pays for the order, and so on.

There are multiple ways to do this. One of the recommended ways is to use **Behavioral Driven Development** design pattern which works on *Given-When-Then* principles.

It's recommended that run this integration test suite during the test phase with DevOps CI/CD pipeline on the test non-prod server.

Advantages

The advantages are as follows:

End-to-end testing of business use cases like login and order payment, and so on.

Externalize this as a service and have use case based integration testing suite.

Run integration test cases on the test server before deploying to the production server.

Identifies inter-services communication issues between microservices.

These integration test cases can be persisted in the source code repository.

Limitations

The limitations are as follows:

It needs to be modified and redeployed if there are any changes in microservices.

You need to provide test data separately to test use cases.

Conclusion

In this chapter, we discussed various important microservices design patterns. We covered most of the real challenges when we migrate monolithic to microservices or design a new greenfield project with a bunch of independent microservices. We also discussed how to solve various real-life challenges with different design patterns, their advantages, and limitations. It's not necessary to apply all these design patterns when you design and develop any microservice project. We should be very thoughtful to use the right design pattern for right use cases because sometimes it may create issues and become an anti-pattern. We need to choose these design patterns based on use cases when solving business processes.

In the next chapter, we will cover how microservices can be designed with the **API First** approach.

Let's meet in the next chapter! Till then happy learning!

We covered all important use cases design patterns. There are other design patterns, which you can refer to online resources. We will give some references also at the end of this chapter.

Points to Remember

Microservice design patterns are generic to any programming language such as Java/Spring, .Net, C++, Node.js, Python, Scala, Angular.js, Go, React.js, and so on.

A monolithic application can be deployed on Kubernetes containers as a big application with large compute resources and hardware as a lift and shift model by making minor configuration changes to make it cloud native; however, these applications can't achieve the real value of containerization like scalability, high availability, and so on. The downsides of monolithic apps are not absolute. Many organizations run monoliths in production. Not all monoliths are bad and not all microservices are useful.

A topic is a logical pub-sub queue where events can be published and consumed. This is called async eventing.

Bounded context is a group of subdomains.

The strangler design pattern is best suited for migration from legacy monolithic applications to microservice.

A single page application is considered to be the best design for a front-end UI application and very popular at present.

A shared database is considered as an anti-pattern for the true microservices architecture because it breaks the principle of independent services per database pattern.

SAGA design patterns guarantee data consistency across multiple microservices in a distributed environment.

Microservice doesn't support **two-phase commit**

Orchestrating microservices handle all routing and maintaining state of services to complete a single use case like online transfer of funds to another bank account.

Messaging topic is recommended between clients and orchestrator layers to provide async high availability. There are two topics needed in this design: the *command* channel to send requests to the orchestrator and the *reply* channel for the outcome.

An event is a completed independent task.

It's important to write logs asynchronously to improve the application performance because in this scenario, the actual API/application response time won't be impacted. This is a recommendation also.

Service discovery is an important pattern to communicate one service by another in a distributed environment. These two are recommended ways to do this: client-side discovery and server-side discovery.

The circuit breaker design pattern is also called a fault tolerant service which avoids services faults during runtime on production servers and makes it more resilient.

The API gateway is a recommended way to expose all backend microservices to client service like mobile, web, and other clients. It provides a single point of entry to the backend APIs. It uses this API access token internally to get access.

It's recommended to run integration testing during the test phase with the DevOps CI/CD pipeline on test non-prod server.

Key terms

Atomicity, Consistency, Integrity, and Consistency

Two-phase commit

Relational Database Management System

Domain Driven Design

Continuous Integration/Continuous Delivery

User Interface

High Availability

Single Page Application

Authentication & Authorization

JSON Web Token

Behavioral Driven Development

Service-Oriented Architecture

Command Query Responsibility Segregation

Load Balancer

References

Microservices design

Micro-frontends design and other similar modernization

Micro-frontends

SAGA design

CHAPTER 3

API First Approach

Application Programming Interface (API) allows two apps/resources to talk to each other and is mostly referred for **Service Oriented Architecture (SOA)**.

API is gaining more popularity when microservices development is booming for modern cloud-native applications or app modernization. We can't imagine microservices without APIs, because there are so many distributed services in a microservice architecture, which can't be easily integrated without the help of API. So, *both Microservices and API compliments each other!* API first approach is a popular approach for exposing microservices for integration. It follows back-end for front-end (BFF) design principle.

This chapter will cover the challenges of using the legacy approach of **code first** design, how to solve these challenges using the API first approach, fundamentals of the API first approach, overview, specifications on how to design and test API. It will cover REST fundamentals and its components. We will also discuss the best practices of API and how to document dynamically with **OpenAPI**. We will discuss in detail how to design a standard API document using OpenAPI standard specification, tools to design and test API with mock tools.

Let's get started!

Structure

In this chapter we will cover the following topics:

Introducing API

API first approach overview

Current challenges with API design

Introducing the API first approach

Need of the API first approach

REST overview

Introducing REST and RESTful API

REST and HATEOAS support

Hypertext Application Language

API design principles

Benefits of API first approach

API components

API best practices

API security

Exposing API on communication protocols

API documentation and specifications

API documentation

API specifications

Planning API design

API specifications

OpenAPI Specifications

RESTful API Modelling Language

API management tools

Designing and generating API docs

Code First with OpenAPI 3.0 and Spring REST API

[Setting up Spring doc OpenAPI with Swagger UI](#)

[Generating API doc with JSR-303 validation](#)

[Generating API doc with `@Operation` and `@ApiResponses`](#)

[API first with SwaggerHub](#)

[Introducing Swagger](#)

[SwaggerHub UI overview](#)

[Testing API with SwaggerHub inspector](#)

Objective

After studying this chapter, you should be able to understand the API first approach fundamentals, benefits, and how it helps in designing microservices. You will learn about REST overview, API components, and relationship with **Hypermedia as the Engine of Application State** and **Hypertext Application Language**. This chapter will help you to understand API best practices, different industry-standard API specifications, API management tools, and how to document API dynamically with OpenAPI Swagger. It will also help to learn how to impose API security on distributed microservices, design API docs using

Introducing API

It's an architectural design specification, a set of protocols that provides an interface to integrate and talk different microservices/monolithic apps and databases with each other. API does talk about how external services can communicate with apps, not *how it works!*

It creates an integration contract between different apps/external clients with a standard set of rules and specifications. It's followed as a development practice for external clients/apps.

API is based on a *contract first* design pattern of development, where all developments happen around APIs specifications and protocols. Developers use the same standard practices across different microservices agile development teams.

The following diagram depicting to API integration with different kind of diagrams:

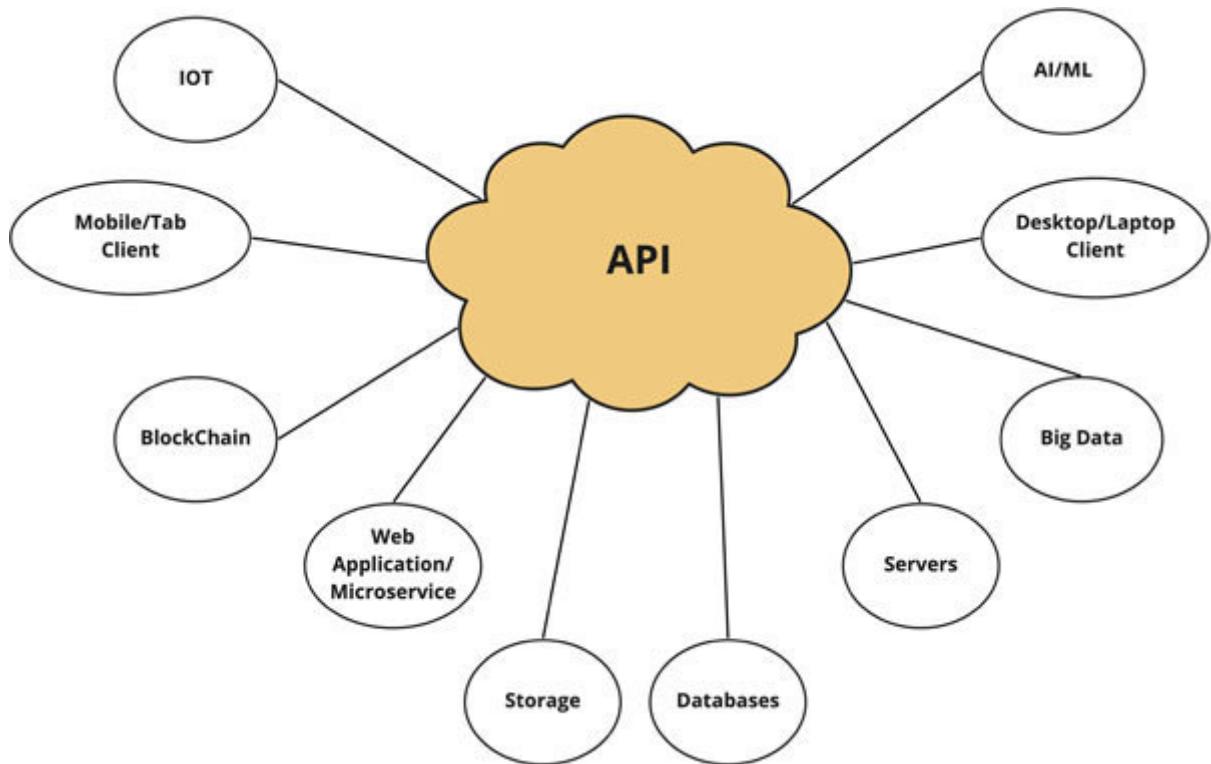


Figure 3.1: API integration with different software components

API first approach overview

In this section, we will discuss the API first approach in detail with its challenges, solutions, and benefits. We will broadly cover these topics:

Current challenges with API design

Introduction to API first approach

Need of API first approach

Current challenges with API design

The traditional approach was to develop an application first and then expose using API; which is now considered to be an anti-design pattern because it has a lot of integration challenges.

Also, API integration is difficult to manage manually for different kinds of client devices, because every web/third-party consumer apps have their own set of API request and response data.

Another problem of this approach is to maintain and test it for various kinds of clients. It's a very tedious process to have different API standard documents for different kinds of clients such as mobile, IOT, web browser, third party COTS, and so on.

Introducing API first approach

An API-first approach means, the entire design and development of microservices apps are based on APIs and it's treated as *first-class* API should be the center of all microservices integration between apps, databases, messaging brokers, and so on.

All agile project development processes should be around API first focused approach. The same principles should be followed by development teams. All business use cases of microservices should revolve around API.

The biggest advantage of API first is, it's a

Need of API first approach

The API-first design approach makes reusable, modular APIs. It also establishes contracts between services and various kinds of clients to talk to each other. It takes a good effort to talk to different business and technical stakeholders to design mutual API contracts. It should be designed based on the business use cases and the client's requirements.

If the API-first approach is considered at the start of a development project, then it avoids a lot of integration failure in the beginning itself.

Dependent APIs can be tested in the testing phase when they both are ready and deployed on test servers.

In API first approach, instead of starting with coding, the development team could start with design, planning, mocking API responses, and tests using those mocks at API designing phase using **OpenAPI** based tools without any actual deployment.

This API first design phase is very important, where the API contracts and specifications are defined after review meetings and feedback of different stakeholders like a business owner, business analyst, architects, development, performance, and testing teams.

OpenAPI is a set of API specifications. We will explain this later in this chapter.

REST overview

Let's go through the RESTful services and its related concepts. We will learn these REST related concepts:

What are REST and RESTful API?

REST and HATEOAS support

Hypertext Application Language

Introducing REST and RESTful API

Representational State Transfer is an architectural pattern to build and expose distributed web services. REST allows microservices to communicate with each other. It's not tied to HTTP communication protocol. However, most common REST implementations use HTTP. It helps client apps to talk to server-side web services which are exposed as REST API:

It's stateless, it doesn't persist client-server requests at the server-side. In the REST approach, it is the client's responsibility to maintain client-server requests/states. There are different ways to maintain this state at the client-side, some of them are sticky session, secure token, browser-side cookies, and so on.

It's faster than SOAP web services. The bigger advantage is, it's not related to any source code language. This can be implemented in any programming language and communicates with cross-programming languages like Java-based microservices can talk to .Net services using REST over HTTP. REST exposes web services as URI, which can be consumed by any apps or external clients like web browser/mobile/IoT devices, and so on.

RESTful API is a service that follows REST architectural principles and HTTP protocol standards. It's a platform agnostic interface. REST API provides a simple, flexible, dynamic, and scalable way to connect distributed applications over the internet.

The following diagram is depicting the REST API use cases with different HTTP operations:

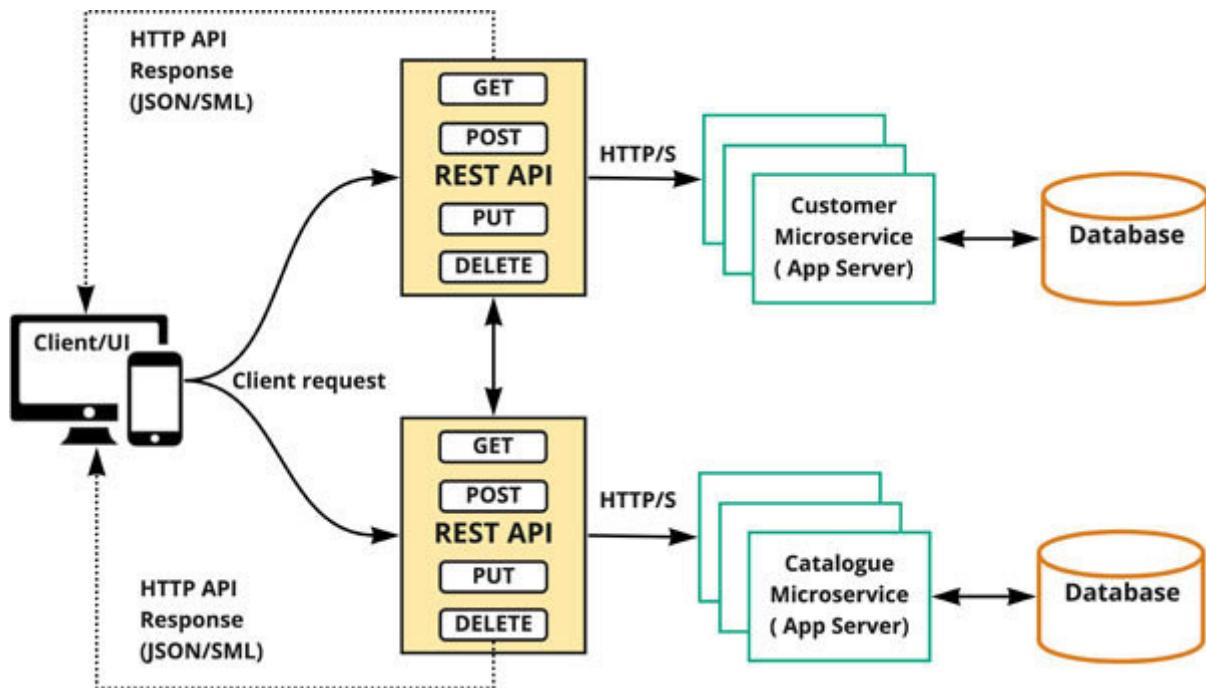


Figure 3.2: REST architecture with use case

In this book, we will cover REST over HTTP implementation.

REST and HATEOAS support

REST also supports HATEOAS style of API design. It's a great use of loose coupling of REST APIs for the given resource. It's based on hypermedia, or in other words URL of resources with operations. In this approach, clients can request a REST API for a resource at server-side and the server send additional URLs of possible operations on that resource in response body to clients. It helps clients to read those HATEOAS hypermedia URLs and take action and call other APIs accordingly.

HATEOAS works well with REST application architecture with its hypermedia concept. The term **hypermedia** refers to any content that contains links to other forms of media such as links of images, file locations, API URLs of related services and text, and so on. Clients can easily navigate to the hyperlink's URLs that come from servers.

In this example, the **links** section of this JSON response from the server has image URL, HTTP's method **GET**, and other details, which can be used by the client app to fetch image by **GET** call without adding extra business logic to hard-code images location from the source code or config files:

```
{
```

```
{  
  "LG 50B6000FHD"
```

```
[  
  {
```

```
  "type" : "GET"
```

```
}
```

```
]
```

```
}
```

```
}
```

You may choose the preceding HAL format by embedding these hyperlinks in the response body, or in HTTP response headers.

Hypertext Application Language (HAL)

When you design a RESTful service, there is a need to specify how to return data and links corresponding to a request. HAL provides a simple format that gives an easy, consistent way to hyperlink between resources in your REST API.

API design principles

API is a de-facto standard of creating cloud native microservices because it embraces it and is a part of standard twelve-factor apps cloud native principles.

These are major API design principles:

API should be the first citizen of the microservices In API first approach, API design should be the priority. All technical design and development should be around the API including agile processes. UI and backend services development are around API.

API should be easy, described, and API should be easy to understand by human stakeholders, self-explainable with well-documented API contracts.

API design and documentation should be first before In this approach API documentation should be designed, created, and validated before actual app development.

Benefits of API first approach

These are major benefits of the API first approach:

Early contract between different development teams and other business stakeholders.

Clear abstraction layer.

Early testing without writing source code.

Decoupling business logic.

Increase developer productivity by parallel API development.

Faster releases to market.

Reduce development cost.

Flexible and scalable API.

Save time by avoiding debugging and other issues in the early stage.

API design visual dashboard support.

Testing APIs before developing them by using mock request and response data.

Reduce API failure risk by early validation.

API components

These are the main API components that define API, rules, and other governance:

It's an independent entity or object which is related to real-life objects Example: order, catalog, student, and so on.

Set of resources. Example: orders.

Add, update, query, and delete are operations on resources. Example: It covers CRUD operations.

It's an identifier of a resource like a REST API, page, and so on.

It has URI and other information like how to access it, like HTTP, FTP protocols. Example: <https://google.com>.

Full REST endpoints path through which API resources can be located. Example:

Request It's a structured data which is being sent in an API request body like JSON in a request. Example: Create catalogue by passing catalogue details:

{

```
"Ceiling  
3400.5  
}
```

Response It is structured data that has been sent by REST API backend services as response with other details like HTTP status code, and so on. For example, response body from backend API, like list of all catalogue items or error message:

```
[  
{  
100.5  
},  
{  
"Ceiling  
3400.5  
}  
]
```

Query It's a query parameter that is embedded in the URL. Multiple queries can be appended in the API endpoint URL. In this example `keywords=iphone&model=ios` is a query parameter:
<https://www.amazon.com/keywords=iphone&model=ios>.

Path It's also a query that is inserted within the endpoint. In this example, `/1` is path parameters:

Request The header contains a bunch of key-value pairs which contain different REST API standards like security token, accept, content-type, and other headers:

The screenshot shows the Postman application interface. At the top, there is a header bar with 'GET' and a dropdown arrow, followed by the URL 'http://localhost:8010/catalogue'. Below the header are tabs for 'Params', 'Authorization', 'Headers (6)', 'Body', 'Pre-request Script', 'Tests', and 'Settings'. The 'Headers (6)' tab is currently selected and highlighted with an orange underline. Underneath, there is a section titled 'Headers' with a link to 'Hide auto-generated headers'. A table below lists six headers with their keys and values:

KEY	VALUE
<input checked="" type="checkbox"/> Postman-Token ⓘ	<calculated when request is sent>
<input checked="" type="checkbox"/> Host ⓘ	<calculated when request is sent>
<input checked="" type="checkbox"/> User-Agent ⓘ	PostmanRuntime/7.26.8
<input checked="" type="checkbox"/> Accept ⓘ	*/*
<input checked="" type="checkbox"/> Accept-Encoding ⓘ	gzip, deflate, br
<input checked="" type="checkbox"/> Connection ⓘ	keep-alive

Figure 3.3: Sample API request headers

Response status There are some standard HTTP status codes that are being returned with API response, which are being returned to clients/consumers. Example: **200** for success (ok), **400** for resource not found, **500** for internal server error, and so on.

These are standard response status codes:

codes: codes:
codes: codes:
codes: codes:
codes:
codes: codes:

codes: codes:

Table 3.1: HTTP generic response status codes

Media It's a data type or format during data exchange. These are a few important uses of media-types:

Vendor for example,

Collection + JSON or and so on.

HTTP cookie (web cookie, browser cookie) is a small chunk of data that contains client requests and maintains the client's state. The server sends a cookie to the user's web browser. Web browser at client side maintains session state between browser and server's API.

HTTP It's predefined operational methods that apply on API resources for **Create, Read, Update, Delete** operations.

These are the main pre-defined HTTP methods:

methods: methods: methods: methods: methods: methods:
methods: methods: methods: methods: methods: methods:
methods: methods:

Table 3.2: HTTP methods

For example, user login can have separate APIs for different business use cases:

POST User can login using credentials functionality.

POST Create new user/Sign-in process functionality.

PUT Reset/update password functionality.

GET Get user details.

DELETE Remove user.

PATCH Partially updates user.

API best practices

Now, we will discuss a few best API practices in detail:

Follow OpenAPI Modern apps API should follow **OpenAPI** specification to make it compatible and portable for all kinds of apps.

API web dashboard It should be developer-friendly with the API management dashboard, which helps to create, manage, and monitor APIs in large systems or microservices environments. There are many API open source and enterprise solutions like OpenAPI based **Google** and so on. They provide a web-based dashboard to manage APIs dynamically and can be exported as source code and shared with development teams.

Web-based HTTP with Most of the apps, databases, and messaging systems use REST over HTTP protocol communication over the internet. REST is widely accepted, supported by most of the clients and logical integration apps, and so on. It's more flexible, has rich features. If you are building an API then you should know the basics about HTTP web protocol and its methods, attributes, and status codes. Also, you should have a good understanding of the REST style of API interfaces, because REST is resource-oriented architectural style.

Return valid structured JSON Don't return plain text message. It should be well-structured JSON, XML, or a similar response.

Example is as follows:

```
{
```

```
{
```

```
"LG 50B6000FHD 127
```

```
}
```

```
}
```

Maintain status API must return HTTP status codes because when a client sends a request to server through REST API, it expects response from the server, if it's a success or failure. There are standard pre-defined error codes for this purpose:

purpose:
purpose: purpose: purpose: purpose: purpose: purpose: purpose:

purpose:	purpose:	purpose:	purpose:	purpose:	purpose:
purpose:	purpose:	purpose:	purpose:	purpose:	purpose:
purpose:	purpose:	purpose:	purpose:	purpose:	purpose:

Table 3.3: HTTP status code format

API Endpoint naming

Name the collections using plural nouns. The reason behind this, the same resource can return a single record or multiple records. It's not recommended to have two separate resources URIs for these two resources. For example, **/orders** is a valid URI name for API, which serves both purposes.

Use nouns instead of verbs. It will be a standard naming convention, because multiple operations can be done on a single resource or object. For example, **/orders** is a noun and correct way because order can be created, updated, deleted, and fetched. It's not recommended to use and so on.

Error handling, return error details with error Server resource should always returns appropriate error code, internal error code and simple human-readable error message for better error and exception handling at client-side apps, for example:

```
{  
    "200": "Connection refused"  
}
```

Return appropriate HTTP response status Every REST endpoint should return a meaningful HTTP response code to handle server responses in a better way like:

200 for success.

404 for not found.

201 resource is created.

304 not modified. Response already in its cache.

400 bad request. The client request was not processed, as the server could not understand what the client was asking for.

401 unauthorized. Not allowed to access resources, and should re-request with the required credentials.

403 forbidden. Client is authenticated, but the client is not allowed access to the page or resource for some reason.

503 services unavailable. The server is down or unavailable to receive and process the request.

Avoid nesting of related Sometimes, resources are related to each other like `/orders` resource object is related to catalogue category and user ID. We should not nest resources like:

```
GET /orders/mobile/111
```

It's recommended to use top-level resource and make other related resources as a query parameter like this:

```
GET /orders?ctg=mobile&userid=111
```

Handle trailing It's always advisable to use only one approach either with trailing spaces like `/orders/` or without it `/orders` to avoid any confusion.

Use sorting, filtering, querying, In many use cases, a simple resource name won't work.

You need to request server API resources to sort data in ascending or descending order:

```
GET /orders?sort=asc
```

Filter on some business conditions like return product catalog responses based on price range:

GET /orders?minprice=100&maxprice=500

Use cases where you want to query products based on their category like searching electronics products based on **mobile** category, for example:

GET /orders?ctg=mobile&userid=111

To improve performance and reduce latency on API calls over the internet, the client requests a subset of records at a single request like **10** records at a time for a given page. It's called pagination:

GET /orders?page=1&page_size=10

Versioning is a very important concept of API, which helps consumers to migrate to newer versions without any outage. In this scenario, some clients can access newer versions, and others can still use older versions.

There are various ways of API versioning:

Using URI It's a standard technique to maintain different versions of the same APIs to support older versions of API resources, if the server-side API resource is upgraded to a newer version. Clients take some time to migrate and use the latest version of

API. A new version to the same API resource can be changed, for example,

The internal version of the API uses the **1.2.3** format like this:

for example,

Major It contains major code changes in business logic or other components. A new major version is added to the new API and the version number is used to route to the correct host.

Minor and patch These are used internally for backward-compatible updates. They are usually communicated in changelogs to inform clients about new functionality or a bug fix. The minor version represents minor changes and the patch contains break-fixes or security patches, and so on.

Using query In this method, version number is added into query parameters in key and value. It's simple to use, however not recommended because it's difficult to route requests to APIs. For example:

Using custom headers: In this method, version number can be added in HTTP request header. It avoids the clutter of URI versions; however, we need to create and manage new headers.

For example: **Accepts-version:**

Using content negotiation: It's also added in the header, allows a single resource representation instead of versioning the entire API which gives us more granular control over versioning. In this method, no need to create routing rules at the source API codes of different versions. This approach is not very popular, because it's difficult to test and verify changes in browsers.

For example: **Accept: application/json; version=1**

API caching is a very much needed feature to improve API read (GET) performance. It caches responses from the API and for the same set of data and makes it available for other similar client requests. It's recommended to use distributed caching techniques in a distributed microservices environment so that the same cached response should be available for multiple instances of the same microservice app.

Rate limiting and Rate limiting is a technique of counting client requests with counter and limit based on the subscription or maximum-allowed limit to control traffic on the server, also it is useful for security reasons to avoid hackers to hit continuously and bring the system down by consuming all the memory and compute resources.

API throttling controls the way API is being consumed by external apps/ or clients. It also indicates a temporary state and is used to control the data that external clients can access through a REST API. When a throttle is triggered, we can disconnect client requests, client apps, device ID, a user or just reduce the

response rate. You can define a throttle at the application, API or user level.

There are multiple ways to implement Spring Cloud Gateway provides rate limiting wrapper using distributed caching such as **Redis** or a similar caching tool. We will discuss more implementation details later in this book.

API gateway It's recommended to expose APIs using API gateway tools to external apps. API gateway takes care of routing and orchestrating to designated server-side API, filtering, rate limiting, throttling, circuit breaker, API, authentication, authorization, and so on out of the box. It makes your API configuration outside of the business logic source code. It makes actual business logic code lighter and easy to debug and maintain.

API security

Here are some important API security practices:

API authentication and authorization All exposed REST APIs be secured and every client call should be authenticated first and authorized before forwarding to any server endpoint. There are multiple ways to implement A&A using API gateway such as **Spring Security** with Spring Cloud Gateway, OAuth2, SSO, and IAM tools provided by public cloud providers like AWS, GCP, Azure, and so on.

Single Sign-On is a recommended approach for A&A. There are multiple popular tools to implement SSO, such as and so on.

Expose API over HTTPS It's recommended to expose all APIs through HTTPS secure protocol over valid SSL/TLS certificates. TLS is the latest technology over SSL. It provides another layer of transport security on transmitted data over the network at network layer 4.

TLS encrypts and decrypts transported data over the network. TLS uses a mix of symmetric and asymmetric encryption methods.

The symmetric method encrypts and decrypts API request and response data using secret tokens known to both sender and recipient; typically, **128** but preferably **256** bits in length.

Asymmetric methods use private and public keys to encrypt and decrypt data. The public key of the recipient to be used by the sender to encrypt the data, but that data can only be decrypted with the private key of the recipient.

Do not expose database models to It's very important to not expose your database schema to the external world. It can be compromised by hackers.

Do not expose unique database ID to Avoid exposing actual unique ID to the external world, mainly when you are using it publicly. Anyone can guess the ID of others and misuse it. You can return a different GUID which should be dynamically generated for the given client request.

Avoid SQL Handle your query parameters to avoid SQL injection. It's a code injection that can destroy databases also by passing destructing queries in the query string of API query parameters. Hackers usually add malicious code on web page input. Many security tools and IAM providers handle SQL injection such as AWS DDOS.

Put APIs behind the It's recommended to always expose API through a security firewall to provide an additional level of security to the external world. Web server and DNS servers provide firewalls such as **Nginx** web servers, **AWS** and so on.

Monitoring It is preferable to monitor the traffic of client apps and server-side APIs, their inter-communication tracing logs, memory, and CPU usage with monitoring and scaling your infrastructure to cater even for higher loads and reduce costs by removing extra hardware if it's not being utilized properly.

There are many monitoring tools such as **Datadog** and so on. Spring framework also provides out-of-the-box support for **Spring Open Tracing**, and **Zipkin** for API tracing. API tracing can be handled at the DevOps side using Kubernetes-based service mesh as well. Service mesh is recommended technology to handle, route, and trace API logs, because it provides Istio's Envoy proxy to route API traffic and gather all the API-related metadata information.

Exposing API on communication protocols

There are different API communication protocols to expose services as API:

It's an XML-based messaging protocol over HTTP. It's widely used, because it was the first integration protocol over HTTP. It works on stub and skeleton concept between consumer and provider apps. It's one of the types of **Remote Procedure Call**

This is based on HTTP protocol. It's widely used flexible data modeling, and supported by OpenAPI and other open source and enterprise API libraries and apps. It supports JSON, and XML. REST can also offer better performance than SOAP because it can cache information.

It's based on only JSON on RPC over HTTP. It's simple to use, has limited features, limited commands. It doesn't have rich features like REST. It's for simplicity for straight forward use cases.

It's an open-source high performance RPC framework that can run on any environment. It supports protocol buffers over HTTP by default for better performance. It supports JSON and other data formats as well. It provides all kinds of functional calls and is not bound to predefined **PUT** HTTP methods. Another advantage is that it appears to be that the calls between apps are local for client rather than a remote one executed over the network.

It's good for microservice architecture and Docker-based Kubernetes apps for a massive number of remote calls. It gives better performance over REST in these use cases. It's faster, flexible, and provides more granular level control over REST.

In the world of microservices, gRPC will become the de-facto standard very soon. The performance benefits and ease of development are just too good to pass up. However, REST will still be around for a long time and supported by a large ecosystem of open sources and enterprise integration patterns. It still excels for publicly exposed APIs and backward compatibility reasons.

Apache Developed at Facebook, it's lightweight and supports JSON or binary data format. It supports various transport methods and several different server-process implementations. It's not bound with API specification; it can be changed according to requirements. It supports 28+ programming languages. Its code generation system takes a simple definition language as input and generates code across supported programming languages that use the abstracted stack to build interoperable RPC clients and servers. Anytime a new version can be created and deployed without affecting code.

There are various reasons to use **Thrift** over something like JSON/HTTP:

Strong JSON is a good choice for scripting-based languages with key-value structured data, which can be easily parsed. However, it's not recommended for strongly typed languages like Java, .Net, C++, and so on. Thrift supports strong, native types and also provides a mechanism for throwing application-level exceptions.

Thrift gives better performance. JSON/HTTP are structured, humanly readable; however, they take more time to parse and impact performance.

Serialization Thrift saves memory space and CPU speed because it directly works on binaries. It also compresses data efficiently.

Versioning Thrift provides versioning data support. It's recommended in a distributed environment where service interfaces may change frequently.

It's a query language that solves limitations of REST. It supports JSON data format over HTTP. It allows clients to structure their requested data and optimize according to their needs without re-transforming. GraphQL with **Backend for Frontend** is used in a number of enterprises to expose APIs.

There are many advantages of choosing GraphQL over REST:

It avoids-over fetching of API response data. It provides easy customization for the required response, which saves a lot of

network bandwidth and backend source code side manual filtering. It allows fetching only required data fields from the response data.

It supports strongly typed schema and data types such as It provides GraphQL **Schema Definition Language** which can work on these data types, it helps GraphQL less error-prone and more validated.

Faster response: It avoids multiple calls to resources like REST. It gets most of the required response data from API in one shot. It saves a lot of time and bandwidth by reducing the number of network round trips to the server and reducing network latency.

In GraphQL, versioning is not required like REST. It adds new fields and deprecates older fields. This approach is intuitive as the client receives a deprecation warning when querying a deprecated field.

It's easy to learn and implement GraphQL with its easy interface library.

GraphQL is not limited to only server-side; it can be used for front-end as well.

GraphQL schema stitching feature allows combining multiple, different schemas into a single schema. In a microservices architecture, where each microservice handles the business logic and data for a specific domain, this is very useful. Every

microservice can define its own schema and customized API response data model for its clients.

Every API protocol has its own advantages and disadvantages. We can choose a better option based on the business use cases and **non-functional requirements**

In this book, we will cover REST API which is based on lightweight HTTP protocol, which is flexible, widely used, **OpenAPI** supported, has rich features over other methods, supported by most of the modern open sources, databases, and other microservices and monolithic apps. It is also easy to implement and manage by development teams. Even non-technical users can also understand REST contracts easily.

In the microservices ecosystem, apps, and databases are distributed and deployed on multiple servers on multi-clouds. They often communicate with each other to solve business use cases. In real life, whenever we use our mobile apps/web browser for checking email, chat, social web site, and weather report, we are indirectly using APIs from our laptop or mobile devices. API exposes backend services to external clients/consumers like web browser/mobile/IoT and other logical clients to make HTTP protocol-based connections to local/remote cloud servers or data centers. These all clients interact with each other using APIs.

SOAP can be also extended and used as a REST API.

API documentation and specifications

In this topic, we will discuss all API and cover these topics:

API documentation

API specifications

API documentation

API documentation is a reference document for APIs which contains detailed information about API usage by developers and other stakeholders. It makes developer life easy to follow the same API standard document for all API microservices development. It contains all the information about API like API name, request, response, header, and other related information.

There are many tools available to generate API documents dynamically like **Postman**, and so on. They also generate static API documents which can be persisted also in source code repositories like GitHub, and so on.

API specifications

API specification contains details of API like a contract, *how it will be integrated with other*. It has all rules for handling different kinds of web, mobile, and IoT clients, and so on. It also defines relationships with different API integration points. **OpenAPI** (formerly the **Swagger**) is one of several API specification languages to define API specification.

OpenAPI specification (formerly Swagger specification) is an API description format for REST APIs. An OpenAPI file describes these following API specifications:

Describe required endpoints and API operations on each endpoint and so on), for example, **POST GET /catalogue** and so on.

Operation query and path input and output parameters for each operation.

Authentication and authorization methods using the security token.

Client metadata info like client unique ID, device type like mobile/IOT/web, and so on.

Request and response data content type, for example,

Contact information, license, terms of use, and other information.

API specifications can be written in YAML/JSON. The format is widely accepted, easy to learn, and readable to both humans and machines.

Planning API design

A good planning is required for API designing. Here are major groundwork analysis steps which should be followed in API first approach:

It's the starting phase where business use cases have to be analyzed. It's the next step after microservices high-level domain-driven design. It involves stakeholder's meetings.

Identify APIs with unique In this step, all use cases have to be analyzed of the microservices and identify unique use cases of APIs and give them resource names like orders, catalog, customer, and so on.

Identify business In this step, we need to identify business contracts, client's requirements, and how backend APIs will send responses for the different kinds of client requests like responses could be different for Web browser and mobile.

Review and validate Arrange review meetings with different stakeholders and validate business contracts which should be agreed upon between the client (consumers) and the server (producers).

Design API In this step, open standard OAPI specification has to be followed, which should be compatible with all kinds of

consumer and producer apps. There are multiple API builder tools available like **Postman**, and so on.

API data In this step, we need to map API resources with the backend database and create data modeling and response structure out of that data model. It's not necessary to return the entire database entity, the service layer in the application filters the data and only returns responses that are requested by the client.

API In this step, apply common rules to API standards and security policies.

Monitoring and managing API using dashboard It's very important to have standard API monitoring dashboard tools to enforce API governance, and monitor APIs on distributed microservices on the cloud.

API specifications

In this section of the book, we will discuss these two popular API specifications and also compare them with each other:

OpenAPI Specifications

RESTful API Modelling Language

[OpenAPI Specifications \(OAS\)](#)

The **OpenAPI Specification** is a community-driven open standard specification within the OpenAPI initiative, a Linux Foundation-funded collaborative project:

OAS defines a standard, language-agnostic interface to REST APIs which allows both humans and apps to discover and understand the capabilities of the web microservice without looking at source code, documentation, or through network traffic inspection. If OAPI is followed by consumer and producer apps, a consumer can understand and interact with the remote microservice with a minimal effort of implementation logic.

OAS documents describe APIs services and are represented in either YAML or JSON formats. These documents may either be produced and served statically or be generated dynamically from an application or API generation tools.

OAS doesn't require rewriting existing APIs of legacy monolithic apps. Technology agnostic doesn't require binding to any programming language. The OAS does not mandate a specific development process such as design-first or code-first. It does facilitate either technique by establishing clear interactions with an HTTP API.

OAS definition can be used by API documentation generation and management tools to visualize the API, code generation tools to generate servers, and client code in various supported programming languages and testing tools.

RESTful API Modelling Language (RAML)

It's a YAML-based REST API modeling specification language. It follows the hierarchical approach of API designing. It makes the whole API lifecycle easy to manage. It's capable of designing an API that doesn't follow all the REST constraints. There are many API management tools available to support this standard like **AWS API Restlet**, and so on.

RAML can be converted to OAPI using

OAS versus RAML

The main difference between these two is that RAML is a top-down specification and OAS is a bottom-up specification. In this book, we will discuss OAS/OAPI based implementation because of a bottom-up specification approach, and rich community support with a wide range of tool support.

API management tools

There are many tools available for managing larger microservices cloud ecosystems. They are capable of designing API based on API standard specifications, expose, route, test, and other operations.

[SwaggerHub](#)

It's a very popular web-based tool based on the OAPI standard. It's based on **Swagger** open-source API management tool which is very popular with SpringBoot framework for generating APIs from source code. **SwaggerHub** is one step ahead and provides all API management services including create, design, test, export, and import APIs. We will discuss more with examples in the latter part of this chapter.

Other API management tools

There are other tools also available for API management – **Google** **IBM** **API Kong Enterprise Chrome PostMan MuleSoft Dell TIBCO Microsoft Azure API Oracle** and so many other vendors and open-source tools.

OpenAPI specification is an initiative of many top IT companies like Google, IBM, SmartBear, and so on under Linux foundation authority. It has been designed as an open global standard that should be followed by all API management tools.

Designing and generating API docs

In this section, we will discuss these two ways of generating API doc. API first is a recommended approach because in this approach we can design, validate, test, and make API governance policy early in the development phase without actual implementation. However, if we already have existing legacy or monolithic apps, then API docs can be generated dynamically using the Swagger API tool.

Let's discuss how we can design API using these two approaches:

Code First with OpenAPI 3.0 and Spring REST API

API First with SwaggerHub

Code First with OpenAPI 3.0 and Spring REST API

In this approach, source code is developed first and then API docs will be generated with the help of the application's REST API source code defined rules. Let's follow these steps to create an API document for a Java application using the **Maven** build tool:

Prerequisite

You need to have minimum of Java 8 on your machine.

Windows/Mac/Linux. We are using MacBook throughout in all the exercises.

Install Maven build tool from the following website:

<https://maven.apache.org/install.html>

SpringBoot v2.27 framework dependency in your Maven's **pom.xml** file.

To have Springdoc, OpenAPI automatically generates the **OpenAPI 3 (OAS)** specification docs for our API. We need to add these dependencies in **pom.xml** file of Maven:

Now run the application, the OpenAPI descriptions will be available at the URL:

http://localhost:8012/v3/api-docs

Result will be shown as follows:

```
1. {
2.     "openapi":"3.0.1",
3.     "info":{
4.         "title":"OpenAPI definition",
5.         "version":"v0"
6.     },
7.     "servers":[
8.         {
9.             "url":"http://localhost:8012/",
10.            "description":"Generated server url"
11.        }
12.    ],
13.    "paths":{
14.        "/orders":{
15.            "get":{
16.                "tags":[
17.                    "order-controller"
18.                ],
19.                "operationId":"fetchOrders",
20.                "responses":{
21.                    "200":{
22.                        "description":"default response",
23.                        "content":{
24.                            "*/*":{
25.                                "schema":{
26.                                    "type":"array",
```

```
27.          "items":{  
28.            }  
29.          }  
30.        }  
31.      }  
32.    }  
33.  }  
34. }  
35. },  
36. "put":{  
37.   "tags": [  
  
38.     "order-controller"  
39.   ],  
40.   "operationId":"updateOrders",  
41.   "responses": {  
42.     "200": {  
43.       "description": "default response",  
44.       "content": {  
45.         "*/*": {  
46.           "schema": {  
47.             "type": "array",  
48.             "items": {  
49.               "type": "object"  
50.             }  
51.           }  
52.         }  
53.       }  
54.     }  
55.   }  
56. },
```

```
57.     "post":{  
58.         "tags":[  
59.             "order-controller"  
60.         ],  
61.         "operationId":"postOrders",  
62.         "responses":{  
63.             "200":{  
64.                 "description":"default response",  
65.                 "content":{  
66.                     "*/*":{  
  
67.                         "schema":{  
68.                             "type":"array",  
69.                             "items":{  
70.                                 "type":"object"  
71.                             }  
72.                         }  
73.                     }  
74.                 }  
75.             }  
76.         }  
77.     },  
78.     "delete":{  
79.         "tags":[  
80.             "order-controller"  
81.         ],  
82.         "operationId":"deleteOrders",  
83.         "responses":{  
84.             "200":{  
85.                 "description":"default response",  
86.                 "content":{  
87.                     "*/*":{
```

```
88.          "schema":{  
89.              "type":"array",  
90.                  "items":{  
91.                      "type":"object"  
92.                  }  
93.              }  
94.          }  
95.      }  
  
96.      }  
97.  }  
98. }  
99. }  
100. },  
101. "components":{  
102.     "schemas":{  
103.         "PurchaseOrder":{  
104.             "type":"object",  
105.             "properties":{  
106.                 "id":{  
107.                     "type":"integer",  
108.                     "format":"int64"  
109.                 },  
110.                 "cname":{  
111.                     "type":"string"  
112.                 },  
113.                 "email":{  
114.                     "type":"string"  
115.                 },  
116.                 "pname":{  
117.                     "type":"string"  
118.                 },  
119.             }  
120.         }  
121.     }  
122. }
```

```
119.         "price":{  
120.             "type":"number",  
121.             "format":"double"  
122.         }  
123.     }  
124. }  
  
125. }  
126. }  
127. }
```

The OpenAPI definitions are in JSON format by default. For YAML format, we can try to add YAML like this:

<http://localhost:8012/v3/api-docs.yaml>

Now, we will discuss these important operations using the Swagger open-source tool:

Setting up Spring doc OpenAPI with Swagger UI

Generating API doc with JSR-303 validation

Generating API doc with **@Operation** and **@ApiResponses**

Setting up Spring doc OpenAPI with Swagger UI

The preceding OAPI implementation only returns JSON format data. If we want to visualize with Swagger like web dashboard, then we can access using this URL:

<http://localhost:8012/swagger-ui.html>

The following screenshot is depicting the OpenAPI definition for the REST APIs:

The screenshot shows the Swagger UI interface running at <http://localhost:8012/swagger-ui/index.html?configUrl=/v3/api-docs/swagger-config>. The title bar says "localhost:8012/swagger-ui/index.html?configUrl=/v3/api-docs/swagger-config". The main header has "Swagger" and "Explore" buttons. Below the header, it says "OpenAPI definition" with "v3" and "OAS3" indicators. A "Servers" dropdown is set to "http://localhost:8012 - Generated server url". The main content area is titled "order-controller". It lists four operations for the "/orders" endpoint: "GET /orders" (blue), "PUT /orders" (orange), "POST /orders" (green), and "DELETE /orders" (red). At the bottom, there is a "Schemas" section showing the "PurchaseOrder" schema:

```
PurchaseOrder <pre>{</pre> id integer($int64)<br/> osname string<br/> oname string<br/> pname string<br/> price number($double)<br/>}</pre>
```

Figure 3.4: OpenAPI REST definition

Generating API doc with JSR-303 validation

If we use the model with JSR-303 bean validation annotations, like **@NotBlank** and the Spring doc OpenAPI generates additional schema documentation with these validation constraints which helps to understand API's validation rules and security policies:

```
PurchaseOrder {  
  
    private Long id;  
  
    @NotBlank  
    = max =  
  
    private String cname;  
  
    @NotBlank  
    = max =  
  
    private String email;  
  
    private String pname;
```

```
private Double price;
```

```
}
```

The screenshot shows the Swagger UI interface at `localhost:8012/swagger-ui/index.html?configUrl=/v3/api-docs/swagger-config`. The title bar includes the Swagger logo and the URL. The main content area is titled "OpenAPI definition v0 OAS3". Below it, a sub-section titled "order-controller" lists four operations:

- PUT** /orders/{id}
- POST** /orders/{id}
- DELETE** /orders/{id}
- GET** /orders

Below the operations is a section titled "Schemas" containing the schema for "PurchaseOrder":

```
PurchaseOrder <pre>v</pre> {<br/>    id          integer($int64)<br/>    cname       string<br/>    maxLength: 30<br/>    minLength: 0<br/>    email       string<br/>    maxLength: 40<br/>    minLength: 0<br/>    pname       string<br/>    price       number($double)<br/>}</pre>
```

Figure 3.5: OpenAPI REST API response

If we use these annotations, it generates documentation for the response codes and add into response.

Generating API doc with `@Operation` and `@ApiResponses`

In some cases, we need to add an additional custom description to API using a couple of OpenAPI-specific annotations. We can annotate our controller's endpoint with `@Operation` and

```
@Operation(summary = "Get order by
```

```
@ApiResponses(value = {
```

```
    @ApiResponse(responseCode = description = "Found  
the content = {
```

```
        @Content(mediaType = schema =  
@Schema(implementation = }),
```

```
        @ApiResponse(responseCode = description = "Invalid  
id content = @Content),
```

```
        @ApiResponse(responseCode = description = "Order  
not content = @Content) })
```

```
public List getOrderById(@Parameter(description = "order id to be  
final String id) throws URISyntaxException {
```

```

List orders = orderService.fetchOrdereDetails(id);

return orders;

}

```

Now, our OpenAPI dashboard looks like this with all additional comments for more verbose details:

The screenshot shows the OpenAPI REST API specifications for an order-controller. The main section displays a GET endpoint for '/orders/{id}' with the following details:

- Parameters:**
 - Name:** id **Description:** order id to be searched **Type:** string (path) **Required:** true
- Responses:**

Code	Description	Links
200	Found the book	No links
400	Invalid id supplied	No links
404	Order not found	No links

Below the main section, there are four colored horizontal bars representing other endpoints:

- PUT /orders/{id}**
- POST /orders/{id}**
- DELETE /orders/{id}**
- GET /orders**

Figure 3.6: OpenAPI REST API specifications

[API First with SwaggerHub](#)

SwaggerHub is a SaaS service and follows OpenAPI specification to create REST API documentation and export programming language-specific code. Let's discuss how to use SwaggerHub and create API docs.

In this section we will cover these topics in detail:

What is Swagger?

SwaggerHub UI overview

In this book, we will cover the OpenAPI v3.0 specification.

Introducing Swagger

Swagger follows OpenAPI specifications standards, which is a set of open-source tools for designing, building, documenting, and consuming REST APIs. There are three major components of Swagger:

Swagger Visualize OpenAPI definitions as interactive API documentation.

Swagger A browser-based editor where you can write OpenAPI definitions.

Swagger It generates server code and client libraries based on OpenAPI specifications.

SwaggerHub UI overview

We can access the SaaS version on **SwaggerHub** in which the **MY hub** page lists all the APIs that you have created or have access to. If you are new to SwaggerHub and have no APIs yet, the list will be empty, but it will change once you start creating APIs.

You can log in to SwaggerHub using a GitHub account or create a new account.

MY hub page, which shows newly created APIs as follows:

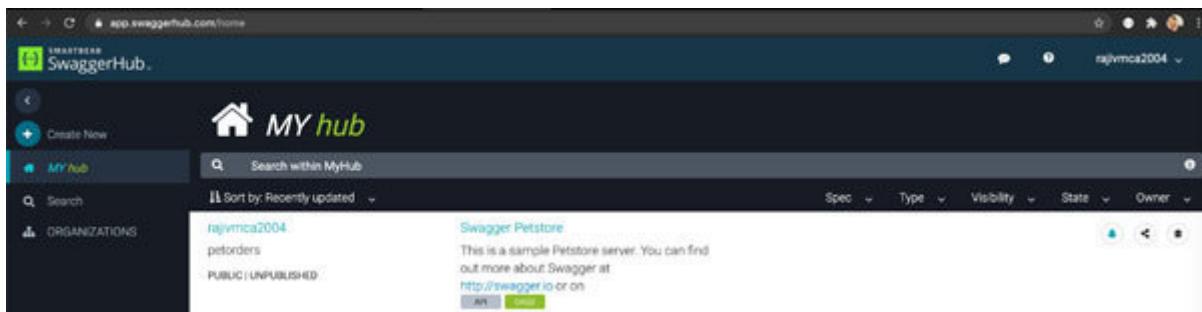


Figure 3.7: SwaggerHub My hub page

It provides a **Swagger Editor** web dashboard, where we can create API and customize by making changes in source code YAML.

This is the screenshot of sample pet store application's OAPI specifications with all APIs on the left side and it gives a code

editor to customize the code and preview section on the right side:

The screenshot shows the SwaggerHub interface. On the left is the 'editor' pane, which displays the OpenAPI 3.0.0 specification for the Petstore API. The specification includes definitions for 'Info', 'Tags', 'Servers', and various API endpoints like 'pet', 'store', and 'user'. The 'pet' endpoint is expanded, showing methods such as GET /pet, POST /pet, and PUT /pet. The 'store' endpoint also has its details expanded. On the right is the 'preview' pane, titled 'Swagger Petstore'. It shows the same API structure but with a more user-friendly interface. It includes sections for 'Terms of service', 'Contact the developer', and 'Apache 2.0'. Below these are sections for 'Servers' (set to 'https://virtserver.swaggerhub.com/rajivmca2004/petorders/1.0.0') and 'pet' (with sub-methods: POST /pet, PUT /pet, and GET /pet/findByStatus). At the bottom of the preview pane, there are four API operation cards: 'POST /pet' (Add a new pet to the store), 'PUT /pet' (Update an existing pet), 'GET /pet/findByStatus' (Finds Pets by status), and 'GET /pet/findByTags' (Finds Pets by tags).

Figure 3.8: SwaggerHub editor and preview page

This following screenshot of OAPI Swagger specifications with API definitions:

This screenshot shows the Swagger Hub interface for the Petstore API. At the top, it displays the URL <https://virtserver.swaggerhub.com/rajivmca2004/petorders/1.0.0>. Below the header, there's a navigation bar with links for 'SwaggerHub', '1.0.0', 'PAGES', 'Terms of service', 'Contact the developer', 'Apache 2.0', and 'Find out more about Swagger'. A 'Servers' dropdown is set to <https://virtserver.swaggerhub.com/rajivmca2004/petorders/1.0.0>. On the right, there's an 'Authorize' button.

The main content area is titled 'pet' and lists several API operations:

- POST /pet**: Add a new pet to the store
- PUT /pet**: Update an existing pet
- GET /pet/findByStatus**: Finds Pets by status
- GET /pet/findByTags**: Finds Pets by tags
- GET /pet/{petId}**: Find pet by ID
- POST /pet/{petId}**: Updates a pet in the store with form data

Figure 3.9: Swagger Hub Petstore page

There is an option to **Export** OAPI Swagger specifications of API to supported **Client Server** and **Download API** options.

This screenshot predicts various codegen options:

This screenshot shows the SwaggerHub interface with the 'Codegen Options' panel open. The left sidebar shows the API structure: 'Info', 'Tags', 'Servers', and a search bar. The main content area shows the 'pet' resource with its methods: POST /pet, PUT /pet, GET /pet/findByStatus, GET /pet/findByTags, GET /pet/{petId}, and POST /pet/{petId}. The 'Servers' section is set to '1.0.0'. The 'Codegen Options' panel on the right lists various frameworks and languages, with 'Server Stub' currently selected. Other options include Client SDK, Documentation, and Download API.

```
openapi: 3.0.0
info:
  title: Swagger Petstore
  version: "1.0.0"
  description: |
    This is a sample Petstore server. You can find out more about Swagger at
    [Http://swagger.io](http://swagger.io) or on
    [Trc.FoundNode.net](http://swagger.io/inc).
  termsOfService: "http://swagger.io/terms/"
  contact:
    email: apiteam@swagger.io
  license:
    name: Apache 2.0
    url: "http://www.apache.org/licenses/LICENSE-2.0.html"
servers:
  # Added by API Auto Mocking Plugin
  - description: SwaggerHub API Auto Mocking
    url: https://virtserver.swaggerhub.com/rajivmca2004/petorders/1.0.0
    url: "https://petstore.swagger.io/v2"
```

Figure 3.10: SwaggerHub codegen options

SwaggerHub provides an advanced feature to export APIs after designing to the required programming languages like Java and Spring. It will create a client and server code, which can be imported as Maven's project. The same exported code can be used by developers to write an implementation of Spring controller classes in our case. It supports many other programming languages and formats. In this book, we have a scope of Spring-based Java applications only.

Swagger Hub also provides the feature to share and collaborate with other development teams over the internet.

Testing API with SwaggerHub inspector

You can use Swagger inspector to test REST APIs with mock data. You need to install the Chrome web browser's **Inspector** plugin before testing.

There is another powerful tool **Chrome Postman SaaS service on the cloud to create API and test with mock data. It also provides load testing using its runner API.**

In this screenshot, we are testing SpringBoot API, which is returning response in JSON:

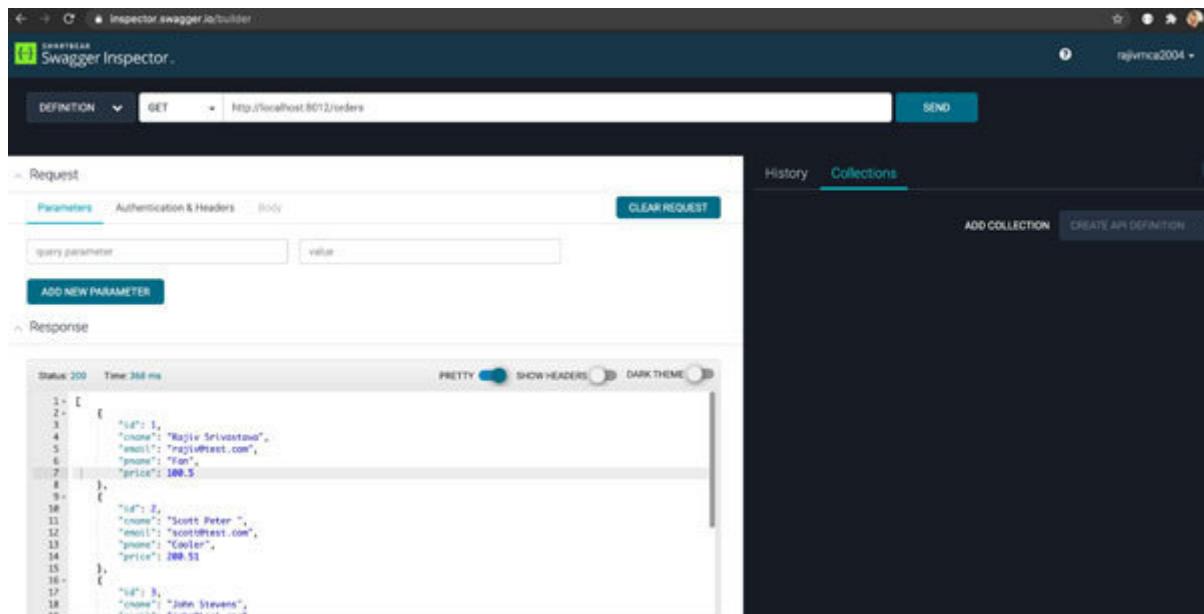


Figure 3.11: SwaggerHub API testing with response

We can create and store these APIs category wise as a group also.

SwaggerHub Enterprise version has more advanced features of design and documentation, functional testing, load testing, monitoring and alerting, mocking, and virtualization. Mocking REST API features can be tested only with the SwaggerHub ReadyAPI enterprise version.

Conclusion

In this chapter, we discussed everything about API overview, REST, and OAPI standard specifications. We discussed the API first approach and its benefits for design and development. We have gone through REST and other communication protocols and deep-dived into REST specifications and their relationship with HATEOAS and HAL. We have discussed all important API best practices. Also, covered how to document REST API using various API management tools, how to plan for API designing. We have discussed SwaggerHub UI dashboard and how to design and create API using this. Finally, we covered testing with SwaggerHub Inspector.

In the next chapter, we will explain how to implement and expose microservices as REST endpoints using SpringBoot and Spring Cloud. We will also discuss how to create a centralized logging server, make microservices resilient by implementing circuit-breakers using Spring Retry and Resilience4j open-source libraries.

Points to Remember

API is an application interface to integrate apps with each other.

API first approach is industry standard which is widely accepted for all new cloud-native microservice projects.

API first is good for the early detection of integration issues.

Always follow API best practices for cloud-native microservice projects.

API should always be secured behind a firewall and must be authorized and authenticated using secure token and embedded roles.

API docs is the right approach to document all the relevant specifications and share with the stakeholder for review and verification.

I recommended to use API management tools to automate APIs and persists.

Always test API by mocking test data before starting development.

API should be verified and agreed upon between API consumers and producers before the actual code development phase.

OpenAPI is a set of industry-standard API specifications.

SOAP can be also extended and used as a REST API.

HATEOS is a standard to add hypermedia links to REST API response and operation, that helps client to decide and send further consecutive requests.

RAML can be converted to OAPI using tools.

You can log in to SwaggerHub using the GitHub account or you can create a new account with your email ID.

SwaggerHub enterprise version has more advanced features of design and documentation, functional testing, load testing, monitoring and alerting, mocking, and virtualization. Mocking REST API feature can be only tested with SwaggerHub ReadyAPI enterprise version

There is another powerful tool called Chrome Postman which is a SaaS service. Also, to create and test API with mock data. It also provides for load testing using its runner service. It's open-source.

Key terms

Simple Object Access Protocol (XML protocol).

Open API standard.

Representational State Transfer.

Application Programming Interface.

Open API Specifications.

RESTful API Modelling Language.

HyperText Transfer Protocol.

Service Oriented Architecture.

Uniform Resource Identifier.

Uniform Resource Locator.

Non-Functional requirements.

JavaScript Object Notation.

JSON Web Token.

Remote Procedure Call.

Google Remote Procedure Call.

Hypermedia as The Engine of Application State.

Hypertext Application Language.

Backend for Frontend.

References

<https://www.openapis.org/>

<https://swagger.io/>

<https://raml.org/>

CHAPTER 4

Build Microservices with Spring Framework

Spring Boot which was introduced in 2014 as part of the Spring framework is microservices ready and is the most popular enterprise Java microservices framework. More than 60% of enterprise applications run on Java and Spring has good integration with almost all popular development libraries. It helps faster development of Java-based microservices with its built-in libraries.

In this chapter, we will cover cloud-related components of Spring Boot and Spring Cloud. We will also cover steps to build microservice using the **Spring Initializr** start.spring.io web portal and expose services as REST API. Later in the chapter, we will learn how to develop sample microservices using **Spring** **Spring** and **Tomcat** app server. We will also learn how to create a centralized configuration using the Spring Cloud config server. The chapter will also cover fault tolerance and resiliency using a Spring circuit breaker.

This chapter will not cover all Spring framework components because it is intended for cloud-centric microservices development.

Let's get started!

Structure

In this chapter, we will cover the following topics:

Introduction to Spring Boot

Spring framework use-cases

Introduction to Spring Cloud

Spring Cloud libraries

Creating a Spring Boot microservice project using **Spring.io**

Introduction to Spring Cloud config server

Building Spring Cloud config server

Building Spring Cloud config client

Fault-tolerance using Spring Cloud Circuit Breaker

Introducing to Resilience4J

Implementing Resilience4J circuit breaker

Objective

After studying this chapter, you should be able to understand the Spring Boot framework and Spring Cloud projects. This chapter will help you in implementing a sample microservice project using **Spring initializr** and application configuration using Spring Cloud Config Server. It will help you to understand and implement fault tolerance and resiliency using the **Resilience4j** circuit breaker.

Introduction to Spring Boot

It's a #1 Java framework. **Spring** is the most popular Java framework in the market, around 60% of enterprise applications run on Java and have good integration with almost all popular development libraries. Java is considered to be the best language for microservices by many developers. Among the application frameworks, there is a clear winner, and it's called *Spring!* Both *Spring Boot* (No. 1) and *Spring Framework* (No. 2) are well ahead of the competition. Java EE is bulky and not suitable for microservices.

There was a survey conducted in 2019 and according to that, *there are 4.2 million Spring developers in the world, around 1.8 million Spring projects created every month and there are around 124 million downloads per*

Spring Boot works on **Model View Controller** design pattern. It provides a simple interface to build Java-based microservices faster without configurations, which increases the developer's productivity. Developers can only focus on the source code development of the business code rather than managing the boilerplate configuration code.

These are a couple of major advantages and features of the Spring framework:

Improves developer productivity by automatically injecting and adding cross-cutting concerns and configurations separately. It helps developers to focus only on developing business logic.

Simplifies build configuration using built-in starter features. It injects all required configurations in the microservice app by just adding starter configuration. The developer doesn't have to write complex code for the configuration.

Helps to create stand-alone and lightweight Spring microservice applications.

Embeds or **Undertow** application servers directly by adding a simple configuration.

Automatically configures Spring and third-party libraries. It supports most of the modern API and libraries.

Spring Boot provides cross-cutting concerns and non-functional requirements such as observability, app metrics, health checks, auto database connection, connection pooling, logging, and so on.

externalized configuration across all the microservices apps.

No strong dependency on for XML configuration.

All configurations can be maintained in simple properties or YAML text files.

Provides built-in and customized annotations for different automated configurations.

It provides various ready-to-use templates to integrate with popular databases, messaging brokers, and other third-party libraries, and so on.

Spring Boot creates a Java application that is embedded with Tomcat and other dependent libraries.

Faster REST API development.

Good integration with almost all popular databases, messaging frameworks, and other libraries.

Orchestration of manual and auto-scheduled jobs/batches.

Provides second-level caching.

Provides Kubernetes support with Spring.

Spring framework use cases

These are some of the use cases of Spring framework:

Build cloud-native microservices.

Web, mobile, and Internet of app development.

Build short-lived and scheduled batch jobs.

Microservice security.

Microservice caching.

Microservice logging.

Microservice API health monitoring, capturing, and analyzing telemetry data with out-of-the-box integration with **Elasticsearch, Logstash, and Kibana** **Elasticsearch, Fluentd, and Kibana** Prometheus, Grafana, and Wavefront.

Introduction to Spring Cloud

Spring Cloud provides tools for developers to quickly build microservices apps in distributed systems. Spring Cloud developers can quickly build up microservices apps. These microservices apps can be deployed locally, on remote data centers, or in any cloud.

The following diagram is depicting major cloud components of Spring Cloud:



Figure 4.1: Cloud components of the Spring Cloud project

These are a couple of major advantages and features of Spring Cloud:

Cloud distributed configuration Spring Cloud server manages centralized configuration which can be accessible for all the dependent microservices apps. **Spring Config Server** project takes care of this.

Service registry and service It provides an advanced way of service registry and discovery. When microservices are deployed in a distributed environment on multiple clouds. **Spring Cloud Eureka** project takes care of this.

API It provides intelligent routing and orchestration of client requests to different REST API endpoints of microservices. It also helps communication between microservices. **Spring Cloud Gateway** project is a recommended solution for API gateway.

Secure JWT one-time token support for authentication and authorization. Spring security project has out-of-the-box JWT support.

Client side load It distributes client traffic load to multiple backend microservices. Spring Ribbon takes care of client-side load balancing.

Fault tolerance using circuit It maintains fault tolerance and resiliency between services. Resilience4J project takes care of this.

Global It ensures that there should be only one thread to access any resource. It can be managed by using *lock* programmatically. The lock will be acquired by the requested resource and will be available only to the other threads after releasing the lock. **Spring Data JPA** provides global locks.

Leadership election and cluster It allows the microservice app to work with other microservice via a third-party system. The

leadership election is used to provide *global state* or *global ordering* without sacrificing the availability. Spring integration project provides leadership election and cluster state.

Distributed It follows the pub-sub model. In this model, the publisher sends a message to the subscriber/consumer for consumption. **Kafka** and **RabbitMQ** are popular high-throughput messaging systems. **Spring Cloud Data Flow** provides great integration with these messaging brokers, also a recommended choice for event-driven programming.

Spring Cloud libraries

At present, the following are active Spring Cloud projects. These are some popular libraries:

Spring Cloud It provides centralized external configuration management as a separate backed service. These configurations can be persisted on the Git source code repository. It maps these configurations directly to Spring Environment but could be used as the key value paid by any microservice app.

Spring Cloud Netflix It provides integration with popular Netflix OSS components like and so on. All these are now deprecated. **VMware** and **Tanzu** now recommend non-Netflix OSS options for all these Spring components.

Spring Cloud It links nodes of a distributed system with a message broker. **Kafka** and **RabbitMQ** integrations are out of the box with this project. It can be used to propagate, state, or configure changes across a cluster, for example, publishing config change events to other services on the cluster.

Spring Cloud It provides an easy integration to deploy and run applications on open-source Cloud foundry.

Spring Cloud Open Service It's a framework that provides Spring Boot apps integration with the **Open Service Broker API**.

Spring Cloud It offers a set of primitives, helps in the leadership election, global locks, and one-time tokens. Currently, it supports

Spring Cloud It provides easy integrations and implementation of **HashiCorp** which provides service discovery, client-side load balancing with ribbon, control bus, and auto-configuration management.

Spring Cloud Security: It provides a set of primitives for building secure Spring Boot microservices apps using OAuth2 and SSO tokens in.

Spring Cloud Sleuth: It provides distributed tracing of microservices. Currently, it supports tracing.

Spring Cloud Data Flow It is an orchestration layer on top of Spring Cloud Task and Spring Cloud Stream for batch processing. It provides intuitive DSL, drag-and-drop GUI to create data pipelines and REST APIs together simplifies the overall orchestration of microservice-based data pipelines.

Spring Cloud It provides a lightweight event-driven microservices framework to quickly build applications that can connect to external systems through a shared messaging broker. It also provides a simple declarative model to send and receive messages using Apache Kafka or RabbitMQ between Spring Boot based microservices apps.

Spring Cloud Stream App They are standalone, executable applications, that communicate over messaging middleware such as Apache Kafka and RabbitMQ. They can run independently on **Cloud Apache Apache** or your local system.

Spring Cloud It's a framework to create short-lived microservices that perform finite amounts of data processing.

Spring Cloud Task App It's a Spring Boot application that may be any batch process including Spring Batch jobs that do not run forever, and they end/stop after a scheduled finite period of data processing.

Spring Cloud It provides integration with open-source **Apache**

Spring Cloud It provides easy integration to backend services like databases, message brokers, and so on.

Spring Cloud It provides the Spring Boot CLI plugin for creating Spring Cloud apps quickly and easily in Groovy. It launches services like **Config Server** from the command line.

Spring Cloud It's a verifier tool that enables **Consumer-Driven Contract** development of JVM-based applications. It is shipped with **Domain Definition Language** which is written in Groovy or YAML.

Spring Cloud It provides an API gateway using Spring MVC for smart routing, API orchestration and weight-based load balancing.

Spring Cloud It provides integrations for Spring Boot apps with OpenFeign through autoconfiguration and binding to the Spring environment and other Spring programming models. Feign is a Java-to-HTTP client binder inspired by and Feign's first goal is to reduce the complexity of binding denominators uniformly to HTTP APIs regardless of RESTfulness.

Spring Cloud It helps with the implementation of business logic via functions as a service (FaaS). It supports serverless architecture, as well as the ability to run standalone.

Official documentation of Spring Cloud:

Creating a Spring Boot microservice project using Spring.io

The following are the steps to create a simple Spring Boot microservice project using **Spring**

We will first create a Spring Boot project from the following website and download this project on a local machine:

<https://start.spring.io/>

Source code reference: <https://github.com/rajivmca2004/catalogue-service>

The following screenshot depicts the Spring initializr portal:



Project

Maven Project Gradle Project

Language

Java Kotlin Groovy

Spring Boot

2.4 (SNAPSHOT) 2.3.2 (SNAPSHOT) 2.3.1 2.2.9 (SNAPSHOT)
 2.2.8 2.1.16 (SNAPSHOT) 2.1.15

Project Metadata

Group	com.online.store.demo
Artifact	catalogue-service
Name	catalogue-service
Description	Demo project for Spring Boot
Package name	com.online.store.demo.catalogue-service
Packaging	<input checked="" type="radio"/> Jar <input type="radio"/> War
Java	<input type="radio"/> 14 <input type="radio"/> 11 <input checked="" type="radio"/> 8

Figure 4.2: Spring initializr page

Add required dependencies by clicking on the **Add Dependencies** button. In this demo, we need these technologies:

Spring Data

Spring Micrometer

Spring Data JPA

H2 in-memory database

Spring Boot DevTools

Refer to [for dependencies and Maven plugins and repositories:](#)

<https://github.com/rajivmca2004/catalogue-service/blob/master/pom.xml>

In the next step, select the project dependencies and libraries:

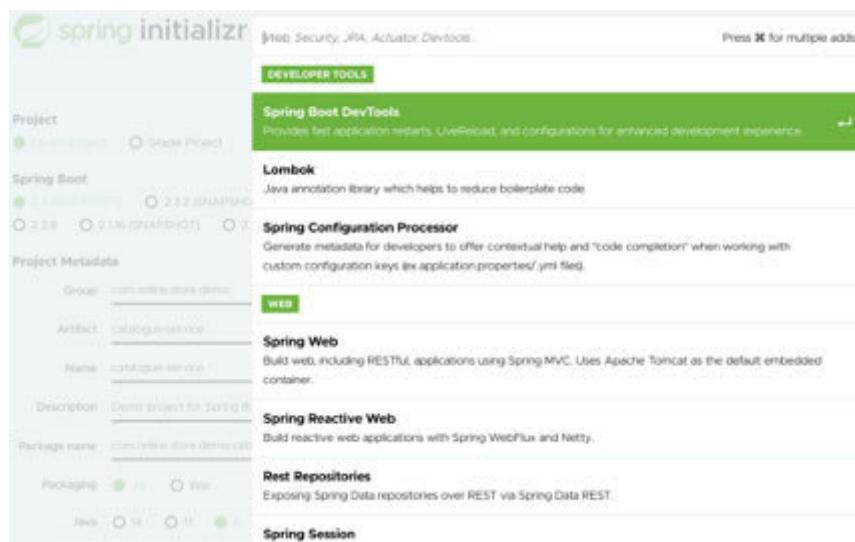


Figure 4.3: Spring initializr add dependency page

Download this project artifact on your system by clicking on the **Generate** button. It will have an initial template, Maven style directory structure, **pom.xml** with all the required dependencies, and the default **application.yaml** configuration file, which requires Spring Boot with the main class **CatalogServiceApplication.java** file.

Export this project as a Maven project in your IDE (Eclipse, **Spring Tool Suite** IntelliJ, and so on). My personal choice would be the STS IDE (based on the popular Eclipse IDE) and the same

will be demonstrated here. You can right-click on the **Project Explorer** window and select **Export** and select the project folder which you have downloaded from the **Spring Initializr** portal. You are all set to write business code and additional configuration.

The following is the screenshot of importing downloaded binaries/files using the STS IDE (Eclipse-based) by selecting the **Existing Maven Projects** option:

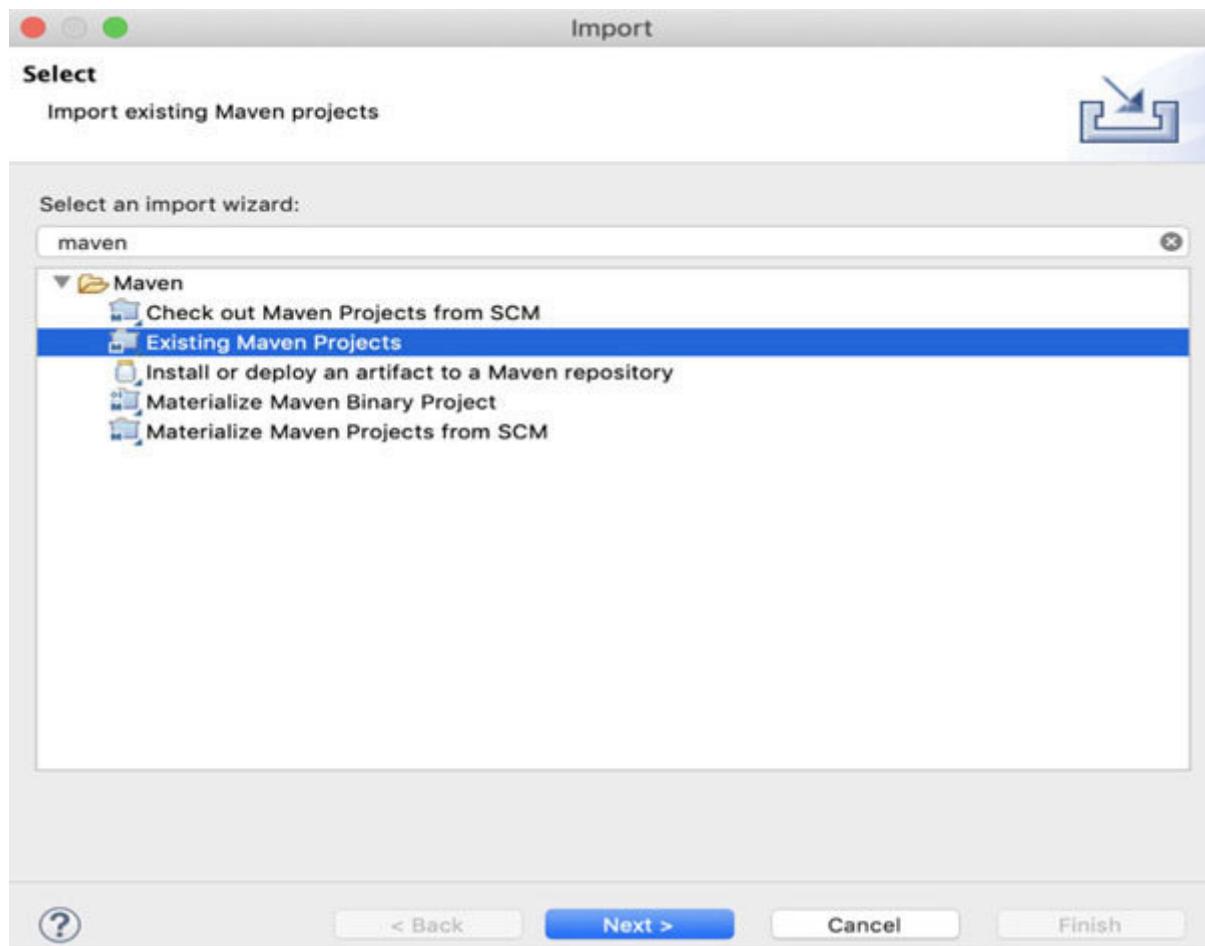


Figure 4.4: STS Eclipse import existing Maven project

Create a controller **CatalogueController** for your Spring Boot application and add your REST API endpoint definition. Create this

REST API:

Create a model **Catalogue** class to persist **Catalogue** data model (hard-coded) into local H2 database in-memory.

Create the **CatalogueRepository** repository and command-line runner **CatalogueRepoCLR** to persist product catalogue data to the H2 database.

Create Spring Boot application configuration file **application.yml** and add custom available port

Create the **CatalogueServiceApplicationTests** unit test class to test the basic functionality during build time.

Expose microservice as REST API. Now, test the application by opening

[

{

100.5

},

{

200.51

},

{

600

},

{

900.5

}

]

Try to Check if the service's health is good,

{

{

{

{

1,

"SELECT 1"

}

},

{

{

499963174912,

357886275584,

10485760

}

},

{

"UP"

}

}

}

Introduction to Spring Cloud Config Server

The Spring Cloud Config Server provides a dynamic way of centralized external configurations across the microservices. It externalized all configurations outside of the microservices apps. If any config change happens, no need to restart and redeploy apps. Config changes can be applied on the fly for the running microservices apps. In this approach, all the configurations are stored remotely outside of the applications.

The following diagram depicts the architecture of the Spring Cloud Config

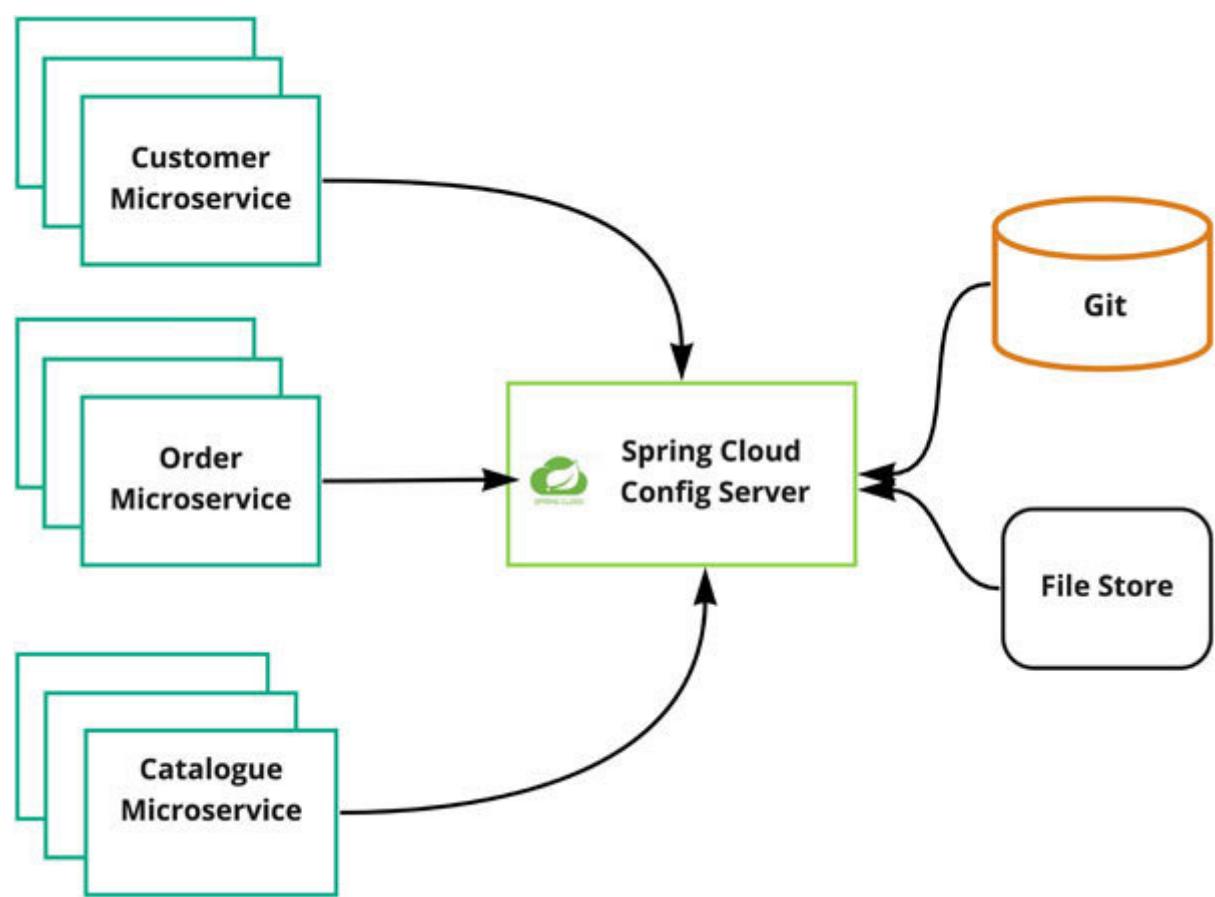


Figure 4.5: Spring Cloud Config Server architecture

These config changes are backed up and stored in Git repository to manage in a better way. All config changes can be managed at the central place for all microservices.

The concepts on both client and server map identically to the Spring environment and **PropertySource** abstractions. As a thumb rule, the order of the declared files is important for the **@PropertySource** annotation. If the same key is defined in two or more files, the value associated with the key in the last declared file will override any previous value(s).

A separate set of environmental configurations can be managed for non-production and production environments. It can also manage many versions of the configurations.

It's one of the twelve-factor principles of cloud-native applications which describe configurations that are likely to vary between different environments. According to this principle, the configuration should be stored in the environment.

Now, let's do some hands-on code exercise by a sample online store Spring Boot client application. In this coding exercise, we will create a Spring Cloud config server and externalize the configuration using the Spring Cloud config. Online store Spring Boot client will consume these external configurations using GitHub.

The source code of this sample Spring Cloud config is available here:

Spring Cloud config server:

<https://github.com/rajivmca2004/spring-config-server-cloud-demo>

Spring Cloud config client:

<https://github.com/rajivmca2004/product-service-spring>

Spring client config property ty files:

<https://github.com/rajivmca2004/client-config>

There are two parts of the Spring Cloud config server for this sample project:

Spring Cloud config Provider of central configuration.

Spring Cloud config Consumer of central configuration.

Prerequisites

These are the basic requirements to proceed with the sample code exercise:

JDK 8+

Spring Boot v2.2.7+

Spring Cloud v2.2.7+

GitHub account

Maven

Git client

[Building Spring Cloud config server](#)

Let's build a Spring Boot microservice app **spring-config-server-cloud-demo** for the Spring Cloud Config Server.. It will run as a standalone service that will run on containers and directly point to the Git code repo server. In this sample code exercise, we will use GitHub public code repository. Let's get started:

Instead of using Git repo, you can also use a local file system and point to the local files with the file path.

Add project

Add these required dependencies for Spring Cloud config server in Maven's **pom.xml** file:

Activate config

Now, we will activate the config server in this Spring Boot app by adding the **@EnableConfigServer** annotation. We just need to add this Java code and the rest will be taken care of by configuration code. This magical annotation will take care of everything automatically:

```
@EnableConfigServer
```

```
@SpringBootApplication
```

```
CentralizedConfigurationServiceApplication {
```

```
    main(String[] args) {
```

```
        args);
```

```
    }
```

```
}
```

Create external config properties and push to

We need to create configuration properties for all non-production and production environments and push them to GitHub. It provides a single source of truth that will be hosted on the Git server. In this case, we are using GitHub for this sample app:

```
mkdir client-config
```

```
cd client-config
```

```
git init
```

```
git remote add origin
```

```
git branch -M master
```

```
git push -u origin master
```

```
echo "myenv.name=default environment" > product-service-spring.properties
```

```
echo "myenv.name=default environment" > product-service-spring-development.properties
```

```
echo "myenv.name=default environment" > product-service-spring-production.properties
```

```
git add+96/.
```

```
git commit -m "First external configuration"
```

```
git push -u origin master
```

Create the Spring

In this step, we will add application configuration. This is an important configuration file, where we define Spring Cloud Config Server properties. Here, we define GitHub URI where we store external configuration.

To keep the local repository branches clean and sync up with remote branch, we need to set **deleteUntrackedBranches** property. It will make the Spring Cloud config server force delete untracked branches from the local repository.

If you are using a private GitHub repository, a secure code repository of your own, or any public cloud-hosted then you need to also provide the Git server's user ID and passwords:

server:

port: 8888

spring:

application:

name: spring-config-server-cloud-demo

jmx:

default_domain: cloud.config.server

```
cloud:
```

```
  config:
```

```
    server:
```

```
      git:
```

```
        uri:
```

```
        force-pull: true
```

```
        deleteUntrackedBranches: true
```

```
management:
```

```
  endpoint:
```

```
    health:
```

```
      show-details: always
```

**Make sure you create a master branch in the GitHub repo.
GitHub creates the main branch by default, which is not
supported by Spring Cloud config.**

Verify external config properties using REST API.

In this last step, the config server has already connected to your Git server and cloned the code to the local host's persistent disk. We can verify this by running the REST API:

```
$ curl http://localhost:8888/product-service-spring/default
```

We have used the Chrome Postman REST client throughout this book for testing purposes. You can download the same for the Google Chrome browser.

The following screenshot depicts the Postman REST client when we hit product-service-spring's **/default** API:

The screenshot shows the Postman interface with a GET request to `http://localhost:8888/product-service-spring/default`. The response body is a JSON object:

```
1 {
2   "name": "product-service-spring",
3   "profiles": [
4     "default"
5   ],
6   "label": null,
7   "version": "60b14989f2eb544f27c7dcc787c442a1bef9964f",
8   "state": null,
9   "propertySources": [
10    {
11      "name": "https://github.com/rajivmca2004/client-config/product-service-spring.properties",
12      "source": {
13        "myenv.name": "default env"
14      }
15    }
16  ]
17 }
```

The status bar at the bottom indicates `Status: 200 OK`, `Time: 1156 ms`, and `Size: 448 B`.

Figure 4.6: REST response with config details

[Building Spring Cloud config client](#)

The next step is to create a sample online store Spring Boot app **product-service-spring** to consume external environment configurations from properties config files.

Add project dependencies.

Let's start by adding the required project dependencies in Maven's

Define local configuration in Spring

Now, let's create a local Spring Boot application configuration and give a URI path for the Spring Cloud config server. If you run this application on your local laptop, you can use localhost, otherwise, you can use the actual path of the Spring Cloud config server on production or other remote cloud-hosted environment.

Also, we need to define an active profile based on the production and non-production environments:

spring:

application:

name: product-service-spring

cloud:

config:

uri: http://localhost:8888

profiles:

active: development

management:

endpoints:

web:

exposure:

include: refresh

server:

```
port : 9010
```

A Bean marked with the **RefreshScope** annotation will be recreated when a configuration change occurs and a **RefreshScopeRefreshedEvent** object is triggered.

Whenever a configuration change occurs in the Git repository, we can trigger a **RefreshScopeRefreshedEvent** object by hitting the Spring Boot actuator **/refresh** API. The **refresh** endpoint should be enabled by setting this property:

```
management.endpoints.web.exposure.include=refresh
```

The **curl** command for the **actuator refresh** endpoint:

```
curl -X POST \
      -H 'content-type: application/json' \
      -d '{}'
```

This will update the configuration values on the config client on demand whenever any changes have been pushed. It will restart the client Spring boot app and reload beans.

Create Spring Boot controller

In this step, we will create a Spring Boot controller class with **@RefreshScope** and we will use the environment variable Let's create a REST API to test this value:

```
@RestController
```

```
@RefreshScope
```

```
CatalogueController {
```

```
    Logger logger =
```

```
    @Autowired
```

```
    private CatalogueRepository catalogueRepository;
```

```
    public Object fetchProducts() {
```

```
        List products = catalogueRepository.findAll();
```

```
        product catalogue
```

```
        return products;
```

```
}
```

server

```
private String envName;  
  
public String getMyEnv() {  
  
}  
}
```

In this example, the configuration can be injected into the client microservices apps by simply using

Verify config value by REST

Finally, we can verify if that config **environment** property has been injected by the exposed REST API which we have created in client Spring Boot microservice app In this example, we are setting an active profile for the development environment. Likewise, we can set active profiles for other production and non-production environments:

environment

Now, hit the `/checkEnvName` API. This screenshot is showing API response default environment because, in the `product-service-spring` Spring Boot application's `application.yml` file, development `env` is set up like this:

The screenshot shows the Postman application interface. At the top, there is a header bar with 'GET' selected, a URL field containing 'http://localhost:9010/checkEnvName', and a 'Send' button. Below the header are tabs for 'Params', 'Authorization', 'Headers (6)', 'Body', 'Pre-request Script', 'Tests', and 'Settings'. The 'Params' tab is active. Under 'Params', there is a table with one row: 'Key' (Value) and 'Value' (Description). The table has columns for 'KEY', 'VALUE', 'DESCRIPTION', and '*** Bulk Edit'. Below the table are tabs for 'Body', 'Cookies', 'Headers (5)', and 'Test Results'. The 'Body' tab is active. On the right side of the interface, there is a status summary: 'Status: 200 OK', 'Time: 6 ms', and 'Size: 186 B'. Below the status summary are buttons for 'Save Response' and a search icon. At the bottom left, there is a note: '1 My default Development'.

Figure 4.7: API response to check current environment

Fault tolerance using the Spring Cloud Circuit Breaker

In this section, we will go through the fault-tolerance concept of distributed microservices environment. The goal of fault tolerance of applications is to ensure high availability (HA) and business continuity (BC). It returns profit for the business by building the customer's trust and providing a better experience. We always recommend this as a high-priority item for the architects while designing highly distributed system.

In the following architecture, the **Order** microservice is calling the **Catalogue** and **Customer** microservices. If any of these two services are down or have some application issues, then the circuit breaker will take control and redirect the client request to another backup service or give a friendly error message to retry based on use cases:

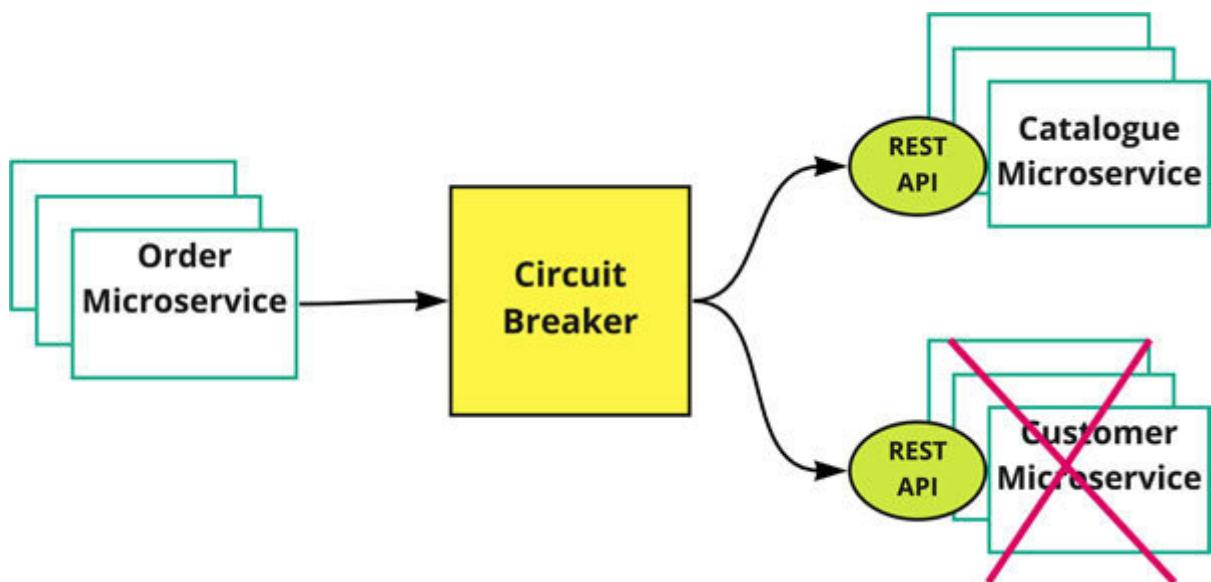


Figure 4.8: Circuit breaker architecture with use case

Let's understand what is **fault** It's a design principle that refers to the ability of an application or system to continue operating without any interruption in case of partial failure of one or more components. It makes applications recover failed components automatically and handles the ongoing operations gracefully with minimal loss. There are many use cases for fault tolerance like one microservice is responding intermittently due to some intermittent issue or during upgrades or patching. To solve this issue, we should design fault-tolerant systems by duplicating, scaling, and retrying mechanisms.

Kubernetes load balancers also provide fault tolerance, so when any server/container is down, the workload is redirected to another server. It provides a reliable and sustainable application platform by avoiding a **single point of failure**

In a cloud-native pattern, fault tolerance can be implemented by distributing load to multiple **Availability Zones Data Centers** and cloned containers.

In Spring Boot apps, there are multiple ways to implement using:

Netflix Hystrix OSS

Resilience4j

Spring retry

Sentinel

Please refer to [Chapter](#) Microservice Design Pattern, for a detailed explanation of circuit breaker design patterns with use cases.

Introduction to Resilience4j

Netflix **Hystrix** is currently in the maintenance phase, there is no plan for further development. It's recommended to use other mentioned alternatives of these APIs. We will recommend which has other features also apart from circuit breaker and better than other circuit breaker APIs. In this chapter, we will create a sample app using Resilience4j.

Spring Cloud circuit breaker offers an orchestration layer so that any implementation can be done easily with many more features:

Resilience4j has these major features:

It has active community support.

It's lightweight based on Vavr.

It supports functional and reactive Java programming.

It has an automatic retry mechanism.

It has rate limiter.

It supports thread pool isolation.

It has built-in micrometer with good integration with Grafana and Prometheus monitoring tools.

It has better developer-level control for circuit breakers and other features.

Good annotation support with Spring Boot.

These are the main modules of Resilience4j:

Circuit breaking.

Rate limiting.

Automatic retrying.

Result caching.

Timeout handling.

Bulkheading.

In this chapter, we will only cover the circuit breaker module. Please refer to Spring Cloud's official documentation for other components.

Implementing Resilience4J circuit breaker

In this section, we will implement the Spring Boot cloud circuit breaker using Resilience4j with a sample application.

Prerequisites

These are the basic requirements to proceed with the sample code exercise:

JDK 8+

Spring Boot v2.2.7+

Spring Cloud starter circuit breaker Resilience4j v1.0.4+

GitHub account

Git client

Maven

Source code references:

Download the **order-service** consumer microservice from GitHub using the following link:

<https://github.com/rajivmca2004/order-service>

Download the **catalogue-service** provider microservice from GitHub using the following link:

<https://github.com/rajivmca2004/catalogue-service>

Let's do some hands-on exercise. In this sample code exercise, we will add Maven dependencies, and then create an auto-configuration bean for the Spring Boot app:

Add Maven dependencies for

We need to add these latest Spring Cloud starter project dependencies in the cloned project. Please check the latest version of this Spring Boot library:

org.springframework.cloud

spring-cloud-starter-circuitbreaker-resilience4j

Create an auto-configuration

In this step, we will create an auto-configuration bean in the main **OrderServiceApplication** class:

```
/*
 * Global circuit breaker custom configuration
 */

@Bean
public Customizer globalCustomConfiguration() {
    // Timeout duration= 5 seconds
    TimeLimiterConfig timeLimiterConfig =
        /* Threshold time= 50%
         * Configures the failure rate threshold in percentage. If
         * the failure rate is
         * or greater than the threshold, the CircuitBreaker
         * transitions to
         * open and starts must be greater than
         * greater than 1.. Default value is 50%.
        */
}
```

*

* Wait duration: ⏺ ms

* waitDurationInOpenState the wait duration which specifies how long the CircuitBreaker should stay open

* slidingWindowSize= ⏺

* slidingWindowSize the size of the sliding window when the CircuitBreaker is closed.

*/

CircuitBreakerConfig circuitBreakerConfig =

```
return factory -> factory.configureDefault(id -> new Resilience4JConfigBuilder(id)
```

```
    .timeLimiterConfig(timeLimiterConfig).circuitBreakerConfig(circuitBreakerConfig).build());
```

}

We need to define these important configurations to configure a default circuit breaker:

Timeout Time duration required to complete the task.

Threshold Its failure rate threshold in percentage. If the failure rate is equal to or greater than the threshold, the **CircuitBreaker** transitions to open and starts short-circuiting calls. The threshold must be greater than 0 and not greater than 100. The default value is

Wait Specifies how long the circuit breaker should stay open.

Size of the sliding window when the circuit breaker is closed.

This is the default configuration for all the REST APIs. If you have multiple circuit breaker configurations, then you can pass multiple circuit breaker arguments in the circuit breaker bean.

Build a circuit breaker and add a

Spring Boot follows MVC architecture. Normally, the client requests go to the controller class first and this class forwards the requests to the service class. Here, the consumer service calls the external microservices REST API endpoints using Spring **RestTemplate** API.

Now in this step, we need to first define the service breaker factory in the **OrderServiceImpl** service class:

```
@Autowired
```

```
private CircuitBreakerFactory circuitBreakerFactory;
```

Now, let's create an instance of the circuit breaker in the **Service** method:

```
@Override
```

```
public String fetchCatalogueServiceCircuitBreaker() throws  
URISyntaxException {
```

```
    String catalogueResponse =
```

```
    CircuitBreaker circuitBreaker =
```

```
        URI catalogueUri = new + catalogueResourceHost + ":" +  
catalogueResourcePort +
```

```
try {
```

Calling CATALOGUE SERVICE-Circuit
breaker*****catalogueUri=> " + catalogueUri);

```
catalogueResponse= circuitBreaker.run() -  
restTemplate.getForObject(catalogueUri, throwable -
```

```
getDefaultCatalogueList();  
  
    } catch (Exception e) {  
  
        logger.error(e.getMessage());  
  
        e.printStackTrace();  
  
    }  
  
    return catalogueResponse;  
  
}
```

Also, we need to create a **fallback** method to call the alternate services if the remote external service is down or not available at that moment. You can also return a meaningful error or warning message. This **fallback** method will be called by the circuit breaker **run** method as a separate method argument:

```
// Circuit breaker-Fall back method if catalogue REST API won't respond
```

```
private String getDefaultCatalogueList() {
```

Calling Fallback API

```
/*
```

```
* Fallback business logic goes here  
*/  
  
Fallback  
}
```

Build a REST controller to use circuit breaker

In this step, we will call the preceding circuit breaker **Service** method by a REST API method in the Spring boot MVC's controller class:

```
//Test Spring Cloud Circuit breaker using Resileince4j
```

```
public String fetchCatalogue() throws URISyntaxException {  
  
    return orderService.fetchCatalogueServiceCircuitBreaker();  
  
}
```

Test and verify the circuit breaker

Now, we are all set to go and verify if this new default circuit configuration works.

This **/catalogues** API endpoint of **order-service** microservice apps internally calls service method **/catalogues** API which is exposed by external **catalogue-service** on port To test this feature, we need to first shutdown **catalogue-service** service.

Since we have already added the circuit breaker configuration, which will provide fault tolerance and try to call external **catalogue** since this service is down, client requests will be forwarded to fallback methods for work around alternate actions.

The following screenshot depicts the API verification page after hitting from Postman REST client:

The screenshot shows the Postman interface with the following details:

- Method: GET
- URL: <http://localhost:8012/catalogues>
- Params tab is selected, showing a table for Query Params:

KEY	VALUE	DESCRIPTION
Key	Value	Description

- Headers tab shows 6 headers.
- Body tab is selected, showing the response body:

```
1 Calling Fallback API
```
- Headers tab shows 5 headers.
- Status bar indicates Status: 200 OK and Time: 24 ms.

Figure 4.9: API verification

Circuit breaker follows default configurations which we have already provided in *Step* So, **order-service** will keep on calling external **/catalogues** service based on timeout, threshold, and other configurations.

Conclusion

In this chapter, we discussed how to create microservices using Spring Boot and inject cloud-native twelve-factor app features like circuit breaker using Spring Cloud. We covered all the major Spring cloud-related components and their fundamentals also with a sample code exercise. We covered how to create a Spring boot project and add dependencies using the **Spring Initializr (Spring.io)** portal. We also discussed Spring Cloud config server, circuit breaker configuration and customization techniques. In the last section, we discussed implementing fault tolerance using Spring Cloud Circuit breaker with Resilience4j.

In the next chapter, we will discuss batch microservices, Spring batch, Spring Cloud task, and Spring batch best practices. We will also discuss creating a data pipeline using **Spring Cloud Data Flow** *Happy learning!*

Points to Remember

Spring is a framework that contains Spring Boot and Spring Cloud API services

These active project references have been taken from the official documentation of Spring Cloud. Please refer to the latest information from the official page:

<https://spring.io/projects/spring-cloud>

External configuration is one of the twelve-factor principles of cloud-native applications which describe the *configuration* that is likely to vary between different environments. According to this principle, the configuration should be stored in the environment.

Spring Cloud config server can be backed up by Git and local file systems both.

Microservices applications need not be restarted for any config changes at central config changes managed through the Spring Cloud config.

In this chapter, we only covered the circuit breaker module. Please refer to Spring Cloud's official document for other components.

Spring Boot works on **Model View Controller** design pattern.

Netflix OSS is currently in the maintenance phase.

We will recommend Resilience4j circuit breaker. It has more features other than a circuit breaker.

Key terms

Model View Controller.

Authorization and Authentication.

Structured Query Language.

No Structured Query Language.

Object Relational Mapping.

Java Persistence API.

JSON Web Token.

Open-Source Software.

Platform as a Service.

Single Sign-On.

Single Point of Failure.

Java achieve.

Open Service Broker.

Availability Zone.

Data Centers.

Transactions per Seconds.

References

Spring <https://spring.io/projects/spring-boot>

Spring <https://spring.io/projects/spring-cloud>

Spring Initializr <https://spring.io>

CHAPTER 5

Batch Microservices

Until now, we have covered independent microservices. Every microservice is built on a **Single Responsibility Principle (SRP)**. There are many use cases where we need to run scheduled short-lived jobs or batch jobs like bank fund transfer or reward generation in online store use cases. These jobs run once for a short life that triggers on scheduled time or action. Sometimes, these batch jobs run in parallel or a sequence.

In this chapter, we will cover the fundamentals of batch microservices, use cases, introduction to **Spring Cloud Spring** and **Spring Cloud Data Flow** using Kafka with sample code exercises. We will go through a couple of Spring Batch best practices, batch auto-scaling techniques, batch orchestration, and composition methods for sequential or parallel batch processing. We will also go through active operation for high availability between data center techniques. In the end, we will go through Spring Cloud Task, Spring Batch alerting, and monitoring tools and techniques.

Let's get started! Happy learning!

Structure

In this chapter we will cover the following topics:

Introduction to batch microservices

Introduction to Spring Cloud Task

Spring Cloud Task features

Getting started with Spring Cloud Task

Introduction to Spring Batch

Getting started with Spring Batch application

Introduction to **Spring Cloud Data Flow**

SCDF objectives

SCDF architecture

SCDF modules

Data flow server

Skipper server

Database

Security

Stream processing

Batch processing

Monitoring

CLI and API library

Installing SCDF

Getting started with SCDF

Introducing to Spring Cloud Stream

Comparing Spring Cloud Task and Spring Batch

Best practices for Spring Batch

Spring Batch auto-scaling

Spring Batch orchestration and composition

Active-active operation for **high availability** between data centers

Alerts and monitoring of Spring Cloud Task and Spring Batch

Objective

After studying this chapter, you should be able to learn batch microservices, Spring Batch, Spring Cloud Task features with a sample batch microservice application. This chapter will help you in-depth understanding of SCDF and how to register a Spring Cloud Task on SCDF and launch it from the SCDF web dashboard. This chapter will help you to learn about Spring Cloud Stream, comparison of Spring Cloud Task, and Spring Batch. You will also learn the best practices, auto-scaling, orchestration, and composition of Spring Batch. You will also learn how to monitor Spring Cloud Task and Spring Batch.

Introduction to batch microservices

A batch microservice is a type of short-lived process or one-time batch execution that launches as the result of some triggers like a timer, due to some action or executed on demand. The following diagram shows the integration between Spring Batch projects:

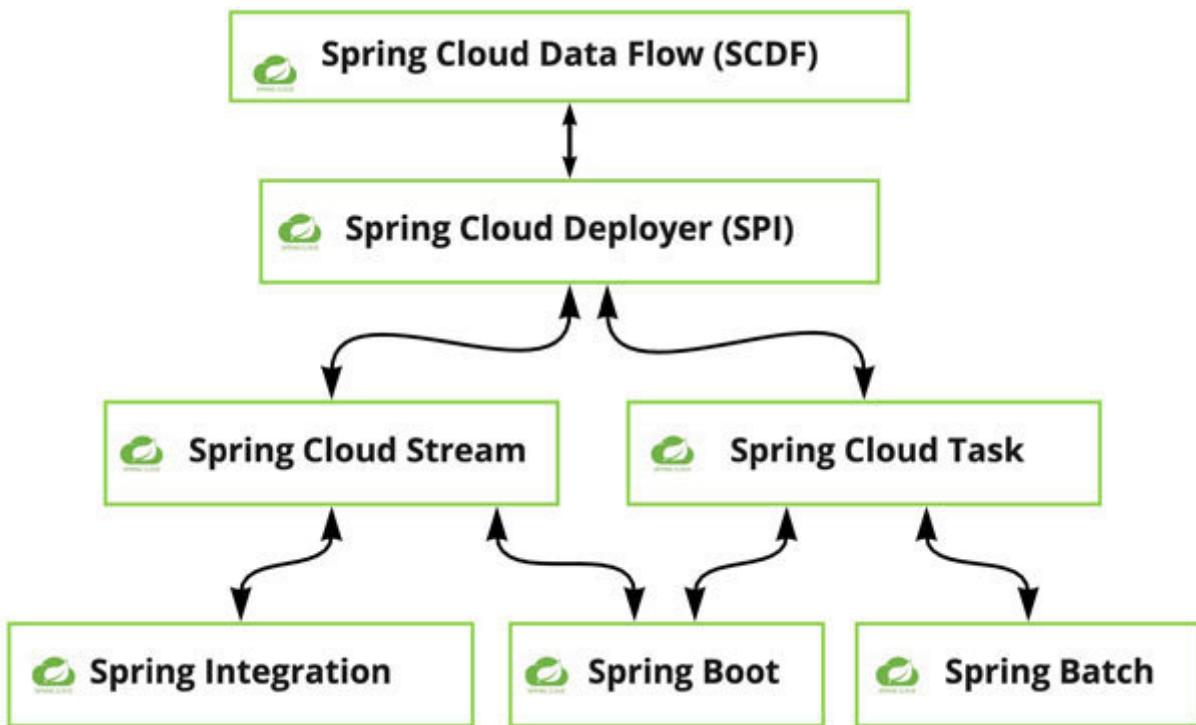


Figure 5.1: Batch microservices components

There are many use cases where we need to run scheduled batch jobs for use cases like bank fund transfer or reward generation an online store. These jobs run just once for a short life which triggers on scheduled time or action. Sometimes, these batch jobs

run in parallel or an ordered sequence. These are a couple of use cases of batch microservices.

Introduction to Spring Cloud Task

Spring Cloud Task is a generic framework to create short-lived services, which can run independently based on demand and scheduled trigger. It also supports easy integration with SCDF.

In general, these perform simple tasks on-demand or in a scheduled manner and then terminate after completion. It records the lifecycle events of a given task. The lifecycle consists of a single task execution that can be done by annotating Spring Boot applications with **@EnableTask** annotation.

Spring Cloud Task uses a in-memory H2 database; however, SQL database can also be integrated which could be **MySQL** or **PostgreSQL** for a similar to the Spring Batch

Currently, the following are the databases that are supported:

H2 (default)

MySQL

PostgreSQL

HyperSQL Database

Oracle

SQL Server

Spring Cloud Task features

These are a few important Spring Cloud Task features:

@EnableTask annotation to auto-configure a **TaskRepository** and **SimpleTaskConfiguration** by default.

Integrates with Spring Batch enabling jobs from a task.

Supports highly scalable, distributed batch architectures like remote chunking, partitioning, and running as a cloud-native microservice.

Integrates with Spring Cloud Stream so that the stream events may trigger the tasks, and tasks events may be consumed by streams.

Getting started with Spring Cloud Task

In this section, we will create a short-lived microservice application using Spring Cloud Task on underlying Spring Boot library.

Spring Cloud Task provides features to run tasks dynamically, allocates resources on demand, control, and retrieve result or output after the task is completed. Let's do this hands-on.

You can refer to Spring Cloud Task official documentation for advanced-level configuration:

Prerequisite

These are basic installation requirements to build a Spring Task:

Java 8+

Spring Cloud Task v2.2.3+

Spring Boot v2.2.3+

Git CLI

Maven CLI

SCDF v2.6.3+ installed on the **Kubernetes** cluster

K8s cluster to install all the SCDF related binaries

Source code reference: <https://github.com/rajivmca2004/spring-cloud-task-demo>

Let's create a simple **Hello world** Spring Cloud Task on the Spring Boot library:

We will follow the same approach, by creating this **spring-cloud-task-demo** project on the Spring Initializr web portal:

<https://start.spring.io/>

The screenshot shows the Spring Initializr interface at <https://start.spring.io/>. The left sidebar has a menu icon. The main area is titled "spring initializr". On the left, there are sections for "Project" (Maven Project selected), "Language" (Java selected), "Spring Boot" (2.3.5 selected), "Project Metadata" (Group: com.online.store.demo, Artifact: spring-cloud-task-demo, Name: spring-cloud-task-demo, Description: Spring Cloud Task Demo project for Spring Boot, Package name: com.online.store.demo.springcloudtaskdemo, Packaging: Jar selected, Java version: 8 selected), and "Dependencies" (Task: SPRING CLOUD selected, MySQL Driver selected). The right side shows detailed descriptions for the selected dependencies.

Project
Maven Project (selected)
Gradle Project

Language
Java (selected)
Kotlin
Groovy

Spring Boot
2.4.0 (SNAPSHOT)
2.3.6 (SNAPSHOT)
2.2.11
2.4.0 (RC1)
2.2.12 (SNAPSHOT)
2.1.18

Project Metadata
Group: com.online.store.demo
Artifact: spring-cloud-task-demo
Name: spring-cloud-task-demo
Description: Spring Cloud Task Demo project for Spring Boot
Package name: com.online.store.demo.springcloudtaskdemo
Packaging: Jar (selected)
Java: 15, 11, 8

Dependencies
ADD DEPENDENCIES... ⌘ + B

Task SPRING CLOUD
Allows a user to develop and run short lived microservices using Spring Cloud. Run them locally, in the cloud, and on Spring Cloud Data Flow.

Spring Data JPA SQL
Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

H2 Database SQL
Provides a fast in-memory database that supports JDBC API and R2DBC access, with a small (2mb) footprint. Supports embedded and server modes as well as a browser based console application.

MySQL Driver SQL
MySQL JDBC and R2DBC driver.

Figure 5.2: Spring Initializr for *spring-cloud-task-demo*

We need to add these Spring Cloud Task Maven dependencies in the **pom.xml** file. Also, we need to add an RDBMS database for the persistence of Spring Cloud Task/Batch job metadata. Here, in this example, we are going to use MySQL RDBMS on a local machine for local testing. We need **Java Persistence API** as an abstraction layer on top of the database layer to make the data layer generic, so that it can be replaced by any other database. Later on, we can connect with MySQL on the Kubernetes cluster.

We can use the **Testcontainers** tool for MySQL testing.

Testcontainers is a Java library that supports JUnit tests, providing lightweight, throwaway instances of common databases, Selenium web browsers, or anything else that can run on a Docker-based container or Kubernetes.

It will be a Spring Boot starter project and on top of that, we will add Spring Cloud Task-related binaries dependencies. You can check the latest version on Maven central portal:

<https://mvnrepository.com/>

We need to add the following Maven dependencies in the **pom.xml** file:

org.springframework.boot

spring-boot-starter-data-jpa

org.springframework.cloud

spring-cloud-starter-task

org.springframework.cloud

spring-cloud-task-core

mysql

mysql-connector-java

com.h2database

h2

test

Add the following **spring-cloud-dependencies** to maintain the artefact and version dependency of Spring Cloud project:

We need Maven plugin and Docker configuration files to create Docker images and deploy on Kubernetes.

Add **@EnableTask** annotation on Spring Boot's main **SpringCloudTask DemoApplication** class for bootstrapping of the Spring Boot like this sample source code configuration:

```
@SpringBootApplication
```

```
@EnableTask
```

```
SpringCloudTaskDemoApplication {
```

```
// Your business logic
```

```
}
```

This annotation uses **SimpleTaskConfiguration** class that in turn registers the **TaskRepository** and its infrastructure code. By default, an in-memory map is used to store the status of the

We need to add this MySQL database connection code in **application.yml** file along with other Spring Boot app specific configurations. You can use MySQL server credentials. Also, add JPA configuration like this following configuration:

spring:

 application:

 name: spring-cloud-task-demo

 datasource:

 url: jdbc:mysql://localhost:3306/springcloudschema?useSSL=false

 username: root

 password:

 jpa:

 hibernate:

```
    ddl-auto: create-drop
```

```
properties:
```

```
    hibernate:
```

```
        dialect: org.hibernate.dialect.MySQL5Dialect
```

```
batch:
```

```
initialize-schema: always
```

```
maven:
```

```
remoteRepositories:
```

```
springRepo:
```

```
url: https://repo.spring.io/libs-snapshot
```

Configure data source to persist all Spring Cloud Task-related job metadata. Here, we are going to use MySQL for a demo application. We have already configured database connection details in **application.yaml** file. We need to pass the **DataSource** as a constructor, which will auto connect with the database and make necessary default setting in **DemoTaskConfigurer** class like the following:

```
DemoTaskConfigurer extends DefaultTaskConfigurer {
```

```
public DemoTaskConfigurer(DataSource dataSource) {  
}  
  
}  
}
```

To manage the lifecycle of Cloud Task, we need a **TaskListener** class that will implement Spring Cloud Task has these three life-cycle methods: and Please refer to this sample code snippet:

```
TaskListener implements TaskExecutionListener {
```

```
    Logger LOGGER =
```

```
    @Override
```

```
    onTaskEnd(TaskExecution arg) {
```

```
        LOGGER.info(<<Task completed!>>);
```

```
}
```

```
    @Override
```

```
    onTaskFailed(TaskExecution arg, Throwable arg) {
```

```
    LOGGER.info(<<Task failed!>>);  
  
}  
  
@Override  
  
onTaskStartup(TaskExecution arg) {  
  
    LOGGER.info(<<Task Started!>>);  
  
}  
  
}
```

In the final config step, we will add an inner class in Spring Boot's main class **SpringCloudTaskDemoApplication**, and declare a **TaskListener** bean that will manage the life cycle of Spring Cloud Task and register listeners available from **TaskExecutionListener** interface. Please refer to this sample code snippet:

```
@Component  
  
HelloWorldApplicationRunner implements ApplicationRunner {  
  
    @Override
```

```
run(ApplicationArguments arg) throws Exception {  
  
    LOGGER.info(<<Hello World from Spring Cloud  
Task!>>);  
  
}  
  
}  
  
@Bean  
  
public TaskListener taskListener() {  
  
    TaskListener();  
  
}  
  
}
```

Now, we can run this Spring Cloud Task. It will log all job-related information on the console log and database.

Introduction to Spring Batch

Spring Batch is a framework that supports on-demand and scheduled short-time lived tasks which is run by some trigger or action. It provides APIs to write batch jobs easily.

Spring Batch provides reusable functions and libraries to process a large number of records typically for bulk tasks like scheduled bank fund transfer including logging, transaction management, job processing statistics, job restart, skip, and resource management. It persists all batch-related metadata into the database to track all the batch steps.

It also enables extremely large volume and high-performance batch jobs through optimization and partitioning techniques.

Spring Batch schedules and interacts with the job using job repo. A job can have more steps in sequence like **Extract, Transform, Load** job. In this process, every step of the Spring Batch does reading data from source, process, and filter based on some business logic and writing it to the destination. Spring Batch automates and enables batch steps in a systematic way and persists all step metadata into the database. If a batch fails due to some issue, then the Spring batch takes care and restarts the batch from where it left using that metadata from the persistent database.

Here are the key concepts of Spring Batch:

It's an independent entity that encapsulates an entire batch process including all the steps.

A job can have multiple steps. It's an independent unit of work. It contains all the required info. It completes its assigned task.

It's a group of key-value pairs that persisted in batch databases. It's controlled by the Spring Batch framework for execution context and operation.

It's a persistent job repository for all of the Spring Batch objects. It provides CRUD database operations on batch metadata.

It represents an interface to launch a job.

It provides inputs for the steps and reads one input at once.

It provides the output of the step once at once.

It's the process business logic of an item.

The following diagram is depicting Spring Batch architecture:

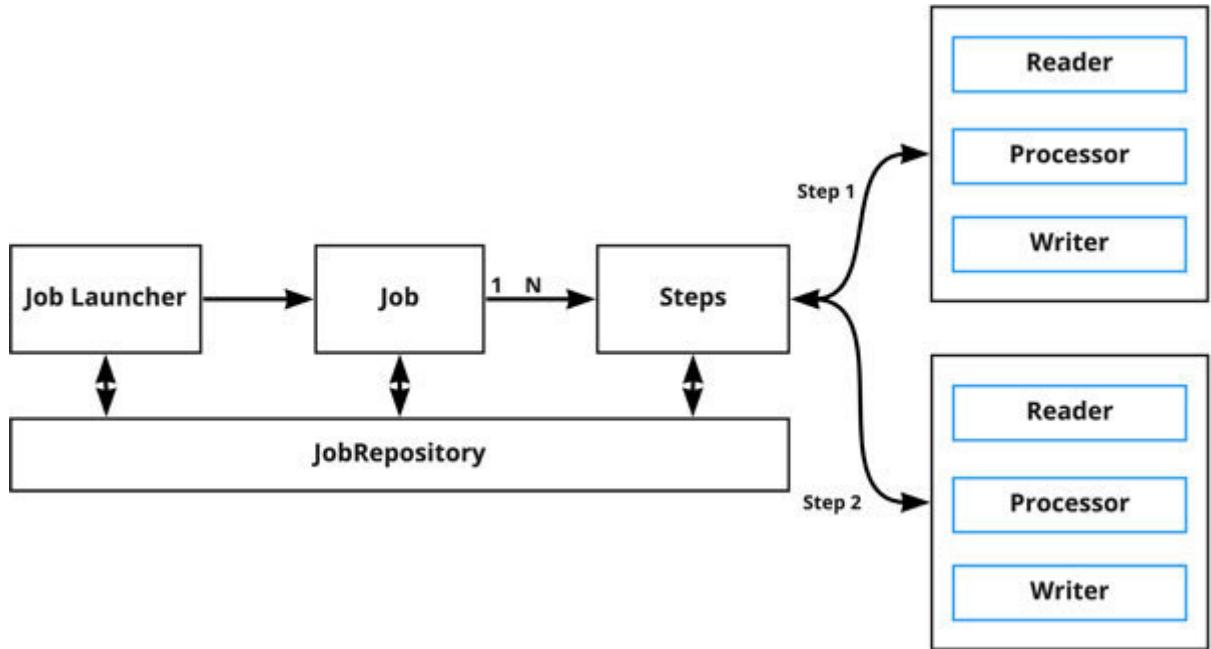


Figure 5.3: Spring Batch architecture

Getting started with the Spring Batch application

In this section, we will create a sample Spring Batch application from scratch. We will cover a sample scenario of an online store and develop a **Hello world** Spring Batch based on the same Spring Cloud Task app, which we have explained with source code in *Getting started with Spring Cloud Task* section above.

You can refer to **Spring Batch official document page** for an advance level of configuration:

Prerequisites

These are basic requirement to proceed with the sample code exercise:

Java 8+

Spring Batch v2.3.4+

Spring Boot v2.2.3+

Git CLI

Maven CLI

SCDF v2.6.3+ installed on **Kubernetes** cluster

K8s cluster to install all SCDF related binaries

Source code reference: <https://github.com/rajivmca2004/spring-cloud-task-demo>

Let's create a simple **Hello world** Spring batch app:

We will refer to the same project which we have created for creating Spring Cloud Task.

We need to add these Spring Batch Maven dependencies in the **pom.xml** file:

org.springframework.boot

spring-boot-starter-batch

org.springframework.cloud

spring-cloud-task-batch

Add **@EnableBatchProcessing** annotation on Spring Boot's main **SpringCloudTaskDemoApplication** class for bootstrapping of Spring Boot:

@SpringBootApplication

@EnableTask

@EnableBatchProcessing

```
SpringCloudTaskDemoApplication {  
  
    // Your job business logic  
  
}
```

One important note, whenever you need to onboard any Spring Batch to SCDF, you need to add an additional **@EnableTask** annotation.

In this final important step, a batch job class is needed to manage the life cycle. We will create a **SpringBatchJobConfiguration** class and add all the job configurations. In the following source code example, we have a single job and three steps:

Step This is a sample step that only prints a sample message.

Step Here, we have three methods which have their own specific responsibilities:

It reads country names from array list collection.

It will convert those country names into upper case.

It prints result.

Step This is a sample step that only prints a sample message. Refer to this source code of **SpringBatchJobConfiguration** class:

```
@Configuration

SpringBatchJobConfiguration {

    Logger LOGGER = LoggerFactory.getLogger(getClass());

    @Autowired
    private JobBuilderFactory jobBuilderFactory;

    @Autowired
    private StepBuilderFactory stepBuilderFactory;

    // This is a sample step which only prints a sample message

    @Bean
    public Step step1() {
        return stepBuilderFactory.get("step1")
                .tasklet(tasklet())
                .build();
    }

    @Override
    public void execute(StepExecution stepExecution) throws Exception {
        Tasklet tasklet = stepExecution.getStep().getTasklets().get(0);
        tasklet.execute(stepExecution);
    }
}
```

```
public RepeatStatus execute(  
    StepContribution contribution,  
    ChunkContext chunkContext)  
  
throws Exception {  
  
    LOGGER.info(<<**job\step`>>);  
  
    return RepeatStatus.FINISHED;  
  
}  
  
}).build();  
  
}  
  
@Bean  
  
public Step step`() {  
  
    .get(<<job\step`>>)  
  
    .String> chunk(`)  
}
```

```
.reader(  
  
new ListItemReader<>(Arrays.asList(<<india>>,  
          <<usa>>, <<singapore>>)))  
  
.processor(  
  
new ItemProcessorString<>() {  
  
    @Override  
  
    public String process(String item)  
  
    throws Exception {  
  
        LOGGER.info(<<job\step` => Processing  
of chunks::>>+item);  
  
        return item.toUpperCase();  
  
    }  
  
})  
  
ItemWriter() {  
  
    @Override
```

```
    write(  
        List<extends String> items)  
  
    throws Exception {  
  
        for (String item : items) {  
  
            LOGGER.info(<<job\step` => Writing of  
            chunks::>>+item);  
  
        }  
  
    }  
  
}).build();  
  
}  
  
// This is a sample step which only prints a sample message  
  
@Bean  
  
public Step step() {
```

```
Tasklet() {  
  
    @Override  
  
    public RepeatStatus execute(  
        StepContribution contribution,  
        ChunkContext chunkContext)  
  
    throws Exception {  
  
        LOGGER.info(<<**job`step`>>);  
  
        return RepeatStatus.FINISHED;  
  
    }  
  
    }).build();  
  
}  
  
@Bean  
  
public Job job1() {
```

```
.start(step1())  
  
.next(step2())  
  
.next(step3())  
  
.build();  
  
}  
  
}
```

Now, we can run this Spring Batch job. It will log all job-related information on the console log. It runs after running Spring Cloud Task and terminates the application after finishing it.

Introducing to Spring Cloud Data Flow (SCDF)

Spring Cloud Data Flow is an orchestration layer on top of Spring Cloud Task and Spring Cloud Stream for batch processing. It visualizes all the tasks and batches. These are some of the features of SCDF:

It's a microservice-based streaming and batch processing Spring project. It can be deployed on the Kubernetes container environment and other fully managed services like Cloud foundry, and so on.

It provides an API for defining Spring Cloud Tasks and microservices using registered applications as named resources, and a web management console dashboard.

It provides tools and APIs to create complex data pipelines and give integration with tons of popular databases, messaging brokers, and so on. Actually, data pipelines consist of Spring Boot apps internally, which can be orchestrated and executed by SCDF triggers.

It also provides a simple API, which can be used from apps, tools, and command lines.

SCDF objectives

At a high level, the following are the major SCDF objectives:

Build data streaming

Support event-driven microservices

Build **Extract, transform, Load** data pipelines

Build short-lived batch jobs

Analytics, and reporting

The recently launched brand new Spring Cloud Data Flow microsite: is the best place to get started with SCDF.

SCDF architecture

The following is an SCDF server architecture:

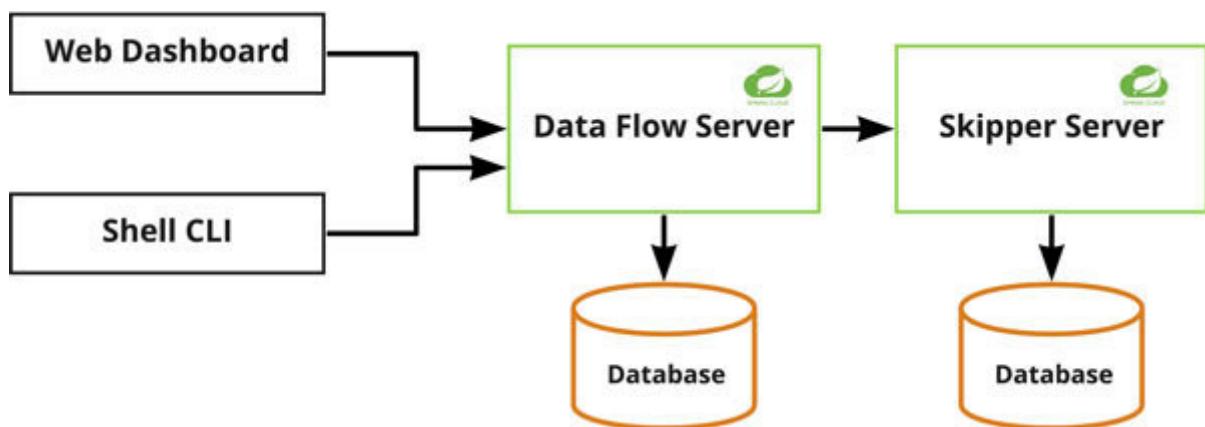


Figure 5.4: Spring Cloud Data Flow architecture

SCDF modules

These are some important SCDF modules, which are building blocks of SCDF architecture:

Data flow server

Skipper server

Database

Security

Stream processing

Batch processing

Monitoring

CLI and API library

Data flow server

This is the main module that handles all the API requests. It also has a web module, which provides a web dashboard to control, manage and orchestrate batch, cloud tasks, stream jobs, and so on. Web dashboard is controlled by a data flow server using web API calls. It helps to create data pipelines, orchestration of Spring Batch or Spring Cloud Task. It also maintains batch job execution order, and so on.

It validates and persists streams, task, batch job definitions and deployments. It handles the delegation of job scheduling, querying detailed tasks, and job execution history.

It uses **Domain Specific Language** for parsing streams and batch jobs definitions. It persists batch jobs metadata into the database to restart batch jobs when it has failed. It helps with data loss issues during batch job's abrupt termination.

It also manages ordering and parallel batch job executions.

Skipper server

This server is responsible for deploying and managing streams on different platforms. It also upgrades and roll-back streams using a blue-green deployment model and stores meta-data of streams. A **Skipper** is a tool that allows you to discover Spring Boot applications and manage their lifecycle on multiple cloud platforms.

Skipper standalone can be integrated with **Continuous Integration** pipelines to help achieve **Continuous Deployment** of applications.

Database

SCDF persists metadata on a SQL database. By default, it stores in an embedded H2 in-memory database. We can configure any SQL database like MySQL, PostgreSQL, Oracle, and so on.

Security

SCDF executable JARS uses **OAuth 2.0 authentication** to secure REST endpoints. Auth token can be used for basic authentication or OAuth 2. It's supported by Cloud Foundry with an in-built LDAP server.

Stream processing

It's the processing of an unbounded amount of data without interaction or interruption like continuous jogging step counts, health checks, and so on.

Batch processing

It's the processing of a finite amount of data without interaction or interruption. Applications that implement batch processing are referred to as ephemeral (short-lived) apps.

We will cover batch microservices in detail in the later section of this chapter.

Monitoring

SCDF provides monitoring capabilities of metrics architecture using Spring Boot **Micrometer**, **Wavefront**, and You can declaratively select monitoring system to use.

SCDF provides out of the box Grafana web dashboard to monitor Spring Cloud Task, jobs, and streams, and so on.

[CLI and API library](#)

SCDF provides CLI shell commands which are very handy to run on a terminal and execute all REST APIs. For example, you can also schedule and orchestrate jobs using the command line.

[Installing SCDF](#)

There are multiple platforms to install SCDF:

Local machine using Docker compose

Cloud Foundry

Kubernetes

Follow this SCDF official installation document for all kinds of platforms:

Getting started with SCDF

Let's do a hand-on example of SCDF.

In this section, we will register the same Spring Cloud Task app **spring-cloud-task-demo** with SCDF and launch the job from the SCDF web dashboard. SCDF provides API and **Data Flow Shell** CLI commands to manage Cloud Task and Spring Batch job operations and orchestrations. SCDF intuitive dashboard provides an easy onboarding platform to cloud tasks, batch, and stream monitoring, and orchestration.

Prerequisites

The following are the minimum specifications required for this sample app:

Installing SCDF

Same Spring Cloud Task demo app

Docker CLI

Let's get hands-on with SCDF with a sample task:

Install and configure SCDF manually on your machine or using Docker images. There are a few options to install and configure. We can choose based on our use case. Refer to this getting started page for the latest release:

<https://dataflow.spring.io/getting-started/>

Run SCDF on your web browser by getting external IP on Kubernetes cluster: < **external** If you are using local, then hit this URL:

Register Spring Cloud Task application by selecting the **Add Application(s)** menu from the left-side **Apps** menu. Refer to this

SCDF application registration screenshot:

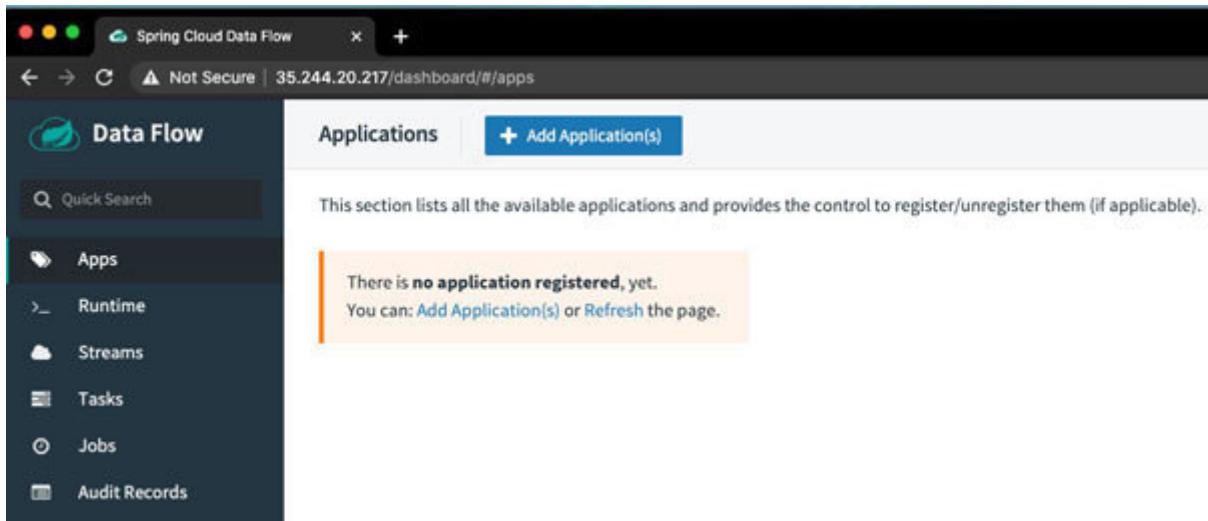


Figure 5.5: SCDF application registration page

We need to first create a Docker image of this demo Spring Boot app which is for Spring Cloud Task. We have already added Docker Maven's plugin and **Dockerfile** in the source code repo. Just run this command on the command line:

```
$maven clean install dockerfile:build
```

Maven command will create a Docker image which you can see by running the following Docker command:

```
$docker images
```

You can register app on local, Cloud Foundry, and Kubernetes. We are using the Kubernetes platform. On this platform batch

microservices can be registered by providing Docker image-related details, and so on. Please refer to this SCDF screenshot:

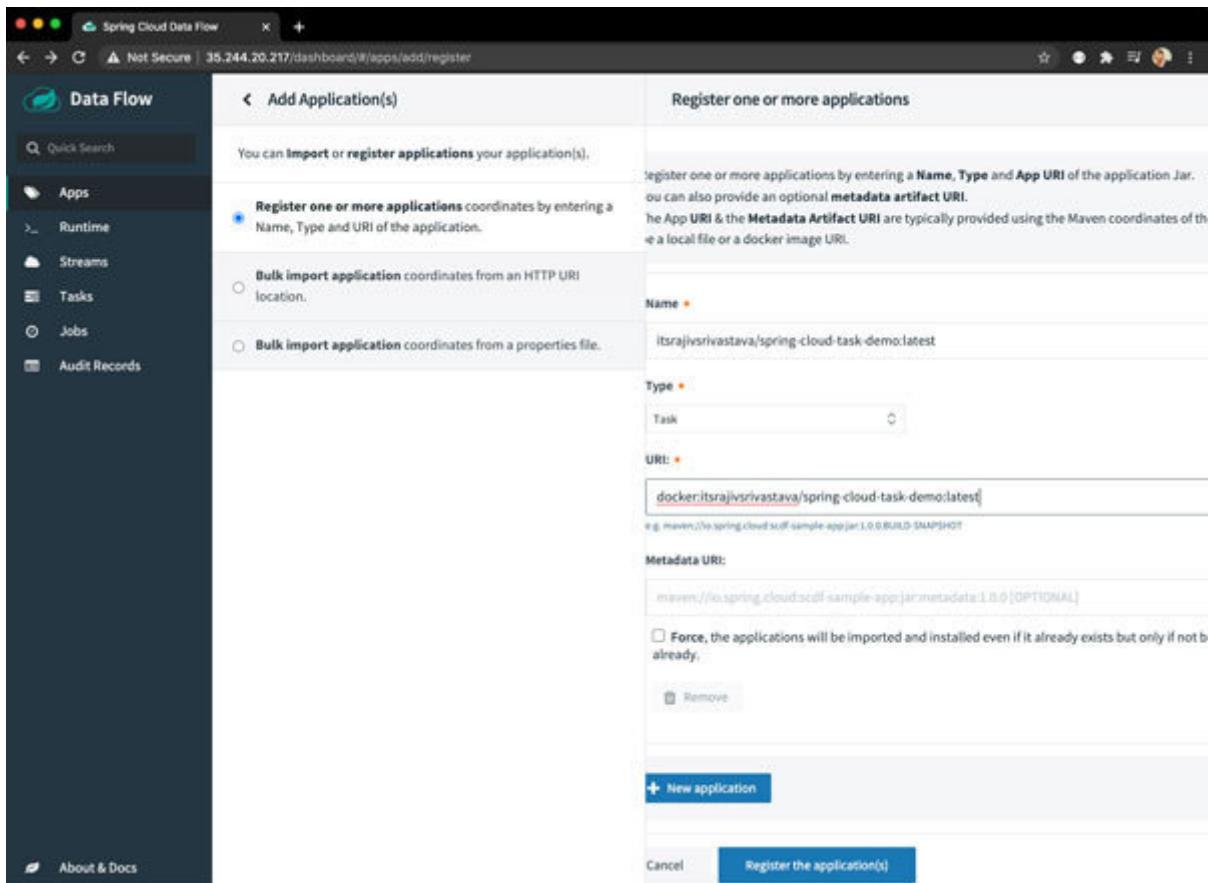


Figure 5.6: SCDF register app page

Now, we will create a task definition in SCDF. Select **Tasks** from the left navigation bar of SCDF and select **Create** SCDF provides a graphical editor that we can use to compose tasks. It provides **START** and **END** nodes in between. We can drag our newly registered app the **Applications** section between these two start and end nodes:

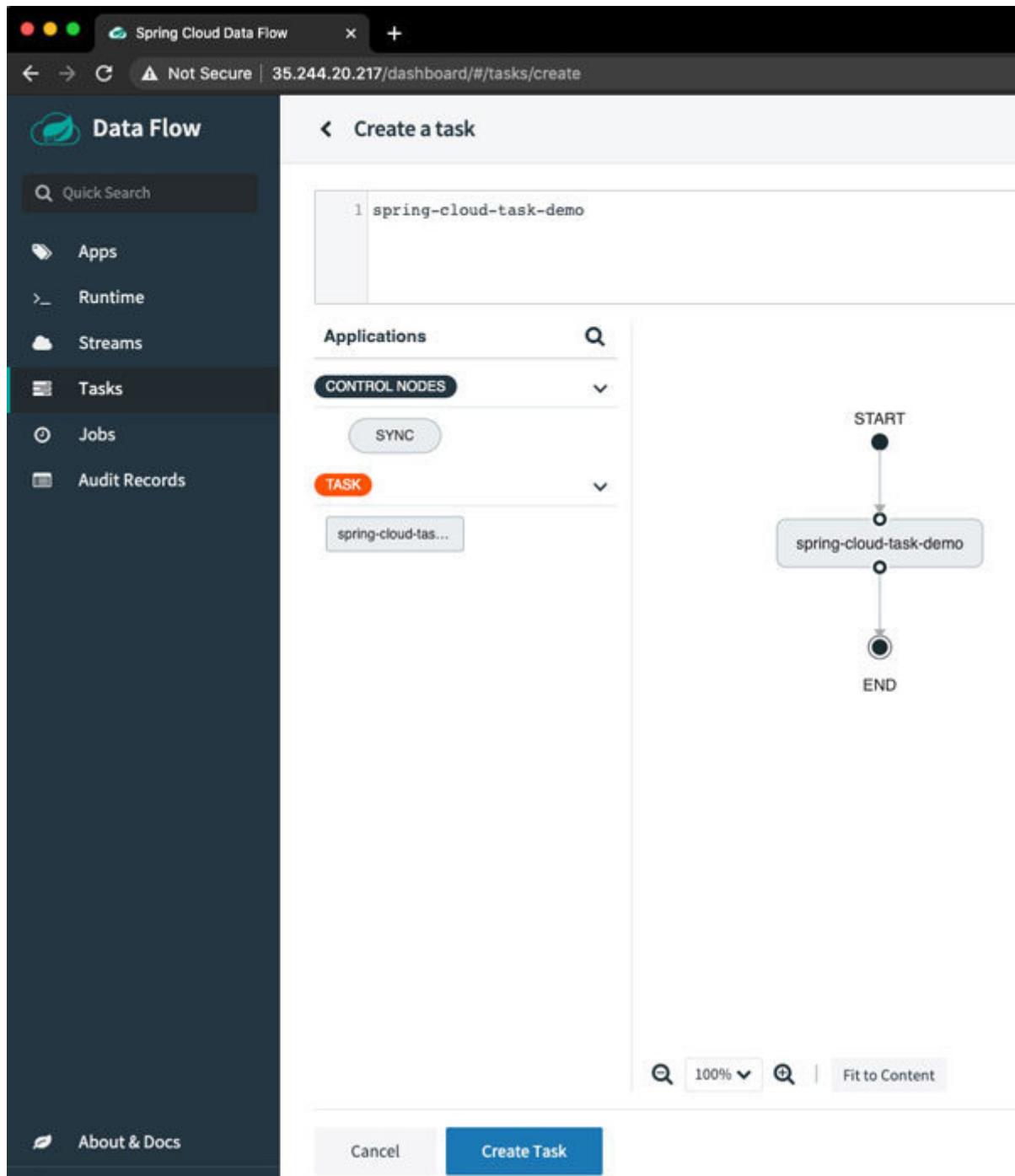
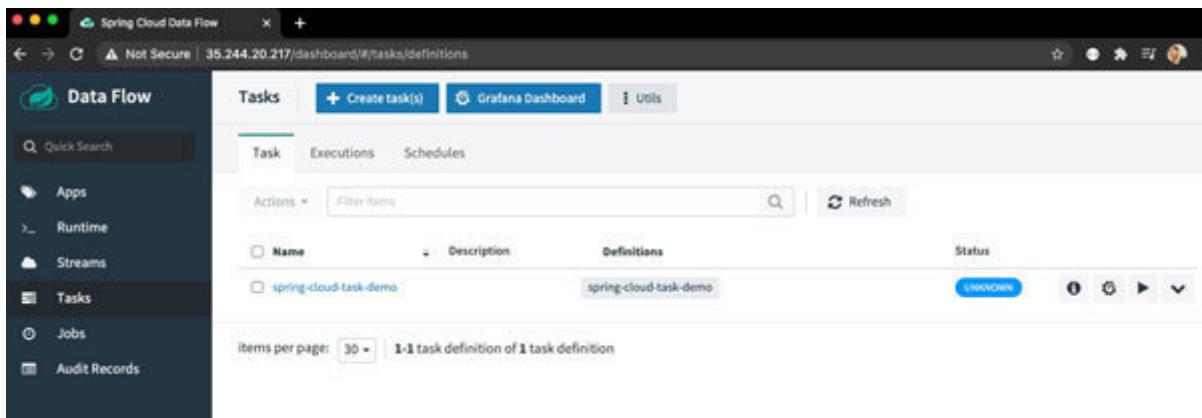


Figure 5.7: SCDF task creation page

In this next step, select **Create task(s)** and provide the task name. It will be a logical name, which will be used at all places for Cloud Task, batch, and stream job orchestrations.

Now, a task is created and ready to be launched. To launch the task, just click on the *play* button. In this step, we can provide command-line arguments and deployment parameters. Refer to the following SCDF screenshot for more understanding:

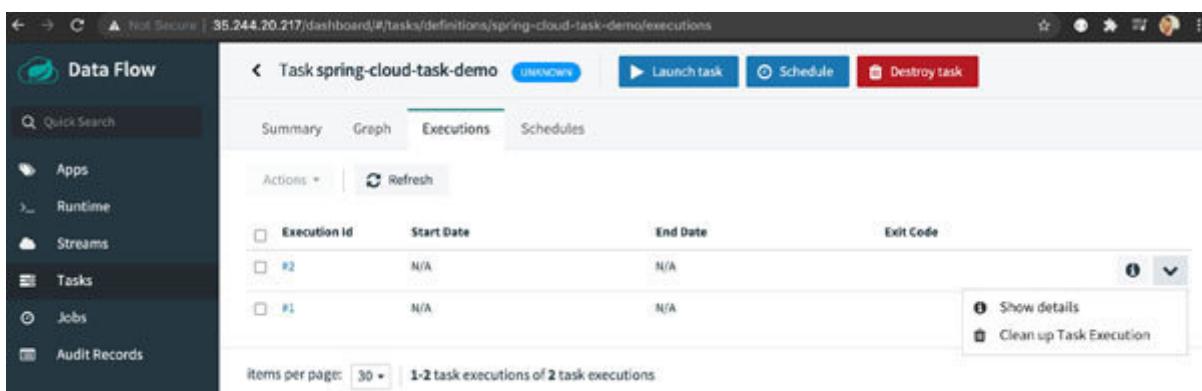


The screenshot shows the Spring Cloud Data Flow (SCDF) interface. On the left, there's a sidebar with options: Apps, Runtime, Streams, Tasks (which is selected), Jobs, and Audit Records. The main area has tabs for Tasks, Executions, and Schedules. Under Tasks, there's a table with one row. The table columns are Name, Description, Definitions, and Status. The Name is 'spring-cloud-task-demo', the Description is 'spring-cloud-task-demo', the Definitions is 'spring-cloud-task-demo', and the Status is 'UNKNOWN'. There are also some icons for actions like edit and delete.

Figure 5.8: SCDF run task launch page

Select the **Launch task** option. It will execute the task on the SCDF platform and records a new task execution. When the execution is complete, the **Status** field turns to green color and shows **Complete** status.

In the final stage, click on the **Executions** tab to view a summary of executions for this task like the following screenshot:



This screenshot shows the 'Executions' tab for the 'spring-cloud-task-demo' task. At the top, there are buttons for Launch task, Schedule, and Destroy task. Below that is a table with two rows of data. The columns are Execution ID, Start Date, End Date, and Exit Code. The first row has Execution ID '#2', Start Date 'N/A', End Date 'N/A', and Exit Code 'N/A'. The second row has Execution ID '#1', Start Date 'N/A', End Date 'N/A', and Exit Code 'N/A'. A context menu is open over the second row, with options 'Show details' and 'Clean up Task Execution'. The sidebar on the left is identical to the previous screenshot.

Figure 5.9: SCDF execution summary page

Introduction to Spring Cloud Stream

Spring Cloud Stream is a framework, based on events for stream messages. In this framework, event-driven microservices are connected with a shared message broker like Kafka or RabbitMQ, and so on.

These are three components of Spring Cloud Stream:

Message producer that sends messages to a destination.

The combination of a source and a sink. A processor consumes messages from a destination and produces messages to send to another destination.

Message consumer that reads messages from a destination.

Applications of these three types are registered with data flow by using the source, processor, and sink to describe the type of the application being registered.

It provides messaging framework integration with **open-source software** and enterprise-managed message brokers. This framework also supports messaging broker objects like producers, consumers, topics, partitions, and consumer groups.

You can also run Spring Cloud Task from Spring Cloud Stream.

Comparing Spring Cloud Task and Spring Batch

These are a few major similarities and comparison of Spring Cloud Task and Spring Batch:

Spring Cloud Task is complimentary to Spring Batch.

Spring Batch can be exposed as a Cloud Task.

Spring Cloud Task makes it easy to run Java/Spring microservice applications that do not need the robustness of the Spring Batch APIs.

Spring Cloud Task has good integration with Spring Batch and SCDF.

SCDF provides the feature of batch orchestration, and UI dashboard to monitor Spring Cloud Task.

In nutshell, all Spring Batch services can be exposed/ registered as Spring Cloud Task to have better control, monitoring, and manageability. They can be better visualized and controlled using the SCDF admin console dashboard.

Best practices for Spring Batch

These are a few best practices of Spring Batch based on my experience and other industry practices:

Use an external file system (volume services) for the persistence of large files with Cloud **Application Services** due to the file system limitations.

Always use SCDF abstraction layer with UI dashboard to manage, orchestrate, and monitor Spring Batch applications.

Always use Spring Cloud Task with Spring Batch for additional batch functionality.

Always register and implement vanilla Spring Batch applications as Spring Cloud Task in SCDF.

Use Spring Cloud Task when you need to run a finite workload via a simple Java microservice.

For **High Availability** implement the best suited horizontal scaling technique from the top scaling techniques based on the use cases on containers (K8s).

For a large PROD system, use SCDF as an orchestration layer with Spring Cloud Task to manage a large number of batches for large data sets.

App data and batch repository should live in the same schema for transaction synchronization.

Spring Batch auto-scaling

In this section, we will cover a couple of auto-scaling techniques of the Spring batch with use cases:

Vertical Vertical scaling is easy and supported with no hassle. Hardware or Kubernetes container size can be increased any time based on the usage of CPU, compute speed and RAM memory for better performance and reliability. As you give the process more RAM, you can typically increase the chunk size which will typically increase overall throughput, but it doesn't happen automatically.

Horizontal There are popular techniques, watch the following YouTube video for detail:

<https://www.youtube.com/watch?v=J6IPlfm7N6w>

and refer to the following GitHub code:

<https://github.com/mminella/scaling-demos>

Multi-threaded Each transaction/chunk executed in its separate threads, the state of the thread is not persisted. This is just an additional option if you don't want non-restart ability.

Parallel Multiple independent steps run in parallel via threads.

Single JVM async item writer/item ItemProcessor calls are executed within a Java **Future**. The **AsyncItemWriter** unwraps the result of the future and passes it to a configured delegate to write.

Data is partitioned then assigned to n workers that are being executed either within the same JVM via threads or in external JVMs launched dynamically when using Spring Cloud Task's partition extensions.

Remote Mostly I/O bound, sometimes when you need more processing power beyond the single JVM. It sends actual data remotely, only useful when processing is the bottleneck. Durable middleware is required for this option.

Spring Batch orchestration and composition

SCDF doesn't watch the jobs. It just shares the same database as the batch job does, so you can view the results. Once a job is launched via SCDF, SCDF itself has no interaction with the job. We can compose and orchestrate jobs by drag and drop and set dependency between jobs, which jobs should run in parallel and which one in sequence. Execution orders can also be set for multiple jobs scheduling.

SCDF orchestrates and systematically composes Spring Batch to better manage the control job execution of batch microservices.

Active-active operation for high availability (HA) between data centers

There are two standard ways: to achieve active-active HA between two data centers or availability zones on the cloud:

Place a shared Spring Batch job repository between two active-active DCs/AZs. Parallel sync happens in the job repository database. App data and batch repo should be in the same schema for better synchronization. Transaction isolation levels are set by default so that one of active DC can run the job and another job should be failed when it tries to re-run the same job with the same parameter. The following diagram is depicting parallel sync of batch jobs by sharing common Spring Batch job repository:

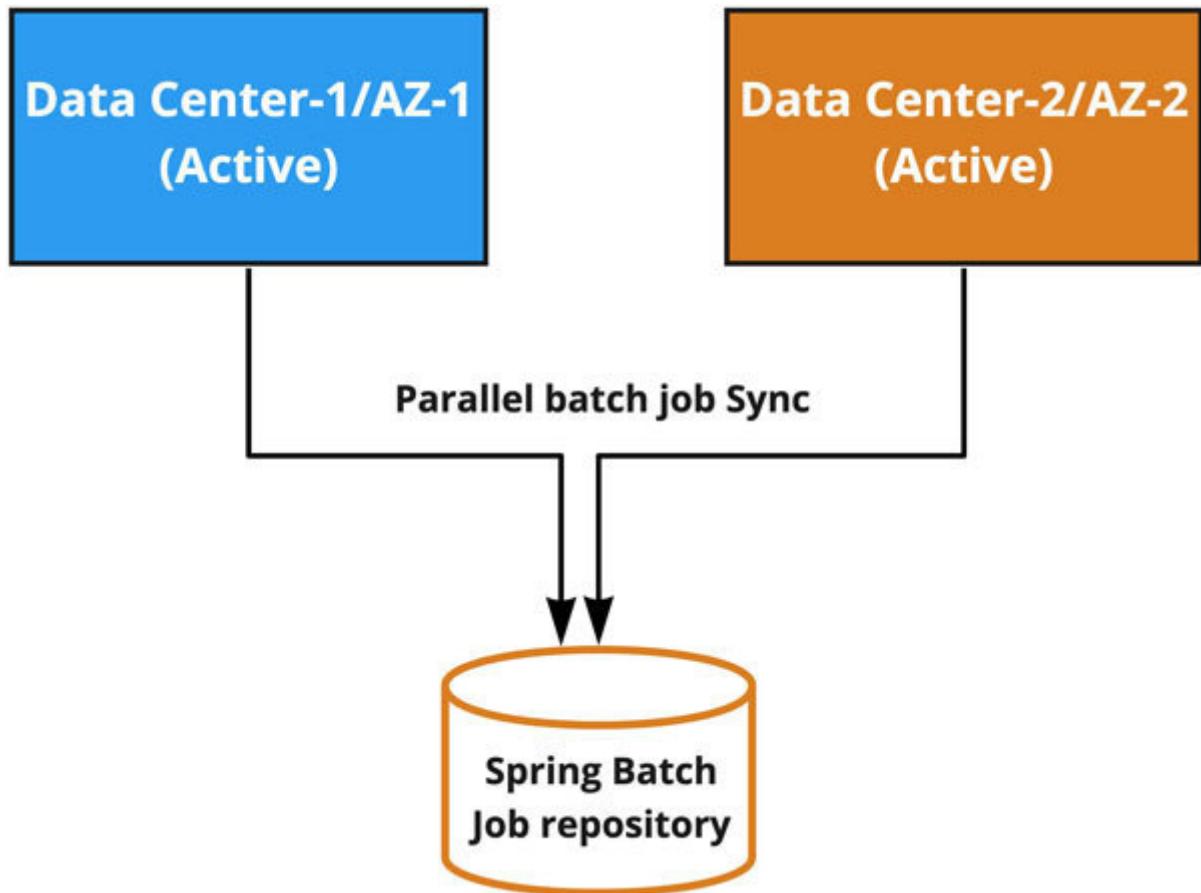


Figure 5.10: Spring Batch job repository parallel sync between DCs/AZs

Spring Cloud Task has this built-in functionality to restrict Spring Cloud Task instances:

<https://docs.spring.io/spring-cloud-task/docs/2.2.3.RELEASE/reference/#features-single-instance-enabled>

Alerts and monitoring of Spring Cloud Task and Spring Batch

These are a few tools and techniques to manage alerting and monitoring of Spring Cloud Task and Spring Batch:

Spring Cloud Task includes Spring Micrometer health check and metrics API out of the box.

Plain Prometheus is not suitable for jobs, because it uses pull mechanism and it won't tell when the job has finished or has some issues. If you want to use **Prometheus** for application metrics with **Grafana** visualization then use Prometheus **Rsocket-proxy** API:

<https://github.com/micrometer-metrics/prometheus-rsocket-proxy>

Conclusion

In this chapter, we have covered the fundamentals of batch microservices, Spring Cloud Task, Spring Batch, Cloud Stream, and SCDF with sample batch microservices. We have covered SCDF architecture with its modules in detail. We have covered different Spring batch microservices. We have discussed Spring batch practices, batch auto-scaling technique, batch orchestration, and composition methods for sequential or parallel batch processing. We have also covered active-active batch operation techniques for high availability between data centers. Also, we have covered Spring Cloud Task and Spring Batch alerting and monitoring tools and techniques.

In the next chapter, we will discuss how to build reactive and event-driven microservices using Spring Cloud and Spring Boot frameworks. We will also cover an overview of non-blocking synchronous REST API, how to develop non-blocking synchronous with event-driven asynchronous REST APIs with sample code examples.

Points to Remember

A batch microservice is a type of short-lived process or one-time batch execution that launches as the result of some trigger like a timer, due to some action or executed on demand.

Spring Cloud Task is a generic framework to create short-lived services

Spring batch is a framework that supports on-demand and scheduled short-time lived tasks run by some trigger or action. It provides APIs to write batch jobs easily.

Spring Cloud Data Flow is an orchestration layer on top of Spring Cloud Task and Spring Cloud Stream for batch processing.

When you need to onboard any Spring Batch to SCDF, you need to add an additional **@EnableTask** annotation.

You can also run Spring Cloud Task from Spring Cloud Stream.

You can refer to Spring Cloud Task official document page for advanced level of configuration:

<https://docs.spring.io/spring-cloud-task/>

Refer Spring Initializr web portal for creating Spring Boot apps:

<https://start.spring.io/>

You can refer Spring Batch official document page for advance level of configuration:

<https://spring.io/projects/spring-batch>

The recently launched brand new Spring Cloud Data Flow microsite is the best place to get started with SCDF:

<https://dataflow.spring.io/getting-started/>

SCDF provides a Grafana web dashboard to monitor Spring Cloud Task, jobs and streams, and so on.

SCDF provides CLI shell commands which are very handy to run on a terminal and execute all REST APIs.

There are multiple platforms to install SCDF-local machine using Docker compose, Cloud Foundry, and Kubernetes.

Spring Batch services can be exposed/registered as Spring Cloud Task to have better control, monitoring, and manageability.

Key terms

Spring Cloud Data Flow.

Application Programming Interface.

Input/Output.

Domain-Specific Language.

Command Line Interface.

Extract, Transform, Load.

References

Batch

Running Spring Batch applications in Cloud

Batch processing with Spring Cloud Data Flow

Spring Cloud

CHAPTER 6

Build Reactive and Event Driven Microservices

Recently, we have been hearing about a new paradigm of coding that is gaining more popularity that is called reactive programming and event-driven microservices. These both programming models are dependent on asynchronous communications and gaining attention to solve most of the business use cases. API based synchronous communication solves most of the CRUD use cases, however, it doesn't suitable for high load or high concurrent use cases. In those situations, asynchronous-based reactive and event-based programming works well to handle the high concurrent load with great performance on these similar use cases.

This chapter will cover the introduction of reactive microservices, non-blocking synchronous APIs, and event-driven asynchronous microservices. We will discuss reactive design patterns. We will also cover how to develop sample reactive with Spring's project **Reactor** with **Spring WebFlux** and event-driven microservices using **Spring Cloud Spring** and We will cover event-driven asynchronous services.

Let's get started!

Structure

In this chapter we will cover the following topics:

Reactive programming overview

Advantages of a reactive programming model

Spring WebFlux and Spring MVC comparison

Non-blocking synchronous API

Challenges with traditional REST API

Reactive Relational Database Connectivity

Developing reactive microservices using Spring Webflux

Event-driven asynchronous microservices

Developing event-driven asynchronous microservices using Spring Cloud Stream and Kafka

Creating producer microservice

Creating consumer microservice

Test event-driven microservice

Objective

After studying this chapter, you should be able to understand reactive design pattern architecture, comparison, and similarities of Spring Webflux and Spring MVC. This chapter will help you to understand non-blocking synchronous API with use cases, traditional REST API challenges, advantages of the reactive programming model, and its usage. It will help you to learn about R2DBC and how to develop reactive microservices using Spring Webflux. This chapter will help you understand event-driven asynchronous microservices and how to implement it using Spring Cloud Stream and Kafka with a real use case.

Reactive programming overview

Reactive programming is a non-blocking asynchronous data flow. It uses building blocks of functional programming based on **Function as a Service** while event-driven programming is process-oriented. It's also called **Functional Reactive Programming**. Reactive programming solves two important things: **performance** and **memory**. It was born as a result of modern apps needing a way of coding that provided fast, user-friendly results.

It gives better performance because client requests won't be blocked. Same system CPU, memory, and I/O resources are shared for multiple threads to avoid wastage of these resources. In this approach, CPU and memory resources won't be blocked by a single request. A **reactive service** can take the higher load because it can switch contexts between different threads and provide non-blocking asynchronous services.

“The reactive design pattern is an event-based architecture for asynchronous handling of a large volume of concurrent service requests coming from single or multiple service handlers” — Baeldung.

Just imagine a real-life analogy to understand reactive programming use cases. When you drive through a restaurant where you place an order at the beginning, then you pay at the

next counter and take delivery at the last counter. When you place your order at the first counter and move your car ahead, another person in the queue will get a chance and place their order, during this time your order is already in process. In this way, there is no blocking of other queued people if the order is getting ready. Reactive programming provides this feature to take multiple requests from different clients/customers and process them in asynchronous ways. It serves concurrent requests in a high load distributed system.

There are a few different reactive programming projects:

Project Reactor

RxJava

Reactive streams

Kotlin flow

Kotlin coroutines are more familiar and popular than Kotlin flow

Loom and **Fiber projects** will add reactive support as a core part of the JDK

In this book, we will cover the popular Spring Boot reactor with Spring WebFlux. We need this WebFlux library to make our services fully reactive. It's a Spring framework project which helps to write reactive non-blocking programs. We can create an async

REST API. **WebFlux** is a framework for reactive web programming base on Project Reactor.

Spring WebFlux supports reactive HTTP and WebSocket clients as well as reactive server web applications, including REST, HTML browser, and WebSocket style interactions. WebFlux allows for more scalability, a stack immune to latency (useful for micro services-oriented architecture), and better stream processing capabilities.

We can easily have an application with the Spring MVC model for one controller and the WebFlux model for another controller.

Advantages of a reactive programming model

The advantages of reactive programming models are as follows:

Faster performance because it's asynchronous and non-blocking.

Better utilization of CPU, memory, and I/O resources.

Handle backpressure efficiently. It means resistance or force opposing the desired flow of data through software. Backpressure is when the progress of turning that input to output is resisted in some way. In most cases that resistance is computational speed or if the software has to wait for the user to take some action.

Follows event-driven model. It's an important aspect which is faced by developers.

A single thread can handle multiple client requests one by one and use a call-back mechanism to inform client requests.

Handle concurrent resources when client requests load/TPS are very high.

It auto-scales based on client requests load.

It provides predictable API response time.

Simpler than regular multi-threading.

Avoid calls back.

In-built mechanism for error recovery.

Easy interactions with UI thread and events.

Make hassle-free concurrency.

Project Reactor is a VMware project for reactive asynchronous programming. One of the implementations of the reactive stream's specification. The Reactor is fully non-blocking and provides efficient demand management.

The Reactor offers two reactive and composable APIs:

Mono The Mono API allows producing only one value. It is a Reactive Streams publisher with basic operators that completes successfully by emitting **0** to **1** element, or with an error.

Flux The Flux can be endless; it can produce multiple values. It extensively implements reactive extensions. Reactor offers non-blocking, backpressure-ready network engines for HTTP (including WebSockets), TCP, and UDP. It is well-suited for microservices architecture.

Spring WebFlux and Spring MVC comparison

Spring MVC helps to create synchronous REST API which is blocking because API will be dependent on the backend services, it will complete one loop when it gets a response from the backend. Both have some similarities too, like they both have **@Controller** based API endpoints, supports reactive clients, and run on and **Undertow** app servers.

We can't apply reactive design patterns for all the use cases. It works for certain use cases where TPS load is high, network latency is low and the cost is high. Similarly, we should not migrate the existing REST API to non-blocking without considering use cases. For example, file upload to any social portal with a non-blocking design will be suitable.

The following screenshot is depicting a comparison of **Spring MVC** and **Spring**

Spring MVC vs Spring Reactive

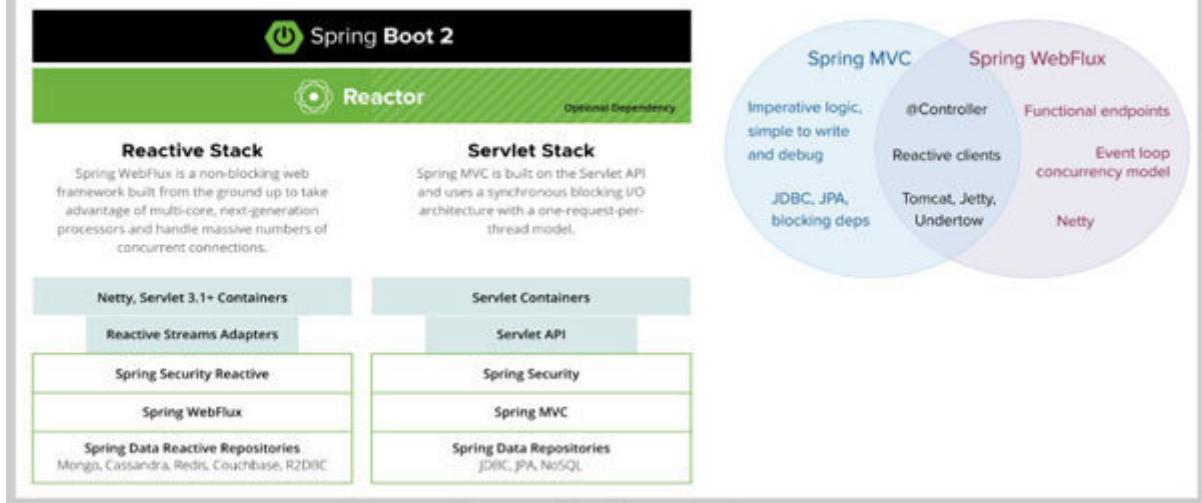


Image Credit: spring.io

Figure 6.1: Spring MVC versus Spring Reactive

Non-blocking synchronous API

Traditional blocking APIs performances are slow in some of the use cases when client request load is very high and backend services/databases response slow. In this scenario, other client requests block because they wait to complete the initial client request processing thread by backend services. The thread that has initiated the operation is blocked and the CPU is waiting idle for the I/O operation to complete. It makes the overall API response slow and sometimes causes application crashes. Blocking API also causes additional operational cost, business loss of losing customer's interest, and obviously these APIs are not resilient. Non-blocking asynchronous API helps to overcome these practical challenges.

Traditional blocking API uses a thread pool to serve a limited number of threads. These threads are being used by different client requests in parallel, however, it can be exhausted soon if these threads will be blocked for the same resource.

The following diagram is depicting traditional REST API request-response communications:

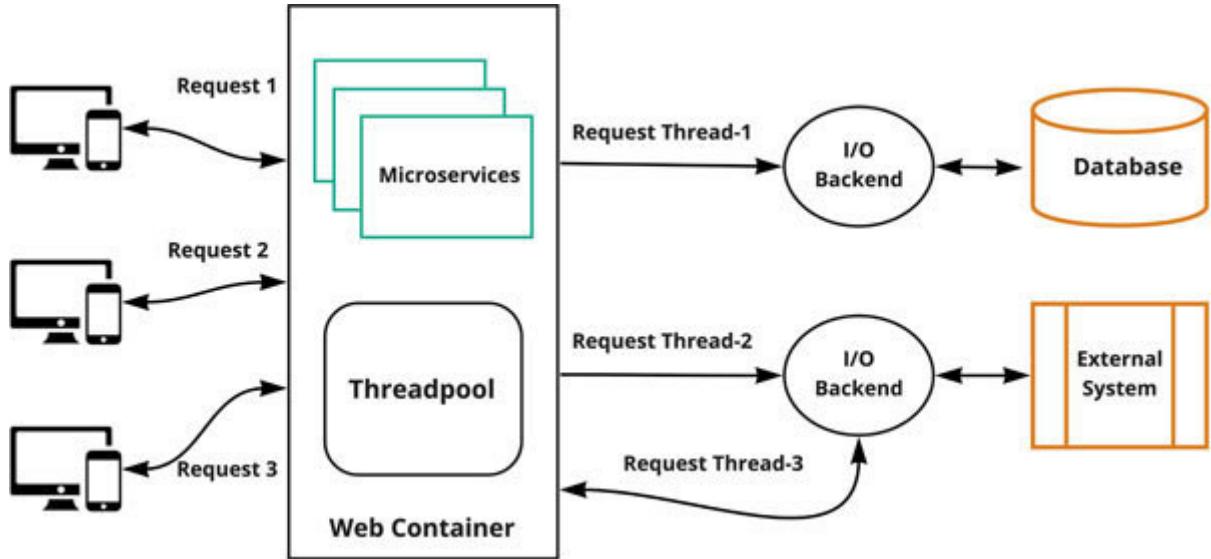


Figure 6.2: Traditional REST API request-response communication

This is a new programming paradigm to write and expose non-blocking REST API. It makes applications faster, resilient, and cost-effective. Non-blocking API can be written using Spring's Reactor project.

Non-blocking API operation initiates the process by sending instructions to the CPU for I/O operations and returning immediately to process other client requests because the thread is available to handle other requests. When the submitted request process is completed, it uses a call-back mechanism and returns the API response from the backend services. It solves a major blocking issue, which we have discussed that slows performance of blocking synchronous API. Its working is a little complex. I hope I have explained this concept in simple words.

In this non-blocking design, the CPU is not waiting for I/O operation. All those precious CPU processing cycles wasted in

waiting for blocking I/O to complete can be used to perform other business operations by handling other threads. As a result, we can run more business logic on the same number of CPU cores, and our system becomes faster, resilient, and cost-effective to run.

The following diagram is depicting the reactive programming model:

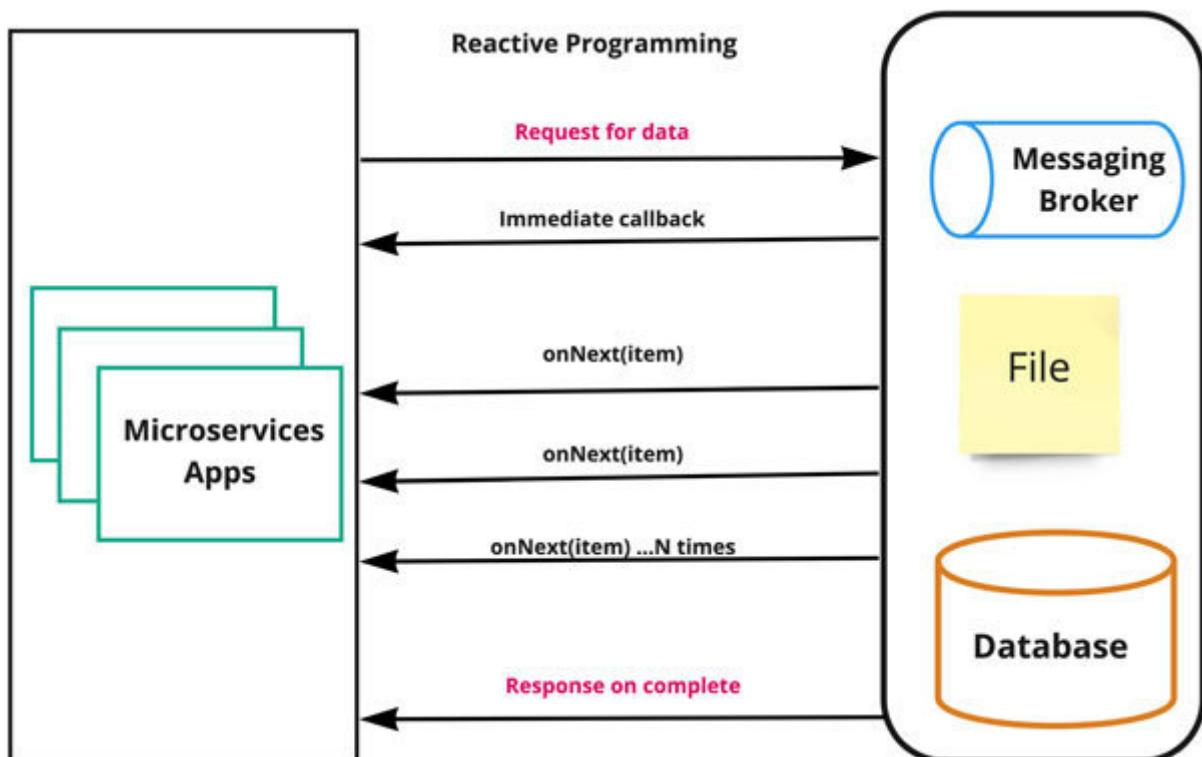


Figure 6.3: Reactive programming model

This is how presentation layer and DB layer interacts in reactive microservices systems:

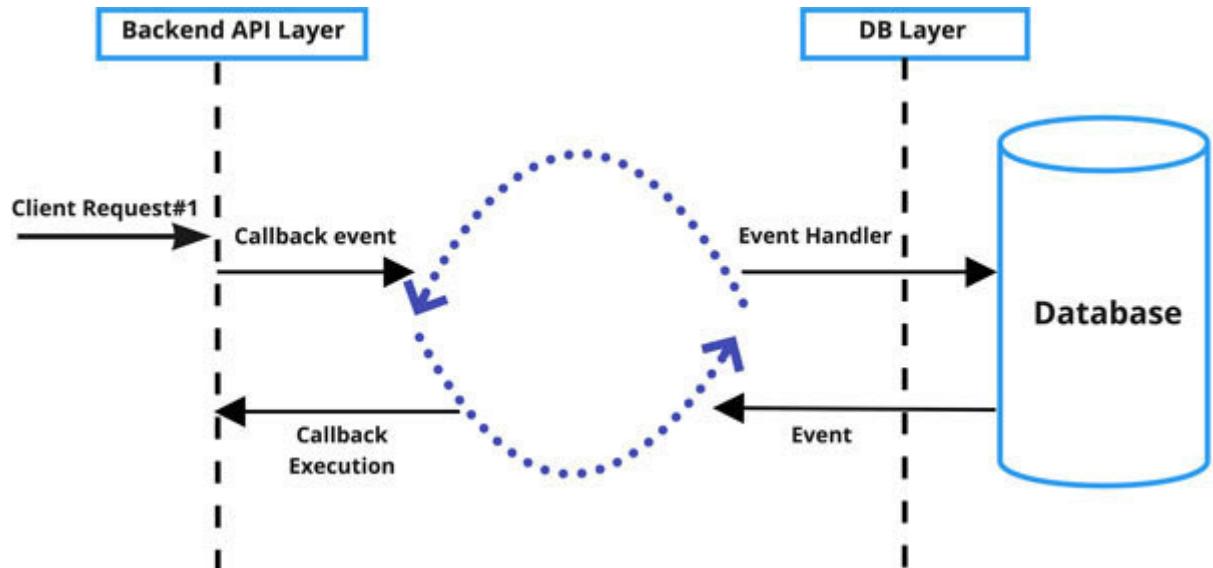


Figure 6.4: Reactive microservices ecosystem

Challenges with traditional REST API

The following are the main REST API challenges:

Slow performance due to synchronous and blocking threads.

Single thread per request model.

Slow performance limitations to handle high concurrent client requests, because many requests get piled up and make the system non-responsive.

CPU, memory, and I/O resources are not properly utilized and optimized.

No backpressure support, if backend services send more responses which can't be handled by the client request.

Reactive Relational Database Connectivity (R2DBC)

End-to-end reactive from application to backend databases are important to leverage 100% reactive advantages. Just imagine, we have developed an application using a non-blocking programming model, however, the underlying database works on a blocking programming model. It creates again the same performance and other issues of traditional non-reactive programming. So, it's important that reactive non-blocking applications should be integrated with non-blocking databases.

Reactive non-blocking applications should be integrated with non-blocking database resources for better performance. It's a very important factor to consider to achieve 100% reactive programming benefits.

R2DBC is a reactive driver which helps to integrate async Spring Boot microservices with SQL databases. It provides repository support for R2DBC and helps Spring Boot applications to perform database operations in a reactive way. At this moment, it supports PostgreSQL, MySQL, and embedded H2 databases.

Developing reactive microservices using Spring Webflux

Let's do some hands-on lab! We will create a reactive microservice with a couple of REST APIs. In this coding exercise, we will create **Order-service** and expose as reactive microservices using Mono and Flux.

Spring Webflux's Mono class is for getting single record and Flux for more than one records.

Prerequisite

These are basic installation requirements to build a reactive microservice:

Java 8+

Spring Boot v2.2.3+

Spring WebFlux

PostgreSQL R2DBC driver

PostgreSQL database

Source code reference: <https://github.com/rajivmca2004/reactive-postgres-spring>

Let's create a simple Java app using Spring Boot, Spring WebFlux, and PostgreSQL database:

We will follow the same approach, by creating this **reactive-postgres-spring** project on the spring Initializr web portal:

<https://start.spring.io/>

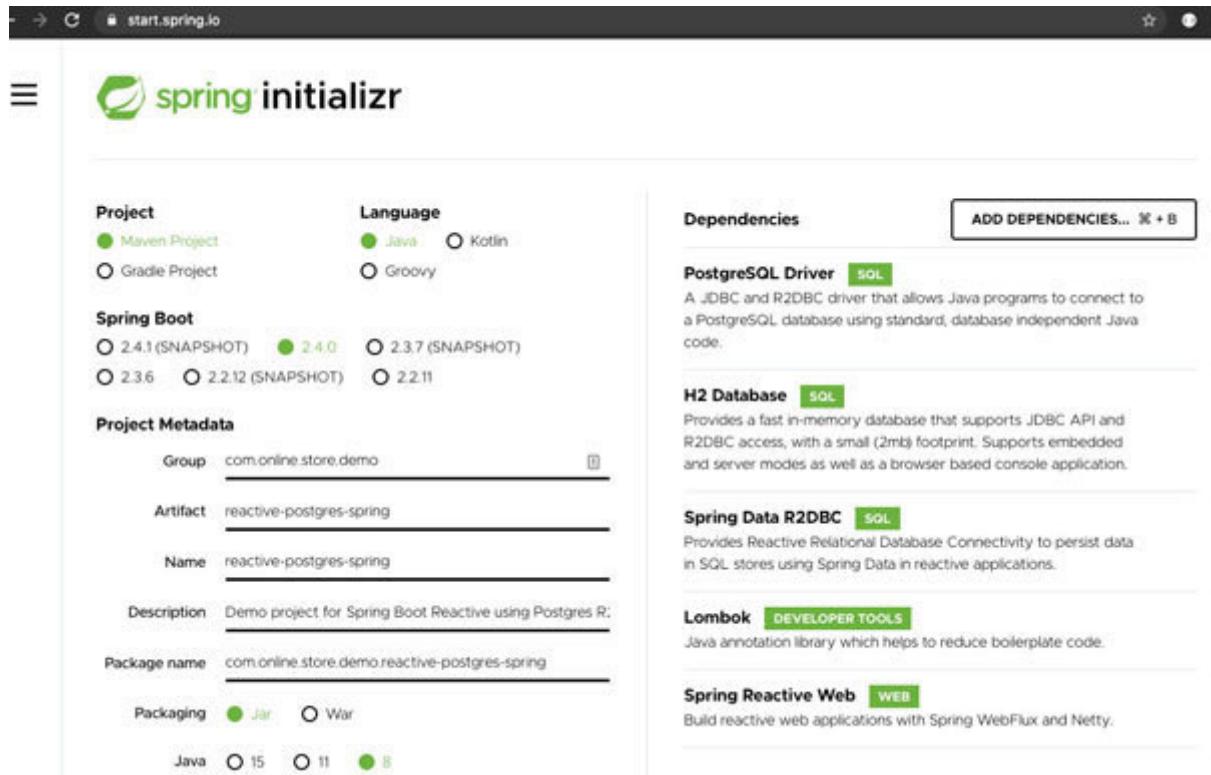


Figure 6.5: Spring Initializr reactive-postgreSQL-spring project

We need to add these Maven dependencies in the **pom.xml** file. Also, we need to add **Spring Data PostgreSQL** and **Spring Reactive**

It's a Spring Boot starter project and on top of that, we will add **Spring WebFlux** and other related binaries dependencies. We can check the latest version on Maven central portal here:

<https://mvnrepository.com/>

We also need to add **Springfox** and **Swagger** related dependencies.
It will add these dependencies:

We will create two **dev** and **test** profiles for this microservice app.
We will use PostgreSQL database for **dev** and embedded H2
database for **test** profile for local testing:

spring:

profiles: dev

r\`dbc:

url:

username: postgres

```
password: password
```

```
logging:
```

```
level:
```

```
    org.springframework.data.r`dbc: Debug
```

```
---
```

```
spring:
```

```
profiles: test
```

```
r`dbc:
```

```
url:
```

```
name: sa
```

```
password:
```

Instead of an H₂ in-memory database, we highly prefer to use **@testcontainers** API to test. **Testcontainers** is a Java library that supports JUnit tests, providing lightweight, throwaway instances of common databases, Selenium web browsers, or anything else that

can run in a Docker container. Refer this official document for more details:

<https://www.testcontainers.org/>

We need to add this SQL script to create table This script will create an **orders** table in the PostgreSQL database:

```
IF EXISTS orders ;
```

```
orders (id INT GENERATED IDENTITY p_name qty
```

Add a Spring bean of the **ConnectionFactoryInitializer** class to create a database connection with the PostgreSQL database. Create database schema by reading **order-schema.sql** file from the class loader.

We can also use the **FlyBase** database for the same. Refer to official documents for more details:

<https://github.com/flybase>

```
@Bean
```

```
public ConnectionFactoryInitializer initializer (ConnectionFactory  
connectionFactory) {
```

```
    ConnectionFactoryInitializer initializer = new  
ConnectionFactoryInitializer();  
  
    initializer.setConnectionFactory(connectionFactory);  
  
    CompositeDatabasePopulator populator = new  
CompositeDatabasePopulator();  
  
    initializer.setDatabasePopulator(populator);  
  
    return initializer;  
}
```

Create a REST controller with these API endpoints for CRUD operations. In this sample code exercise, we are going to expose these REST APIs:

```
@Autowired  
  
private OrderService orderService;  
  
/*  
 * Update order for single (Mono) order Id=> POST:  
 http://localhost:8080/orders
```

```
 */
```

```
@PostMapping
```

```
@ResponseStatus(HttpStatus.CREATED)
```

```
public Mono<Order> createOrder(@RequestBody Order order) {
```

```
    return orderService.createOrder(order);
```

```
}
```

```
/*
```

```
 * Update order by Id for single (Mono) PUT:
```

```
http://localhost:8080/orders/{orderId}
```

```
 */
```

```
public Mono<Order> updateOrder(@PathVariable Integer orderId,  
@RequestBody Order order) {
```

```
    return orderService.updateOrder(orderId, order).map(updatedOrder ->  
        ResponseEntity.ok(updatedOrder))
```

```
        .defaultIfEmpty(ResponseEntity.badRequest().build())
    );
```

```
}
```

```
/*
```

```
 * Delete order by Id for single (Mono) DELETE:  
http://localhost:8080/orders/{orderId}
```

```
*/
```

```
public Mono< ResponseEntity > deleteOrder(@PathVariable Integer orderId) {
```

```
    return orderService.deleteOrder(orderId).map(r ->  
        ResponseEntity.ok().build())
```

```
        .defaultIfEmpty(ResponseEntity.notFound().build());
```

```
}
```

```
/*
```

```
 * Get all orders (Flux - Many) list => GET:  
http://localhost:8080/orders
```

```
*/
```

```
@GetMapping
```

```
public Flux getAllOrders() {
```

```
    return orderService.getAllOrders();
```

```
}
```

```
/*
```

```
 * Get order by order id for single (Mono) => GET:  
http://localhost:8080/orders/{orderId}
```

```
*/
```

```
public Mono<Order> getOrderById(@PathVariable Integer orderId) {
```

```
    Mono<Order> order = orderService.findById(orderId);
```

```
    return order.map(u ->  
        ResponseEntity.ok(u).defaultIfEmpty(ResponseEntity.notFound().build());  
    );
```

```
}
```

```
/*
```

```
 * Get orders (Flux) by quantity => GET:  
http://localhost:^·^·/orders/q{ty}/{qty}
```

```
*/
```

```
public Flux getUsersByQuantity(@PathVariable int qty) {
```

```
    return orderService.findOrdersByQuantity(qty);
```

```
}
```

```
/*
```

```
 * Search orders (Flux) by order ids => POST:  
http://localhost:^·^·/orders//search{id}
```

```
*/
```

```
public Flux fetchUsersByIds(@RequestBody List orderIds) {
```

```
    return orderService.fetchOrders(orderIds);
```

```
}
```

Create a **OrderRepository** object repository by extending **ReactiveCrud Repository** class. This Spring DAO class provides standard CRUD methods. In this interface, we are only creating a custom query method for searching orders from the database by using quantity number

```
OrderRepository extends ReactiveCrudRepository<Integer> {  
  
    * from orders where qty >=  
  
    Flux<Integer> findOrdersByQty(@Param("qty") int qty);  
  
}
```

Then create an **Order** model bean for holding DAO objects and map them with database column names. Please refer to the reference source code from the GitHub repository.

Create **OrderService** class to execute business logic and call DAO methods. Please refer to the reference source code from GitHub repository.

Insert a couple of records to the PostgreSQL database in the **Order** table using PostgreSQL's **pgAdmin** web tool:

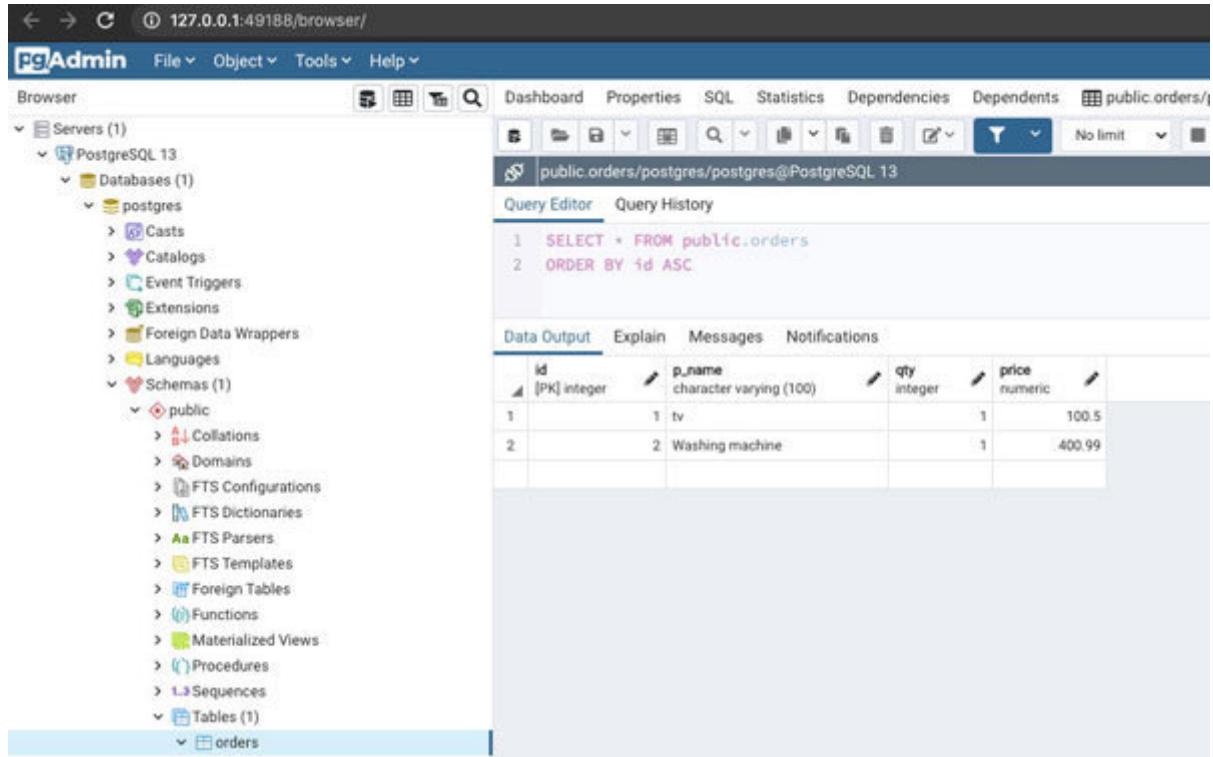


Figure 6.6: PostgreSQL PgAdmin web query tool

It's time to test these APIs using the Chrome browser **Postman** client. Hit **/orders** REST API which will get a response for all **orders** details:

The screenshot shows a REST API client interface. At the top, there is a header bar with 'GET' selected, a URL field containing 'http://localhost:8080/orders', and a 'Send' button. Below the header are tabs for 'Params', 'Auth', 'Headers (6)', 'Body', 'Pre-req.', 'Tests', and 'Settings'. The 'Params' tab is active, showing a table titled 'Query Params' with one row: 'Key' and 'Value'. The 'Body' tab is also visible. At the bottom, there is a status bar showing '200 OK 8 ms 184 B' and a 'Save' button. The main content area displays the JSON response:

```
1 [  
2 {  
3   "id": 1,  
4   "p_name": "tv",  
5   "qty": 1,  
6   "price": 100.5  
7 },  
8 {  
9   "id": 2,  
10  "p_name": "Washing machine",  
11  "qty": 1,  
12  "price": 400.99  
13 }]  
14 ]
```

Figure 6.7: REST API response

Now, hit REST API `/orders/1` to get details of a specific order ID number:

The screenshot shows the Postman interface with a GET request to `http://localhost:8080/orders/1`. The response is a single JSON object:

```
1 {  
2   "id": 1,  
3   "p_name": "tv",  
4   "qty": 1,  
5   "price": 100.5  
6 }
```

Figure 6.8: Response of a single order ID

Hit `orders/qty/1` REST API to get details of those orders where quantity is

The screenshot shows the Postman interface with a GET request to `http://localhost:8080/orders/qty/1`. The response status is 200 OK, with a response time of 37 ms and a size of 184 B. The response body is a JSON array containing two order objects:

```
1 [  
2 {  
3   "id": 1,  
4   "p_name": "tv",  
5   "qty": 1,  
6   "price": 100.5  
7 },  
8 {  
9   "id": 2,  
10  "p_name": "Washing machine",  
11  "qty": 1,  
12  "price": 400.99  
13 }  
14 ]
```

Figure 6.9: Response of $qty > 1$ order

Hit `orders/search/id` REST API to get details of a given list of order numbers:

The screenshot shows the Postman interface with a POST request to `http://localhost:8080/orders/search/{id}`. The request body is set to `[1,2]` in JSON format. The response is a 200 OK status with a JSON payload containing two order items:

```
1: [
  2: {
    "id": 2,
    "p_name": "Laptop",
    "qty": 2,
    "price": 700.5
  },
  3: {
    "id": 3,
    "p_name": "tv",
    "qty": 1,
    "price": 100.5
  }
]
```

Figure 6.10: REST API query response for order ID 1 and 2

Event-driven asynchronous microservices

These services are based on event-driven microservices architecture. In this design, microservices interact with each other asynchronously. It's based on non-blocking communication between request and response, because a microservice doesn't need to wait for the response from dependent microservice. It makes services independent, faster, and easy to integrate.

It's based on messaging broker technology. In this design approach, one microservice work as a **producer service**, and other microservice works as a **consumer**. Producer microservice sends event messages to the messaging broker's topic, consumer reads that event message and takes the desired action. It gives near to real-time async communication.

It provides loosely coupled system. In this approach, instead of requesting data when needed, apps consume them via events (messages). Therefore, overall app performance increases. **Loose coupling** is one of the main key feature of a microservice environment, which makes easy integration and enhancements.

Event-driven programming solves **two-phase commit** challenges. Two-phase commit has two phases: **prepare** and **commit** phase. It's not recommended for microservices, because it's synchronous and thus blocking. It creates performance issues and app failures.

for the high speed of client requests. The event-driven **Saga pattern** is asynchronous and non-blocking. It solves 2PC issues. The Saga pattern is another widely used pattern for distributed transactions. The Saga pattern is asynchronous and reactive. In a Saga pattern, the distributed transaction is fulfilled by asynchronous local transactions on all related microservices. The microservices communicate with each other through an event hub].

There are many messaging brokers which can be used in this architecture like Kafka, RabbitMQ, Google Pub/Sub, AWS Kinesis, and so on.

Event-driven architecture is a software architecture paradigm. Event producers and event consumers are two major components of EDA. The producer senses any state change and sends this state as an event/message to the message broker. Transmission of events across multiple microservices happens via the event channel/messaging broker. There are two types of the EDA model:

Pub/Sub In this model when the event occurs or is produced, then the system will put this event as a message in the event stream, and the subscriber who is listening to this stream will consume this event.

Event In this model, events are written in logs and event consumers don't subscribe to an event stream. It can read events from any part of the stream at any time.

Using the EDA approach, developers can develop loosely coupled real-time and distributed systems. For example, when a customer places an order using an online eCommerce portal, it interacts with various other microservices asynchronously like product catalogue order, payment services and completes the order. Eventing can be used for IoT data streaming use cases, analytics, and externalizing log aggregation.

There are some challenges also for event-driven systems. Scalability and reliability are two major concerns, there we need to write producer/consumer code that will produce and consume events. Also, we need to write specific business logic to process these async event messages for a specific use case.

These are some forms of event-driven design patterns and their usages:

Event In this pattern, a system sends a message to another system and notify asynchronously. In this pattern, the source system doesn't bother or wait for a response from the target system. It works on fire and forget principle. The main disadvantage of this pattern is the communication gap between source and target systems. The source system won't know what to do next, because the source system won't get a response after sending the event message to the target system.

Event-carried state This pattern works when source systems update clients of a system in such a way that they don't need to contact the source system in order to do further work. For example, in

the eCommerce catalog management system, when any change happens at product data, it sends and notifies target systems to change their dependent data set. The disadvantage of this pattern is to maintain multiple copies of data at various source and target systems.

Event In this pattern, every state change in the system (through event-driven programming) is recorded at the persistent/storage layer and re-built entirely end-to-end transaction system flow from the persisted recorded data. The event store becomes the principal source of truth, and the system state is purely derived from it. A Real-life example is recording version control commits and restoring to any version during failure in code.

It's based on the **Command Query Responsibility Segregation** design pattern. It recommends having two separate data structures (model) for read and write operations. We can use CQRS without event-driven programming. It's not dependent on events. In complex use cases, the same data model doesn't fit for both write and read. Maybe, the client wants to add only a small portion of the data changes frequently. However, read returns more data based on the query. This can be handled by applying separate data models for separate read and write purposes.

Developing event-driven asynchronous microservices using Spring Cloud Stream and Kafka

Let's do some hands-on We will create an event-driven microservice. In this coding exercise, we will create two microservices that will communicate to each other asynchronously using an open-source **Kafka** messaging broker. We also need an open-source **Zookeeper** to manage and orchestrate Kafka clusters.

Also, we will use the Spring Cloud Stream framework to implement this event-driven eCommerce-based use case quickly and easily. Spring Cloud Stream helps to reduce boilerplate code and just focuses on the core business logic of our application. It has support for almost all popular messaging broker platforms. We just need to write some spring beans and define some cluster properties and It's also supporting multi-message binding by connecting message brokers together such as Kafka and RabbitMQ. It's also support public cloud vendor-specific messaging platform binding like AWS Kinesis, Google Pub/Sub, Azure Event Hubs, and so on.

Prerequisite

These are basic installation requirements to build a reactive microservice:

Java 8+.

Spring Boot v2.2.3+.

Spring Cloud Stream.

Spring Kafka template.

Kafka messaging broker. The installation steps can be found here:

<https://kafka.apache.org/quickstart>

Zookeeper library that comes with the same Kafka installation preceding step.

Source code reference:

<https://github.com/rajivmca2004/order-event-driven-service>

<https://github.com/rajivmca2004/order-event-driven-consumer-service>

Let's create a producer and consumer Spring Boot Java microservices using Spring Cloud Stream and Kafka messaging broker.

Creating producer microservice

Let's create producer service first:

We will follow the same approach, by creating this **order-event-driven-service** project on the Spring Initializr web portal:

<https://start.spring.io/>

We need to add the following dependencies:

The screenshot shows the Spring Initializr interface. In the 'Project' section, 'Maven Project' is selected. Under 'Language', 'Java' is chosen. In the 'Dependencies' section, the 'Spring Web' dependency is selected, indicated by a green bar. Other listed dependencies include 'Cloud Stream', 'Spring for Apache Kafka Streams', and 'Spring for Apache Kafka'. The 'Project Metadata' section contains fields for Group (com.online.store.demo), Artifact (order-event-driven-service), Name (order-event-driven-service), Description (Event driven project using Spring Boot and Kafka), Package name (com.online.store.demo.order-event-driven-service), and Packaging (JAR). Java version 15 is selected.

Figure 6.11: Spring Initializr of producer microservice

Now, add Spring Boot related application configuration in **application.yml** file. Here we need to define the microservice

application name with port Also, activate Spring micrometer for health check APIs.

Add Kafka related configuration like default port **9092** and event producer topic name **order_topic** where event messages can be sent and consumed. We will send JSON messages, so define **application/json** content type:

spring:

application:

name: order-event-driven-service

cloud:

stream:

kafka

kafka:

binder:

brokers:

bindings:

event-producer:

destination: order_topic

contentType: application/json

server:

port : 9001

management:

endpoint:

health:

enabled: true

show-details: always

endpoints:

web:

exposure:

include: '*'

jmx:

exposure:

include: '*'

Define **/produce** REST API to send order event messages after creating a new order to messaging broker:

@RestController

OrderController {

@Autowired

private OrderEventStreamInputService
orderEventProducerStreamService;

public Boolean createOrders(@RequestBody final PurchaseOrder
purchaseOrder) throws URISyntaxException {

return eventStreamService.produceEvent(purchaseOrder);

}

```
}
```

Create a data model **PurchaseOrder** which will be used to transform JSON messages to Java objects:

```
PurchaseOrder {  
  
    private Integer id;  
  
    private String data;  
  
    bytePayload;  
  
    public Integer getId() {  
  
        return id;  
  
    }  
  
    setId(Integer id) {  
  
        = id;  
  
    }  
  
    public String getData() {
```

```
    return data;  
  
}  
  
setData(String data) {  
  
    = data;  
  
}  
  
getBytePayload() {  
  
    return bytePayload;  
  
}  
  
bytePayload) {  
  
    = bytePayload;  
  
}  
  
}
```

Create an outbound interface which will have annotated with
@Output and topic name:

```
{  
  
    String OUTBOUND = «order_topic»;  
  
    @Output(OUTBOUND)  
  
    MessageChannel producer();  
  
}
```

Create a class **OrderEventStreamConfig** which will be annotated with **@EnableBinding** and **OrderEventProducerStream** class:

```
OrderEventStreamConfig {  
  
}
```

Now, create a producer class **OrderEventProducerStreamService** which will send purchase order data objects to Kafka's **order_topic** topic:

```
@Service  
  
OrderEventProducerStreamService {  
  
    @Autowired
```

```
private OrderEventProducerStream eventStream;

public Boolean produceEvent(PurchaseOrder purchaseOrder) {

    purchase order events=> id: purchaseOrder.getId() Actual message:
    purchaseOrder.getData();

        purchaseOrder.setBytePayload(purchaseOrder.getData().
getBytes());

    //Set Message channel or topic "order_topic"

    MessageChannel messageChannel =
eventStream.producer();

    //Send payload JSON message to Kafka message broker

    return
messageChannel.send(MessageBuilder.withPayload(purchaseOrder

        .setHeader(MessageHeaders.CONTENT_TYPE,
MimeTypeUtils.APPLICATION_JSON).build());

    }

}
```

It's time to run Zookeeper and Kafka by running these commands in separate terminals:

-- Browse to Kafka bin installation folder on your terminal

/Users//kafka_/bin

-- e.g. --

-- Start Zookeeper --

./zookeeper-server-start/config/zookeeper.properties

-- Start Kafka server --

./kafka-console-consumer.sh --topic order_topic --from-beginning --bootstrap-server

-- Create Kafka topic --

./kafka-topics.sh --describe --topic order_topic --bootstrap-server

--Check Kafka messages --

./kafka-console-consumer.sh --topic order_topic --from-beginning --bootstrap-server

Creating Consumer Microservice

Now, we need to a create consumer microservice to consume event messages from Kafka messaging broker:

In the next step, we will create **order-event-driven-consumer-service** consumer microservice, which will consume messages from Kafka message broker topic

The screenshot shows the Spring Initializr web interface. It has several sections:

- Project**: Maven Project (selected), Gradle Project.
- Language**: Java (selected), Kotlin, Groovy.
- Dependencies**: ADD DEPENDENCIES... button.
- Spring Boot**: Version 2.4.1 (selected), 2.4.2 (SNAPSHOT), 2.3.8 (SNAPSHOT), 2.3.7, 2.2.13 (SNAPSHOT), 2.2.12.
- Project Metadata**: Group: com.online.store.demo, Artifact: order-event-driven-consumer-service, Name: order-event-driven-consumer-service, Description: Demo project for Spring Cloud Stream for event driver, Package name: com.online.store.demo.order-event-driven-consumer-service, Packaging: Jar (selected), War.
- Dependencies** section:
 - Spring Boot DevTools** (DEVELOPER TOOLS): Provides fast application restarts, LiveReload, and configurations for enhanced development experience.
 - Cloud Stream** (SPRING CLOUD MESSAGING): Framework for building highly scalable event-driven microservices connected with shared messaging systems (requires a binder, e.g. Apache Kafka, RabbitMQ or Solace PubSub+).
 - Spring for Apache Kafka** (MESSAGING): Publish, subscribe, store, and process streams of records.
 - Spring for Apache Kafka Streams** (MESSAGING): Building stream processing applications with Apache Kafka Streams.
 - Spring Web** (WEB): Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Figure 6.12: Spring Initializr for consumer microservice

Now, add the Spring Boot-related application configuration in the **application.yml** file. Here we need to define the microservice

application name with port Also, activate Spring micrometer for health check APIs. Add Kafka-related configuration like default port **9092** and event producer topic name **order_topic** where event messages can be sent and consumed. We will send a JSON message, so define **application/json** content-type:

spring:

application:

name: order-event-driven-consumer-service

cloud:

stream:

kafka

kafka:

binder:

brokers:

bindings:

event-consumer:

destination: order_topic

contentType: application/json

server:

port : 9002

management:

endpoint:

health:

enabled: true

show-details: always

endpoints:

web:

exposure:

include: '*'

jmx:

exposure:

include: `'*'`

Create a data model **Message** which will be used to consume messages from Kafka topic and transform JSON serialized messages to Java objects:

```
Message {
```

```
    private Integer id;
```

```
    private String data;
```

```
    bytePayload;
```

```
    public Integer getId() {
```

```
        return id;
```

```
}
```

```
    setId(Integer id) {
```

```
        = id;
```

```
}
```

```
public String getData() {
```

```
    return data;
```

```
}
```

```
    setData(String data) {
```

```
        = data;
```

```
}
```

```
    getBytePayload() {
```

```
        return bytePayload;
```

```
}
```

```
    bytePayload) {
```

```
        = bytePayload;
```

```
}
```

```
}
```

Create an inbound interface which will be annotated with **@Input** and topic name:

```
OrderEventConsumerStream {  
  
    String INBOUND =  
  
        @Input(INBOUND)  
  
    SubscribableChannel consumer();  
  
}
```

Create a class **OrderEventConsumerStream** which will be annotated with **@EnableBinding** and **OrderEventConsumerStream** class:

```
OrderEventConsumerStreamConfig{  
  
}
```

Now, create a producer class **OrderEventConsumerStreamService** which will consume purchase order data object to Kafka's **order_topic** topic:

```
@Configuration
```

```
OrderEventConsumerStreamService {  
  
    @StreamListener(OrderEventConsumerStream.INBOUND)  
  
    consumeEvent(@Payload Message msg) {  
  
        purchase order message consumed ==> id: " + msg.getId() + "  
        Purchase Order message: "  
  
        + msg.getData() + " bytePayload: " +  
        msg.getBytePayload();  
  
    }  
  
}  
}
```

Test event driven microservice

Let's verify this event-driven use case and communication between order producer and consumer microservices:

Hit **/orders/produce** REST API producer microservice will send a purchase order message/payload to Kafka topic:

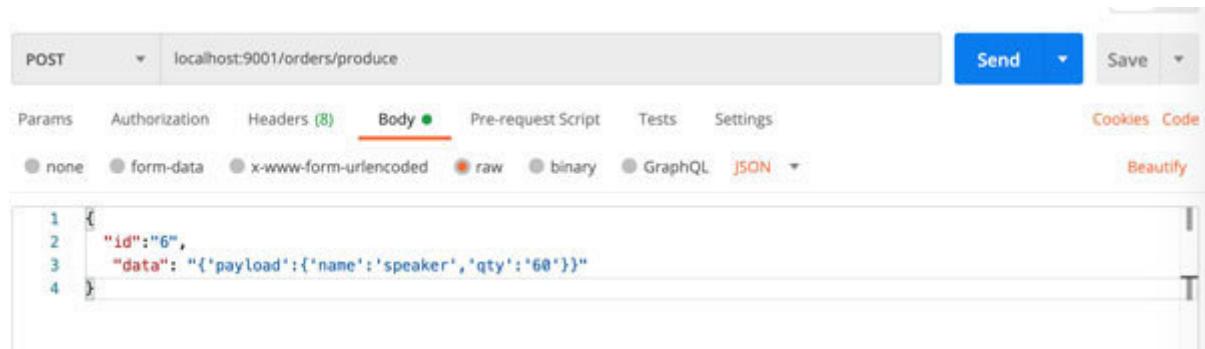


Figure 6.13: Send message to Kafka topic

Hit **/orders/produce** REST API consumer microservice will receive purchase order message from Kafka topic:

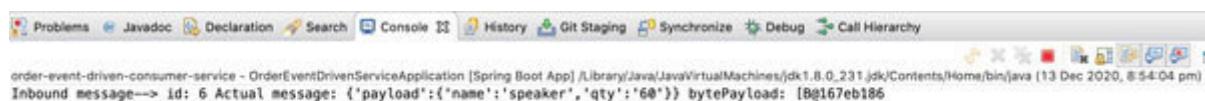


Figure 6.14: Console log of consumer microservice from Kafka topic

Conclusion

In this chapter, we discussed a detailed reactive design pattern overview and architecture. We also compared Spring WebFlux and Spring MVC. We discussed a non-blocking synchronous API with its use cases. We have gone through the challenges of traditional REST API. We also discussed the advantages of the reactive programming model and its usage. Also, we did a hands-on lab of reactive microservices using Spring Webflux.

We have discussed event-driven asynchronous microservices design and its advantages. Finally, implemented event-driven microservices by building consumer, producer, and how to test these use cases using Spring Cloud Stream and Kafka.

In the next chapter, we will cover overview of API gateway and its advantages. It will also cover the best practices, tools, and techniques of securing microservices.

Points to Remember

We can easily have an application with the Spring MVC model for one controller and the WebFlux model for another controller.

We can't apply reactive design patterns for all the use cases. It works for certain use cases where TPS load is high, network latency is low and the cost is high. Similarly, we should not migrate the existing REST API to non-blocking without considering use cases. For example, file upload to any social portal where non-blocking design will be suitable.

Spring Webflux's mono class is for getting single records and Flux for more than one record.

Key terms

Event-driven architecture.

Transaction per second.

Reactive Relational Database Connectivity.

Create Read Update Delete.

Data Access Object.

References

Project

Spring Cloud

Spring

Kafka official

Spring Reactive REST

Spring web

What do you mean by

CHAPTER 7

The API Gateway, Security, and Distributed Caching with Redis

The API gateway is an aggregator technology based on the gateway design pattern. It's based on a front-end controller design pattern where all client traffic requests are handled and forwarded to backend services. The API gateway works as a reverse proxy which protects server-side API resources by providing a unified interface and other advanced features of API traffic routing management, API orchestration, authentication, authorization, security, load balancing, monitoring, tracing, rate limiting, caching, circuit breaker and so on.

This chapter will cover an overview of the API gateway and its advantages with source code examples like how to expose REST APIs of microservices externally with the Spring Cloud Gateway, rate limiting, distributed caching, and SSO with the help of **Spring** and **Spring Cloud**. This chapter will also cover best practices, tools, and techniques of securing microservices.

Let's get started!

Structure

In this chapter, we will cover the following topics:

Overview of the API gateway

API gateway features and advantages

Best practices of the API gateway design

Overview of the Spring Cloud Gateway

Implementing the Spring API Gateway

Prerequisite

Overview of Distributed caching

API caching with Redis distributed caching

Redis cluster architecture for high availability (HA)

Implementing Redis distributed caching with the Spring Cloud Gateway

Prerequisite

API gateway rate limiting

Implementing API gateway rate limiting with Redis and Spring Boot

Prerequisite

Best practices of API security

API security with SSO using Spring Security, OAuth2, Keycloak, OpenID, and JWT tokens

Prerequisite

Objectives

After studying this chapter, you should be able to learn about the API gateway features, advantages, and best practices. This chapter will help you to implement to route traffic to microservices, rate limiting with Redis distributed caching, and API authentication using Spring Security, Oauth2, Keycloak, and JWT API secret tokens of the Spring Cloud Gateway. This chapter will help you to understand Spring Cloud Gateway predicates, API security guidelines, and best practices. It also orchestrate multiple API calls at backend, when any API is dependent on many other REST APIs.

Overview of the API gateway

It's a reverse proxy which protects server-side API resources by providing a unified interface. The API gateway is an interface between front-end client requests and back-end microservices. It works as a filter for any incoming requests to the back-end microservices layer and routes its traffic to the designated microservice and provides features. It provides a facade-based controller which provides an abstraction layer which decides which API will serve which client request.

It's deployed separately as a separate orchestrator application. Clients and other microservices first interact with the API gateway, it has all routing and filtering business logic which routes the client request to the appropriate microservice APIs for further processing.

Refer comparison of service mesh with the API gateway in the last [Chapter 12: Service Mesh and Kubernetes Alternatives of Spring Cloud.](#)

This is a high-level architecture of the API gateway and communication with microservices:

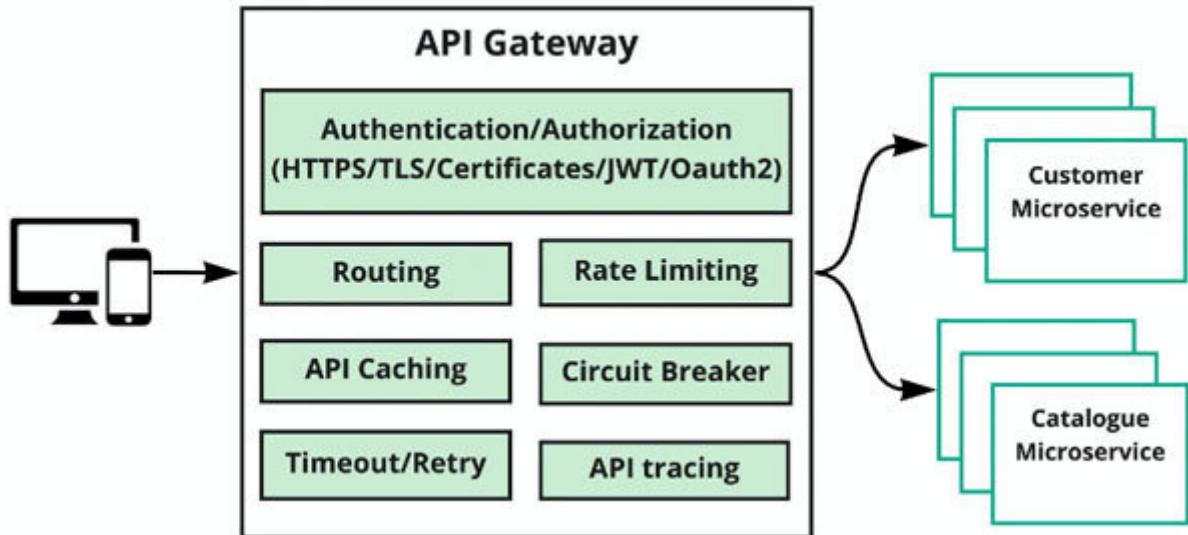


Figure 7.1: API gateway architecture

The API gateway manages multiple workloads from different kinds of clients like web, mobile, IoT, and so on. In **Backend for Frontend** or the micro-frontend microservices pattern, there could be separate API gateways for web, mobile, and third-party applications, which forwards requests to backend services using **REST** and **gRPC** protocols.

API gateway features and advantages

These are important API gateway features and advantages for the microservices architecture:

API traffic It works as a proxy service between front end clients/ UI and backend microservices. It has the responsibility to route client requests to target microservices based on customized routing rules. It provides a single point of controlling and monitoring traffic of all REST APIs. It helps to configure and add new routing of APIs with simple configuration.

API It reduced round trips by aggregating multiple API calls for a single client request.

Decoupling Decoupling microservices inter-communication by connecting directly through the API gateway.

Path Web apps are exposed to external services such as public API names, which is different from internal service URI names due to security and privacy. It also helps to re-write the API path to the internal service URI path.

Service It can be also integrated with various service discovery APIs and services for service discovery like Spring Cloud Discovery Client Eureka, Zuul2, HashiCorp Consul, and so on.

It saves a lot of round trips of client requests to backend REST APIs by caching API responses to distributed caches like Redis or GemFire, and so on. Client apps which have similar requests will be served by an already cached response from the caching server, without hitting backend APIs. It also improves the API response time and performance. It also saves extra round trips and computes resources and memory.

Authentication & Authorization It provides very important API security features by integrating with external A&A APIs like Spring Security with the JWT token and other external authentication providers like SSO based Okta, OAuth2, and so on.

Rate The API gateway also has a feature to restrict the limit of client calls to any given REST API because of security, paid subscription and cost saving reasons, and so on. The API gateway can be integrated with any persistence database like Redis or GemFire, and so on to store and track APIs calling history by its client and a counter for any number of hits. This is all configurable.

User It also provides a user quota to decide which user can access which API and how many times. Users can't access the API after the quota is full.

API The throttling counter is based on certain conditions based on the REST API configuration like the HTTP method type, header, endpoint URI, and so on. It increases the counter when any API requests satisfy these combinations of condition(s). The time period is per second for throttling.

API Some API gateways also provide the tracing feature to trace request-response API logs of microservices, which helps to debug any issue during intercommunication of microservices.

Load balancing to multiple It has an internal rule-based mechanism to route the load based on some condition like weight-based load balancing. For example, client requests can be routed to two different APIs based on weightage like 60% and

High It can be scaled also by deploying multiple API gateways to provide **high availability**

The API gateway provides a centralized proxy server to manage rate limiting, bot detection, authentication, authorization, CORS security filter, and so on. API gateways allow setting up a datastore such as Redis to store the client's session.

Best practices of the API gateway design

Here are some best practices for enterprise grade API Gateway:

Orchestrate and aggregate multiple dependent API calls in a common API gateway module.

Cache frequent client REST API responses for better performance and avoid extra backend/DB calls. Also, configure adequate eviction policy for cached data.

Implement client-side SSL/TLS certificates for HTTPS to secure REST APIs endpoints. Prefer mTLS for both client and server-side public-private key verification.

Rotate TLS certificates periodically for full proof security.

Handle client A&A at the API gateway layer as a separate service outside of business logic and provide access to client requests based on authentication and authorization.

Encrypt data when it is transmitted across multiple data centers/AZs.

Your REST API should not be exposed to the public Internet. It should be accessed internally behind the firewall and can be

accessed only through the API gateway after client authentications and verifications.

Monitor API communications using tracing logs. It helps to quickly debug applications.

Observe metrics of microservices using API gateway plug-ins.

Deploy the API gateway on a separate container with multiple replicas set for HA.

Configure the API gateway auto alert feature for any malfunction at microservices REST API endpoints.

Add filters which work as a firewall to filter unwanted API requests to reduce the number of calls.

API gateway internal API communication should happen using HTTP calls and external using HTTPS through public side load balancers.

Restrict client requests at the API gateway layer to avoid DDoS kind of unethical hacking.

Overview of the Spring Cloud Gateway

The Spring Cloud Gateway is an API gateway. It's an open source based on the Java language. It's a Spring framework project. It has tons of features. It can be embedded with code and can also be deployed as a separate service and scaled easily on Kubernetes containers.

The Spring Cloud Gateway is capable of handling client requests traffic by routing to desired microservices using the gateway handler mapping and aggregate responses of different back-end REST API endpoints of microservices.

Internally, the Spring Cloud Gateway runs on Netty the non-blocking web server which provides asynchronous request processing for faster non-blocking processing of client requests.

It can be configured using application properties/YAML files and embedded as a Java code, which helps developers to write the configuration code:

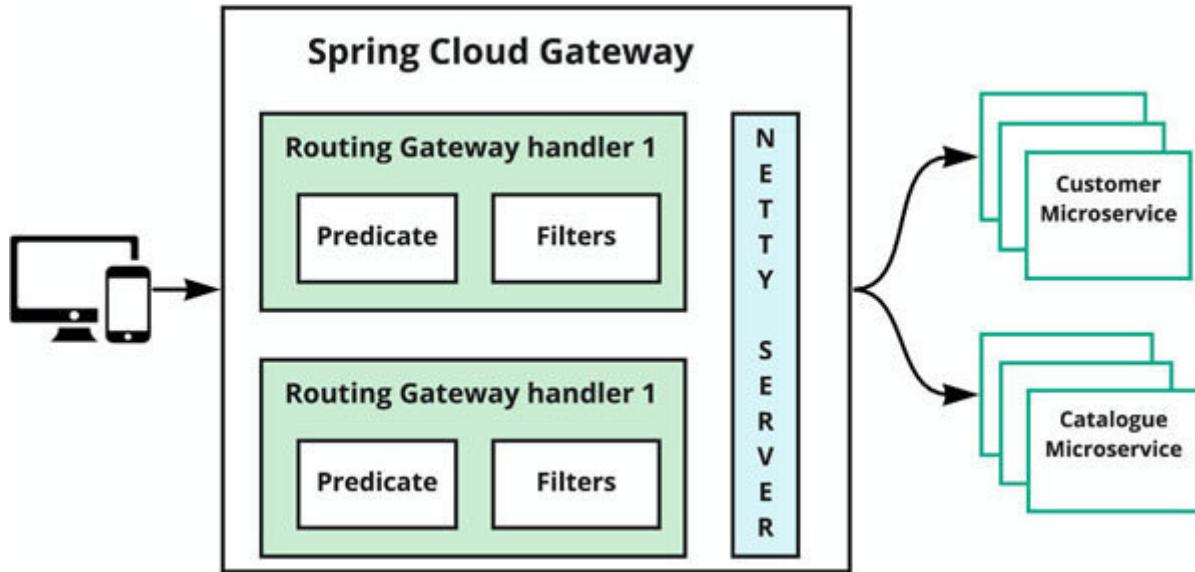


Figure 7.2: Spring Cloud Gateway architecture

It's based on the following three major pillars:

Route traffic to It handles routing client requests to designated REST API endpoint destinations. Imagine a use case where only the **/catalogue** API is being called. In this use case, the API gateway forwards client requests with payload to the **Catalogue** microservice.

It helps to add the condition on incoming client requests like checking request URI, parameter, or assigning weight.

It helps to implement the Spring framework web filters. Developers can modify the request and response based on the client's preferences or security reasons. Developers can also add their custom filtering logic to filter incoming client requests and outgoing responses with no source code changes.

Implementing Spring API Gateway

Let's do some hands-on coding labs. In this section, we will implement the Spring Cloud Gateway routing features. In this code exercise, we will cover major API gateway features route traffic, filters, and predicates.

We will use the **dynamic routing configuration** by making changes in application properties files. It will make changes applied dynamically without restarting microservices web apps.

It will be deployed as a separate microservice on the Kubernetes container and can be deployed on multiple containers to provide HA.

Prerequisite

The following are the basic installation requirements to build:

Java 8+

Spring Boot v2.4.1+

Spring Cloud Gateway

Java v8.x+

Source code reference:

<https://github.com/rajivmca2004/spring-gateway-demo>

Let's create a simple Java app using Spring Boot and the Spring Cloud Gateway. We will use the existing microservices to forward client requests which have been developed in previous chapters:

Create the **spring-gateway-demo** project using the Spring Initializr web portal:

<https://start.spring.io/>

Here is the screenshot for the reference:

The screenshot shows the Spring Initializr web interface. On the left, there are sections for 'Project' (Maven Project selected), 'Language' (Java selected), and 'Spring Boot' (2.4.2 (SNAPSHOT) selected). Below these are 'Project Metadata' fields: Group (com.online.store.demo), Artifact (spring-gateway-demo), Name (spring-gateway-demo), Description (Demo project for Spring Cloud Gateway), Package name (com.online.store.demo spring-gateway-demo), and Packaging (Jar selected). At the bottom left, Java version options (15, 11, 8) are shown with 8 selected. On the right, there's a 'Dependencies' section with an 'ADD DEPENDENCIES...' button, followed by sections for 'Gateway' (Spring Cloud Routing selected), 'Spring Boot Actuator' (OPS), and 'Spring Boot DevTools' (Developer Tools).

Figure 7.3: Spring Boot project initial setup

Now, we will define Spring Cloud Gateway routes, predicates, and filters in the **application.yaml** file.

In this example, we will configure API gateway routing for two separate microservices **customer-management-service** and We will test and verify these in the upcoming points.

We will use filters attributes of the Spring Cloud Gateway in the rate limiting implementation section in the same chapter:

1. **spring:**
2. **application:**

```
3.      name: spring-gateway-demo
4.      redis:
5.          host: localhost
6.          port: 6379
7.      cloud:
8.          gateway:
9.              routes:
10.                 - id: catalogues_route
11.                   uri: http://localhost:8010
12.                   predicates:
13.                     - Path=/catalogue
14.                     - Weight=group^, 6
15.                 - id: customers_route
16.                   uri: http://localhost:8011
17.                   predicates:
18.                     - Path=/customers
```

If the client goes to `http://localhost:8080/customers`, then it will route to the **customer-management-service** microservice REST API `http://localhost:8011/customers` which is running as a separate web service on a different container and port number:

The screenshot shows a Postman interface with a GET request to `http://localhost:8080/customers`. The response body is a JSON array containing four customer objects:

```
[{"id": 1, "name": "Rajiv Srivastava", "email": "rajiv@test.com"}, {"id": 2, "name": "Scott Peter ", "email": "scott@test.com"}, {"id": 3, "name": "John Stevens", "email": "john@test.com"}, {"id": 4, "name": "Sharad Verma", "email": "sharad@test.com"}]
```

Figure 7.4: REST API response using

If a client goes to `http://localhost:8080/catalogue`, then it will route to the **customer-management-service** microservice REST API `http://localhost:8010/catalogue` which is running as a separate web service on a different container and port number:

The screenshot shows the Postman client interface with a GET request to `http://localhost:8080/catalogue`. The 'Params' tab is selected, showing a single query parameter 'Key' with value 'Value'. The 'Body' tab is selected, displaying the JSON response:

```
1 [  
2   {  
3     "id": 1,  
4     "name": "Fan",  
5     "price": 100.5  
6   },  
7   {  
8     "id": 2,  
9     "name": "Cooler",  
10    "price": 200.51  
11  },  
12  {  
13    "id": 3,  
14    "name": "Refrigerator",  
15    "price": 600.0  
16  },  
17  {  
18    "id": 4,  
19    "name": "Television",  
20    "price": 900.5  
21  }  
22 ]
```

The status bar at the bottom indicates: Status: 200 OK Time: 20 ms Size: 280 B.

Figure 7.5: REST API response using Postman client

Distributed caching overview

When there is a need to improve the performance of web applications (microservices) every millisecond counts, the API gateway provides a powerful feature of distributed caching where API responses can be cached and be available for all distributed microservices. It may be spun to multiple servers on separate Kubernetes containers. In caching, objects/data store in high-speed static RAM memory for faster access. Memory caching is effective because all microservices apps access the same set of cached data. The objective of distributed cached memory is to store program instructions and data that are used repeatedly by clients.

Distributed caching is an important caching strategy for decreasing a distributed microservices apps latency and improving its concurrency and scalability for better performance. The cache eviction strategy should be also configured regularly to replace it with the latest fresh data.

According to a research by

New research by Google has found that 53% of mobile website visitors will leave if a webpage doesn't load within 3 seconds.

The average load time for sites is 19 seconds on a 3G connection and 14 seconds on a 4G connection.

Read performance can be improved by caching frequently used API responses like popular product metadata and so on.

API caching with Redis distributed caching

Redis is an open source in-memory data structure project implementing a distributed caching and in-memory key-value database. Redis supports different kinds of abstract data structures such as strings, lists, maps, sets, sorted sets, hyperlog, bitmaps, streams, and spatial indexes.

Redis is a high performant, in-memory, data structure server (not just a key-value store). On large-scale distributed systems with a high number of API calls per second, Redis is a perfect distributed caching solution for this kind of distributed enterprise microservice architecture. It's faster than usual database calls because Redis serves data directly from static RAM cache memory.

Apps are responsible to fetch data from the database and push to the Redis cluster on the leader node which updates/writes all new cache data entry into the Redis cluster. The Redis leader writes/updates data to the Redis follower nodes. The Leader does replicate writes to one or more Redis followers. The **leader-follower replication** is done asynchronously. The Redis server runs in the following two modes:

Redis leader mode

Redis follower/Redis replica mode

We can configure Redis to choose a mode to write and read from. It is recommended to serve writes through the Redis leader and reads through the Redis follower.

The following diagram depicts the Redis architecture and how it works with microservices apps on Kubernetes containers:

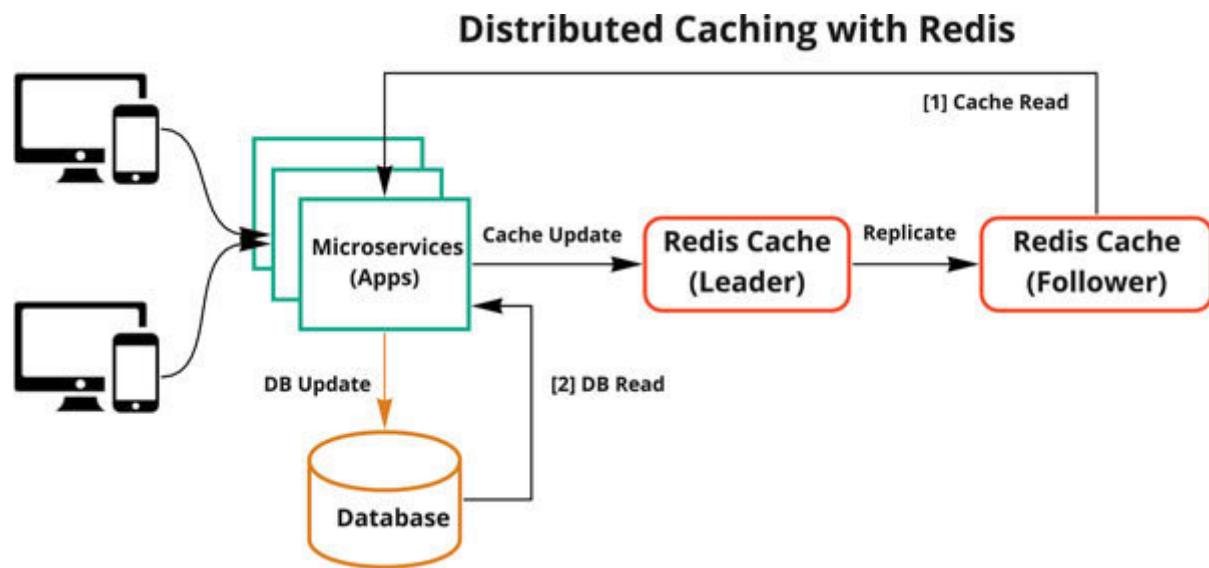


Figure 7.6: Distributed caching with Redis

Redis cluster architecture for high availability (HA)

Redis works on the **leader-follower** architecture. By default, Redis creates six nodes (three leaders + three followers) with a single replication. It's a preferred architecture where three leader nodes are needed for a failover recovery; if any leader goes down, then other leader(s) can take control and manage Redis follower nodes.

All leader nodes talk to all followers through the Gossip protocol.

Every leader should have one follower minimum; certainly can have more followers than leaders, which would be preferable to having a single follower per leader so that you can have one fail and still have a backup follower for some redundancy post-failover.

Clients write on the leader node and read from follower nodes. Clients can directly connect with leader nodes for reads if followers are not available or down. Every leader node replicates the cached data to its follower. It could be one or many followers. They are all configurable.

All leaders and followers check the health status of every node by using the gossip protocol.

The following diagram depicts the Redis's leader-follower recommended cluster architecture:

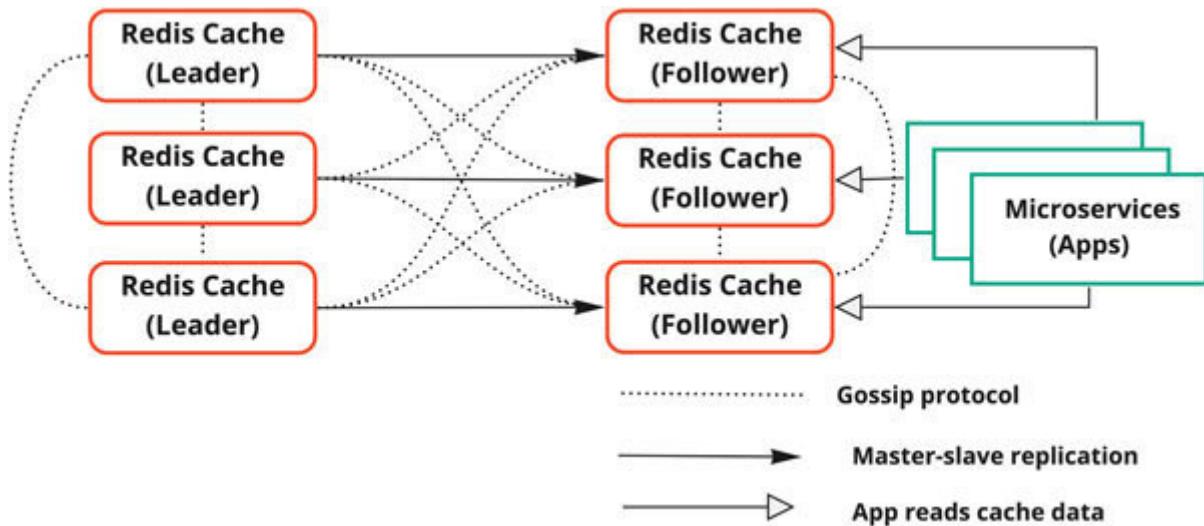


Figure 7.7: Redis leader-follower cluster architecture

Implementing Redis distributed caching with the Spring Cloud Gateway

Let's do some hands-on coding labs. In this section, we will implement distributed caching using Redis and the Spring Cloud Gateway.

Redis is an open-source data structure store to cache data in key-value pairs. It's used as a database, cache, and message broker.

This sample Spring Boot microservice app will cache data in Redis, which will be available for any client requests for the same query request. The Redis caching service will be deployed in a distributed environment on multi nodes on a cluster which will be available for all the microservices to cache and use.

Cached data will have an eviction policy to be removed from cache after a certain period. Also, data should be updated/refreshed periodically, and so on.

Prerequisite

These are basic installation requirements to build:

Spring Boot v2.4.1+

Spring Cloud Gateway v2.4.1+

Spring Redis data v2.4.1+

Redis v6.0.9+

Java v8.x+

Source code reference:

<https://github.com/rajivmca2004/catalogue-cache-service>

Let's create a simple Spring Boot Java microservice using the Spring Cloud Gateway:

Create the **catalogue-cache-service** project using the Spring Initializr web portal:

<https://start.spring.io/>

Add **Spring Data Redis** (Access + Spring embedded H₂ Database for the test environment API can also be used), **Spring Data REST Repositories** to expose data as REST APIs, and **Spring Boot DevTools** for easy and auto deployment of Spring Boot apps when any changes happen at the source code or configuration. Here is the screenshot for reference:

The screenshot shows the Spring Initializr web interface. At the top, there's a logo and the text "spring initializr". Below it, there are sections for "Project", "Language", and "Dependencies".

Project: Maven Project (selected), Gradle Project.

Language: Java (selected), Kotlin, Groovy.

Dependencies: ADD DEPENDENCIES... (button) + 8

Selected Dependencies:

- Spring Data Redis (Access+Driver)** NOSQL: Advanced and thread-safe Java Redis client for synchronous, asynchronous, and reactive usage. Supports Cluster, Sentinel, Pipelining, Auto-Reconnect, Codecs and much more.
- Spring Web** WEB: Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.
- H2 Database** SQL: Provides a fast in-memory database that supports JDBC API and R2DBC access, with a small (2mb) footprint. Supports embedded and server modes as well as a browser based console application.
- Spring Data JPA** SQL: Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.
- Rest Repositories** WEB: Exposing Spring Data repositories over REST via Spring Data REST.
- Spring Boot DevTools** DEVELOPER TOOLS: Provides fast application restarts, LiveReload, and configurations for enhanced development experience.

Project Metadata:

- Group: com.online.store.demo
- Artifact: catalogue-cache-service
- Name: catalogue-cache-service
- Description: Catalogue project for distributed caching with Red
- Package name: com.online.store.demo.catalogue-cache-service
- Packaging: Jar (selected), War
- Java: 15 (radio button), 11 (radio button), 8 (radio button selected)

Figure 7.8: Spring Boot project initial setup

This Spring Boot microservice application will have the Redis server configuration details and JPA configurations as follows:

```
1. spring:  
2.   application:  
3.     name: catalogue-service  
4.   jpa:  
5.     hibernate:  
6.       ddl-auto: update  
7.   cache:  
8.     type: redis  
9.   redis:  
10.    host: localhost  
11.    port: 6379  
  
12.  
13. server:  
14.   port : 8010  
15. springdoc:
```

The Redis server runs on the default port which can be configurable. This sample microservice will run on port **8010** and cache API response in RAM memory.

Add the **@EnableCaching** annotation on the Spring Boot **main** class to enable auto caching configuration to Spring containers:

```
1. @EnableCaching  
2. @SpringBootApplication  
3. CatalogServiceApplication {  
4.  
5.   main(String[] args) {  
6.     args);
```

```
7.    }
8. }
```

Create a data model to hold data objects and persist in cache. Also, implement the **Serializable** interface to serialize this Java object. Add the **serialVersionUID** property in this entity bean:

```
1. @Entity
2. Catalogue implements Serializable{
3.
4.     serialVersionUID = 'L;
5.

6.     @Id
7.     @GeneratedValue(strategy = GenerationType.AUTO)
8.     private Long id;
9.
10.    @Column(nullable =
11.    private String name;
12.
13.    @Column
14.    private Double price;
15.
16.    public Catalogue() {
17.
18.    }
19.
20.    public Catalogue(String name, Double price) {
21.        = name;
22.        = price;
23.    }
```

```
24.  
25.      // Getter and setters  
26.  
27. }
```

Create a data repository using JPA:

```
2. CatalogueRepository extends JpaRepository<Long> {}
```

Create a runner class by implementing **CommandLineRunner** and insert hard-coded values which will be persisted in the embedded H₂ database for testing purpose. You can persist in any SQL database in the production environment:

```
@Component
```

```
CatalogueRepoCLR implements CommandLineRunner {
```

```
private CatalogueRepository catalogueRepository;
```

```
@Autowired
```

```
public CatalogueRepoCLR(CatalogueRepository catalogueRepository)  
{
```

```
    = catalogueRepository;
```

```
}
```

```
@Override  
  
run(String... args) throws Exception {  
  
    //Warning: Values are hard-coded for POC purpose. These value  
    //should be outside the code  
  
    catalogueRepository.findAll().foreach(System.out::println);  
  
}
```

} Next, expose REST APIs by creating a controller class of Add the **@Cacheable** annotation to those APIs, which you want to cache their response object for other client-requests for the same query data.

Let's try out these options on the Redis cache:

Read from Redis cache (API) In this case, the **/catalogue/{id}** REST API will do all these tasks: cached data to the Redis database, pull data response from database and store in the Redis cache, and pull cached data from the Redis cache. Redis caches data

into catalogues key and iterates the response based on the given unique ID. The client retrieves data next time from the Redis cache for the same query:

```
//Cache catalogue object for feteched id as a Redis catalogues key

@Cacheable(value = key = "catalogue" + id)

public Optional<Catalogue> fetchCatalogue(@PathVariable Long id) {
    Catalogue catalogue = catalogueRepository.findById(id);
    if (catalogue != null) {
        Cacheable<Catalogue> cacheable = new Cacheable<Catalogue>(catalogue);
        cacheable.setKey("catalogue" + id);
        cacheable.setTtl(Duration.ofSeconds(1));
        cacheable.setCacheManager(cacheManager);
        cacheManager.getCache("catalogues").put(cacheable);
    }
    return catalogue;
}
```

Update into the Redis There is a separate **catalogue/update** API for updating this **Catalogue** object data in the Redis cache:

```
//Updating Cache
```

```
@Cacheable(value = key = "catalogue" + id)
```

```
public Catalogue updateCatalogue(@RequestBody Catalogue catalogue) {
```

```
catalogueRepository.save(catalogue);

return catalogue;

}
```

Clear the Redis cache: You need to add the `@CacheEvict` annotation to those methods which you want to use to remove the cache data from cache. Cache can be cleared for given data by DELETE HTTP API Please refer to the following code:

```
//Clear Cache

@CacheEvict(value = key =
    deleteUserByID(@PathVariable Long id) {

catalogue with id id);

catalogueRepository.deleteById(Long.valueOf(id));

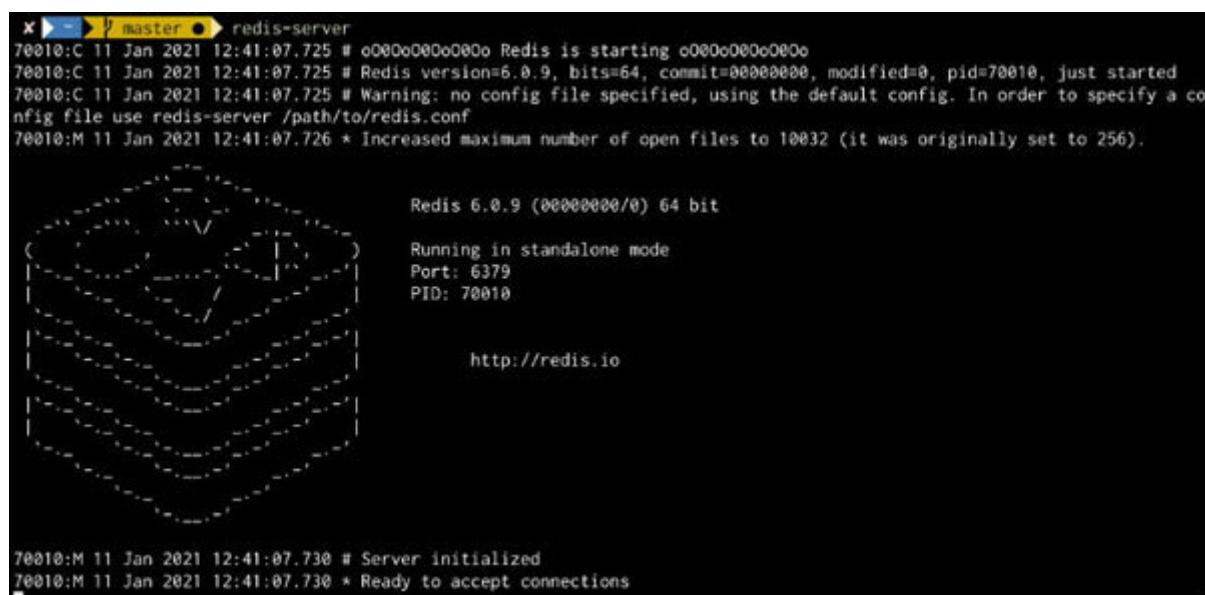
}
```

It's time to download and set up the Redis open source by referring to this **Quick start** reference:

<https://redis.io/topics/quickstart>

You can deploy and run Redis on Docker or Kubernetes by pulling the official image of Redis.

Run the Redis server by the command. You can also check the Redis version here. We can see the Redis version is **6.0.9** and it's running on the default port It can be customized by making changes in the Redis configuration:



```
x -> master > redis-server
70010:C 11 Jan 2021 12:41:07.725 # 000000000000 Redis is starting 000000000000
70010:C 11 Jan 2021 12:41:07.725 # Redis version=6.0.9, bits=64, commit=00000000, modified=0, pid=70010, just started
70010:C 11 Jan 2021 12:41:07.725 # Warning: no config file specified, using the default config. In order to specify a co
nfig file use redis-server /path/to/redis.conf
70010:M 11 Jan 2021 12:41:07.726 * Increased maximum number of open files to 10032 (it was originally set to 256).

                               Redis 6.0.9 (00000000/0) 64 bit
                               Running in standalone mode
                               Port: 6379
                               PID: 70010

                               http://redis.io

70010:M 11 Jan 2021 12:41:07.730 # Server initialized
70010:M 11 Jan 2021 12:41:07.730 * Ready to accept connections
```

Figure 7.9: Redis server status

Run the following command **redis-cli ping** to verify the Redis server running and health status. Redis will respond with a **PONG** message if it is running healthy:

```
-> master > redis-cli ping  
PONG  
-> master >
```

Figure 7.10: Check Redis status using CLI

Now, we can verify by calling the REST API using Postman or any other REST client. It will try to fetch from the Redis server first; if not available, then this API will pick from the database and cache it to Redis and return the same response:

The screenshot shows a Postman request configuration and its resulting response. The request is a GET to `http://localhost:8010/catalogue/1`. The response is a 200 OK status with a response time of 6 ms and a body size of 199 B. The response JSON is:

```
1 {  
2   "id": 1,  
3   "name": "Fan",  
4   "price": 100.5  
5 }
```

Figure 7.11: API response from Redis cache

We will keep on hitting the same API three times. Every time Redis will increment its cache hit count:

```
~ ➤ ⚡ master ➤ redis-cli flushall
OK
~ ➤ ⚡ master ➤ redis-cli incr mycounter
(integer) 1
~ ➤ ⚡ master ➤ redis-cli incr mycounter
(integer) 2
~ ➤ ⚡ master ➤ redis-cli incr mycounter
(integer) 3
```

Figure 7.12: Redis CLI output

Also, if you see the application logs, you will notice that log writes for first time, it won't write from subsequent API calls as shown in the following screenshot:

```
2021-01-11 15:06:56.423 INFO 82484 --- [nio-8010-exec-1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
2021-01-11 15:06:56.425 INFO 82484 --- [nio-8010-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 2 ms
2021-01-11 15:35:21.783 INFO 82484 --- [nio-8010-exec-5] c.o.s.d.controller.CatalogueController : Getting user with ID 1.
2021-01-11 16:16:38.863 WARN 82484 --- [l-1 housekeeper] com.zaxxer.hikari.pool.HikariPool : HikariPool-1 - Thread starvation or clock leap detected (housekeeper delta=28m19s432ms).
```

Figure 7.13: Spring Boot app server logs

Another easy way to demonstrate this is for the API response to return a timestamp and thereafter for all subsequent hits the timestamp stays the same.

API gateway rate limiting

Rate limiting is a technique of restricting overwhelming client calls to app server APIs. It also prevents from DDoS (Distributed Denial of Service). It improves the API performance to limit API requests to backend resources. It's also known as rate limiting, which is an essential component of Internet security.

A **Distributed Denial-of-Service** attack is a malicious attempt to disrupt the normal traffic of a targeted API server or microservices REST API endpoints by overwhelming the target API or network infrastructure with unexpected flood of internet traffic.

API providers or owners typically measure processing limits in **Transactions Per Second**. Some systems may have physical limitations on data transmission; they may have dependency on compute or memory hardware resources. This is called **backend rate**.

To secure an API from being overwhelmed, API owners often restrict a fix limit on the number of requests, or the quantity of data the clients can consume. This is called **application rate**.

When a client sends too many requests, API rate limiting can throttle client connections instead of disconnecting them immediately. Throttling allows clients to still use your services while still protecting the API. We should always consider that

there is always the risk of API requests timing out, and the open connections also raise the risk of DDoS attacks.

Implementing the API gateway rate limiting with Redis and Spring Boot

Let's do some hands-on coding labs. In this section, we will implement API rate limiting using Redis and the Spring Cloud Gateway.

This sample Spring Boot microservice app will count inbound client requests and expose REST API through this microservice.

Prerequisite

These are basic installation requirements to build:

Spring Boot v2.4.1+

Spring Cloud Gateway v2.4.1+

Spring Redis data v2.4.1+

Redis v6.0.9+

Java v8.x+

Source code reference:

<https://github.com/rajivmca2004/spring-gateway-demo>

Here are the steps to follow:

Create the **spring-gateway-demo** project using the Spring Initializr web portal:

<https://start.spring.io/>

Add dependencies of Spring Data Reactive Spring Boot and Spring Boot

The screenshot shows the Spring Initializr web interface. On the left, there are sections for 'Project' (Maven Project selected), 'Language' (Java selected), and 'Spring Boot' (2.4.1 selected). Under 'Project Metadata', fields include 'Group' (com.online.store.demo), 'Artifact' (spring-gateway-demo), 'Name' (spring-gateway-demo), 'Description' (Rate limiting project for Spring Boot), 'Package name' (com.online.store.demo.spring-gateway-demo), 'Packaging' (JAR selected), and 'Java' (11 selected). On the right, under 'Dependencies', 'Gateway' (Spring Cloud Routing) is selected. Other listed dependencies include 'Spring Data Reactive Redis' (NOSQL) and 'Spring Boot Actuator' (OPS). A button for 'ADD DEPENDENCIES...' is visible.

Figure 7.14: Spring Boot project initial setup

Rate limiting with the Spring Cloud Gateway requires to configure the following two parameters:

How many requests per second for client without any drop.

Maximum number of requests a client is allowed to do in a single second (TPS). This is the number of tokens the token bucket can hold. Setting this value to **zero** will block all requests.

A steady rate is accomplished by setting the same value in **replenishRate** and Temporary bursts can be allowed by setting **burstCapacity** higher than In this case, the rate limiter needs to be allowed sometime between bursts (according to as two consecutive bursts will result in dropped requests **HTTP 429 Too Many**

In this example, we have set and Let's customize the **application.yaml** file:

```
1. spring:  
2.   application:  
3.     name: spring-gateway-demo  
4.   redis:  
5.     host: localhost  
6.     port: 6379  
7.   cloud:  
8.     gateway:  
9.       routes:  
10.        - id: catalogues_route  
11.          uri:  
12.            predicates:  
13.              - Path=/catalogue  
14.              - Weight=group1, 6  
15.            filters:  
16.              - name: RequestRateLimiter  
17.                args:  
18.                  redis-rate-limiter.replenishRate: 2  
19.                  redis-rate-limiter.burstCapacity: 4
```

We need to add this configuration in the source code to activate the rate limit functionality with Spring Boot. The **GatewayFilter** takes an optional **keyResolver** parameter and parameters specific to the rate limiter implementation using Redis. The **keyResolver** parameter is a bean that implements the **KeyResolver** interface. It allows you to apply different strategies to derive the key for limiting requests:

```
//It requires only for Rate limiting feature
```

```
@Bean
```

```
KeyResolver userKeyResolver() {
```

```
    return exchange -> Mono.just("1");
```

```
}
```

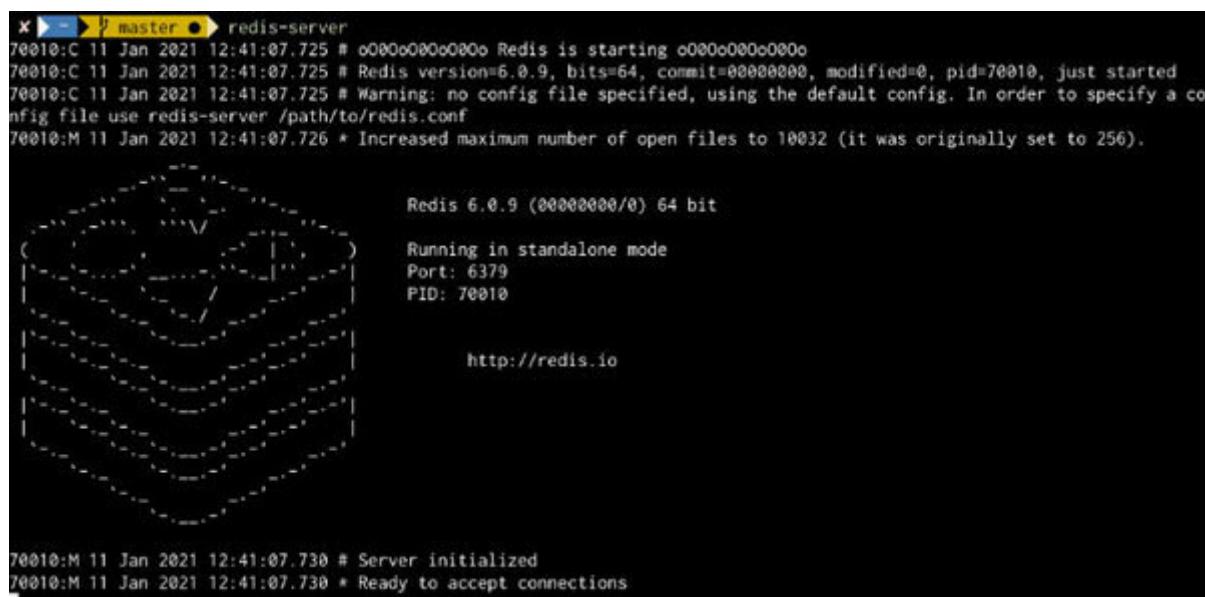
It's time to download and set up the Redis open source by referring to this Quick start reference:

<https://redis.io/topics/quickstart>

You can deploy and run Redis on Docker or Kubernetes by pulling the official docker image of Redis.

Run the Redis server by the **redis-server** command. You can also check the Redis version here. We can see that the Redis version

is **6.0.9** and it's running on the default port It can be customized by making changes in the Redis configuration:



The screenshot shows a terminal window with the following text output:

```
x ➤ master ➤ redis-server
70010:C 11 Jan 2021 12:41:07.725 # o000o000o000o Redis is starting o000o000o000o
70010:C 11 Jan 2021 12:41:07.725 # Redis version=6.0.9, bits=64, commit=00000000, modified=0, pid=70010, just started
70010:C 11 Jan 2021 12:41:07.725 # Warning: no config file specified, using the default config. In order to specify a config file use redis-server /path/to/redis.conf
70010:M 11 Jan 2021 12:41:07.726 * Increased maximum number of open files to 10032 (it was originally set to 256).

Redis 6.0.9 (00000000/0) 64 bit
Running in standalone mode
Port: 6379
PID: 70010

http://redis.io

70010:M 11 Jan 2021 12:41:07.730 # Server initialized
70010:M 11 Jan 2021 12:41:07.730 * Ready to accept connections
```

Figure 7.15: Run Redis server

Run the following command **redis-cli ping** to verify the Redis server running and health status. Redis will respond with a **PONG** message if it's running healthy:



The screenshot shows a terminal window with the following text output:

```
- ➤ master ➤ redis-cli ping
PONG
- ➤ master ➤
```

Figure 7.16: Check Redis server status

Now, try to hit the **/catalogue** API multiple times using Mock tests or REST clients like Chrome browser's **Postman Runner** tool, which is capable of creating multiple threads and hitting APIs. If you exceed the number of allowed requests, the Spring Cloud Gateway returns response with code **HTTP 429 Too Many Requests** and will not process the incoming request:

```
1. $curl -v
2. * Trying
3. * TCP_NODELAY set
4. * Connected to localhost port 8080
5. > GET /catalogue

6. > Host:
7. > User-Agent:
8. > Accept: */*
9. >
10. < Too Many Requests
11. < X-RateLimit-Remaining: 0
12. < X-RateLimit-Requested-Tokens: 1
13. < X-RateLimit-Burst-Capacity: 4
14. < X-RateLimit-Replenish-Rate: 2
15. < content-length: 0
```

which shows you the number of requests you may send in the next second.

Best practices of API security

Here are some important API security practices:

API authentication & It's very important that all exposed REST APIs should be secured and every client call should be authenticated first and authorized before forwarding to any server endpoint. It could be implemented using the secret JWT token using the Spring Cloud Gateway, OAuth2, and Spring Security libraries.

Expose API over HTTPS Expose all APIs through the HTTPS secure protocol over valid SSL/TLS certificates, for example:

GET https://amazon.com/orders?page=1&page_size=10

Do not expose database models to It's very important to not expose your database schema to the external world. It can be compromised by hackers.

Do not expose the unique database ID to Avoid exposing the actual unique ID to the external world; mainly when you are using it publicly. Anyone can guess the ID of others and misuse it. You can return a different GUID which should be dynamically generated for the given client request.

Avoid SQL Handle your query parameters to avoid SQL injection. It's a code injection which can destroy databases also by passing the destructing query in the query string of API query parameters. Hackers usually add malicious code on the web page input.

Put APIs behind the It's recommended to always expose the API through the security firewall to provide additional level of security to the external world.

Monitoring It's preferable to monitor traffic of client apps and server-side APIs, their inter-communication tracing logs, memory and CPU usage with monitoring and scaling your infrastructure to cater even for higher load and reduce costing by removing extra hardware, if it's not being utilized properly.

API security with SSO using Spring Security, OAuth2, Keycloak, OpenID, and JWT tokens

Let's do some hands-on coding labs. In this section, we will do a code exercise and implement SSO using the Spring Cloud Gateway, OAuth2, OpenID, and Keycloak using the JWT security token.

Spring Cloud Gateway OAuth2 support is a major feature of microservices security. We can also integrate social login like Google, Facebook, LinkedIn, and so on using OAuth2 and Keycloak.

Spring Security added OAuth2 support for WebFlux starting with the 5.1.x GA. OAuth2 login configuration for Webflux is similar to the one for a standard Web MVC application.

We will create two Spring Boot microservices projects which will be deployed on separate Kubernetes containers and can be scaled to multiple instances:

SSO API gateway which will authenticate the client/user and forward authenticated request to another microservice **spring-gateway-oauth2-keycloak-server** which will host the client REST API with actual business logic. This microservice will work as an API gateway filter between client requests and REST API service

providers. It will run on the **Netty** server to support reactive processing.

It will hold actual business logic and work as a REST API provider. In this case, it will return client scope values. It will run on the Tomcat app server.

The following diagram depicts the A&A process flow step by step between the authorization server and resource server:

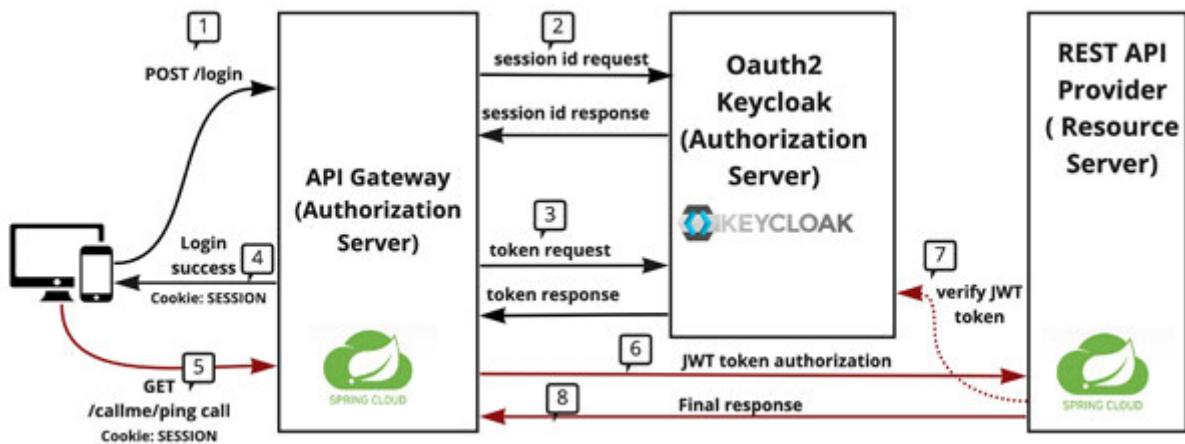


Figure 7.17: SSO architecture between auth server and resource server

Step-by-step A&A process details:

The client will initiate login by API.

The API gateway sends session ID requests and gets responses from the Keycloak **Single Sign-On** authorization server.

The API gateway sends token requests and gets the same from the authorization server.

Clients will get a login success response from the authorization server.

Clients will send the same token in the request header of the REST API call to the API gateway server.

The API gateway forwards the request to the resource server after JWT authorization.

The resource server validates the JWT access token.

Finally, the resource server sends the final response after JWT authorization.

Prerequisite

These are basic installation requirements to build a Spring Boot microservices project where we will implement SSO security:

Spring Boot v2.4.1+

Spring Cloud Gateway v2.4.1+

Keycloak v12.0.1+ server -

Java v8.x+

Docker Desktop with CLI

Source code reference is as follows:

<https://github.com/rajivmca2004/spring-gateway-oauth2-keycloak>

<https://github.com/rajivmca2004/spring-gateway-oauth2-keycloak-server>

Here are the Keycloak installation steps:

Keycloak installation and First, we need to install and set up the Keycloak open source to implement SSO using OpenID and JWT security token technologies.

Keycloak is an open-source **Identity and Access Management** solution for modern applications. It provides SSO, identity brokering and social login, user federation, client adapters, an admin console, and an account management console. You can refer to the official page for the latest version and different deployment instructions:

<https://www.keycloak.org/getting-started/getting-started-docker>

We have used Docker container image to deploy Keycloak and its admin can simply run the following command to deploy the Keycloak Docker image, which will deploy the Keycloak server and web admin console app with default credentials By default, Keycloak exposes the API and a web console on port

```
$ docker run -p -e KEYCLOAK_USER=admin -e  
KEYCLOAK_PASSWORD=admin
```

Log in to the Keycloak web console by using the same user credential and select the **Administration Console** option:

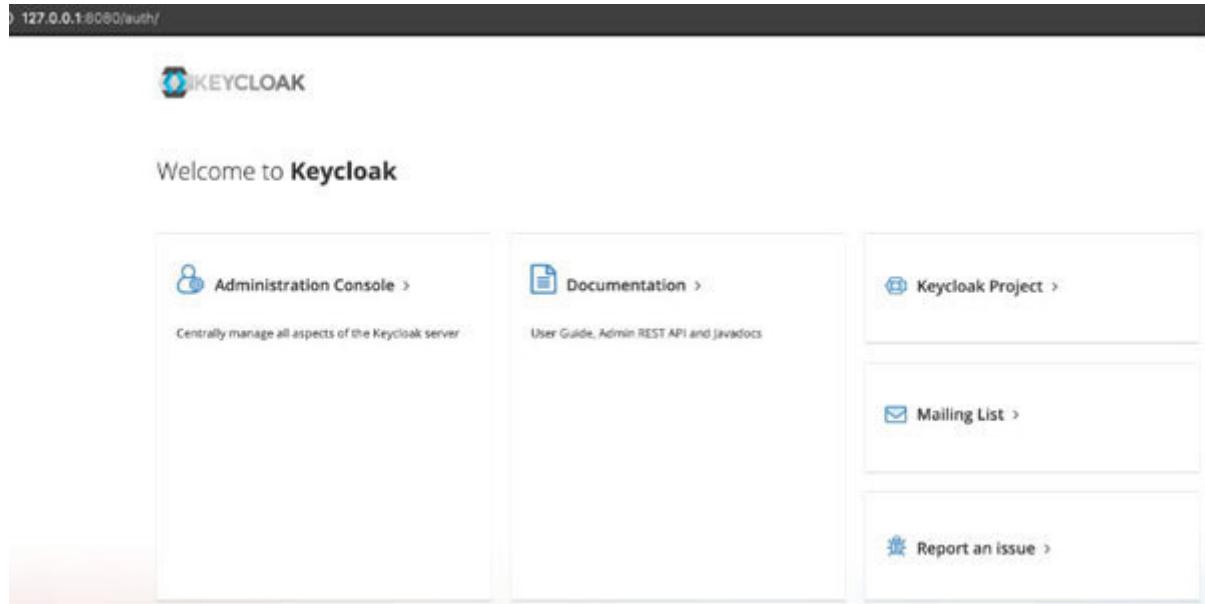


Figure 7.18: Keycloak admin dashboard UI

Create two clients. Both of them should have the **confidential** value **Access Type** section, a valid redirect URI set. In this case, we are just using a simple wildcard for the redirection:

We will assign test roles for this client later.

Default access without the test role:

The screenshot shows the Keycloak administration interface. The left sidebar is titled 'Master' and contains sections for 'Configure' (Realm Settings, Clients, Client Scopes, Roles, Identity Providers, User Federation, Authentication), 'Manage' (Groups, Users, Sessions, Events, Import, Export), and a search bar. The 'Clients' section is currently selected. The main content area is titled 'User-with-test-scope' and shows the 'Settings' tab selected. The configuration fields include:

- Client ID: user-with-test-scope
- Name: (empty)
- Description: (empty)
- Enabled: ON
- Always Display in Console: OFF
- Consent Required: OFF
- Login Theme: (empty dropdown)
- Client Protocol: openid-connect
- Access Type: confidential
- Standard Flow Enabled: ON
- Implicit Flow Enabled: OFF
- Direct Access Grants Enabled: ON
- Service Accounts Enabled: OFF
- Authorization Enabled: OFF
- Root URL: (empty)
- Valid Redirect URLs: * (with a plus sign icon)
- Base URL: (empty)
- Admin URL: (empty)

Figure 7.19: Keycloak client configuration page

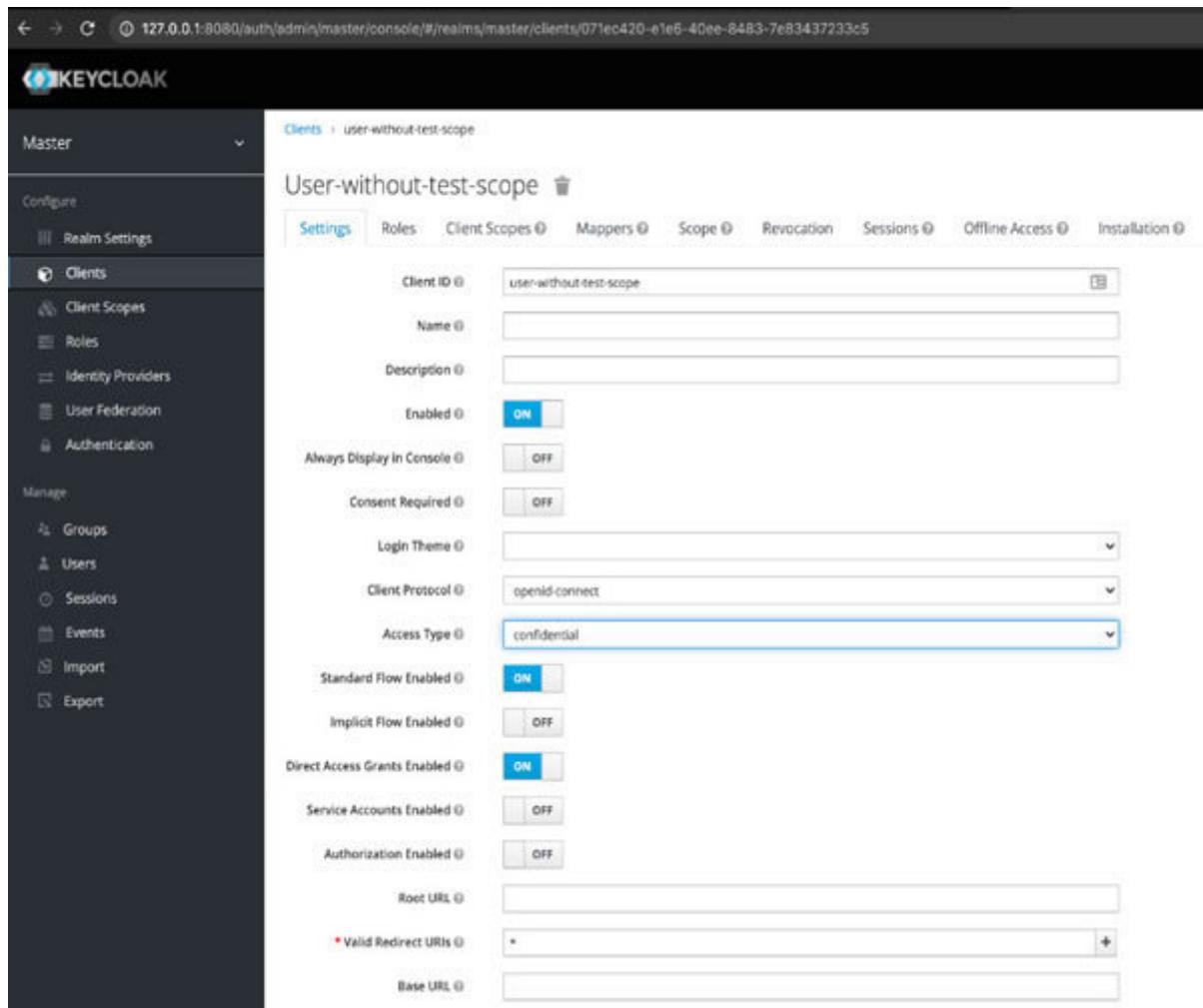


Figure 7.20: Create a new client using Keycloak

These two new clients will be shown on the Keycloak admin console:

The screenshot shows the Keycloak administration interface at the URL `127.0.0.1:8080/auth/admin/master/console/#/realm/master/clients`. The left sidebar is titled 'Master' and contains sections for 'Configure' (Realm Settings, Clients, Client Scopes, Roles, Identity Providers, User Federation, Authentication) and 'Manage' (Groups, Users, Sessions, Events, Import, Export). The 'Clients' section is selected. The main content area is titled 'Clients' and shows a table with the following data:

Client ID	Enabled	Base URL	Actions
account	True	http://127.0.0.1:8080/auth/realms/master/account/	Edit Export Delete
account-console	True	http://127.0.0.1:8080/auth/realms/master/account-console/	Edit Export Delete
admin-cl	True	Not defined	Edit Export Delete
broker	True	Not defined	Edit Export Delete
master-walm	True	Not defined	Edit Export Delete
security-admin-console	True	http://127.0.0.1:8080/auth/admin/master/console/	Edit Export Delete
user-with-role-scope	True	Not defined	Edit Export Delete
user-without-test-scope	True	Not defined	Edit Export Delete

Figure 7.21: Client's status page

Create a default client scope **TEST** for this use case for testing purpose:

The screenshot shows the 'Add client scope' page in the Keycloak administration interface. The left sidebar is identical to Figure 7.21. The main content area is titled 'Client Scopes > Add client scope' and shows the 'Add client scope' form. The form fields are as follows:

Name *	TEST
Description	[empty]
Protocol	openid-connect
Display On Consent Screen	ON
Consent Screen Text	[empty]
Include In Token Scope	ON
GUI order	[empty]

At the bottom right are 'Save' and 'Cancel' buttons.

Figure 7.22: Add Keycloak client scope page

Assign this **TEST** role to the **user-with-test-scope** client only:

The screenshot shows the Keycloak interface for managing client scopes. On the left, a sidebar menu includes 'Master', 'Configure' (with 'Realm Settings' and 'Clients' selected), 'Client Scopes', 'Roles', 'Identity Providers', 'User Federation', and 'Authentication'. Under 'Manage', there are 'Groups', 'Users', 'Sessions', 'Events', 'Import', and 'Export'. The main content area is titled 'User-with-test-scope' and shows the 'Client Scopes' tab selected. It has tabs for 'Settings', 'Roles', 'Client Scopes' (selected), 'Mappers', 'Scope', 'Revocation', 'Sessions', 'Offline Access', and 'Installation'. Below the tabs are two sections: 'Default Client Scopes' and 'Optional Client Scopes', each with an 'Available Client Scopes' list and an 'Add selected' button. To the right, under 'Assigned Default Client Scopes', are 'email', 'profile', 'roles', 'TEST', and 'web Origins'. Below that, under 'Assigned Optional Client Scopes', are 'address', 'microprofile-jwt', 'offline-access', and 'phone'.

Figure 7.23: Keycloak add role page

Now, it's the right time to create Spring Boot microservices:

Create the **spring-gateway-oauth2-keycloak** project using the Spring Initializr web portal:

<https://start.spring.io/>

Add dependencies of **OAuth2 Spring Spring** and



Project Maven Project Gradle Project

Language Java Kotlin Groovy

Spring Boot 2.5.0 (SNAPSHOT) 2.4.2 (SNAPSHOT) 2.4.1 2.3.8 (SNAPSHOT) 2.3.7

Project Metadata

Group com.online.store.demo

Artifact spring-gateway-oauth2-keycloak

Name spring-gateway-oauth2-keycloak

Description Demo project for Spring Cloud Gateway with OAuth2 and Keycloak integration.

Package name com.online.store.demo.spring-gateway-oauth2-keycloak

Packaging Jar War Docker

Java 15 11 8

Dependencies

Gateway SPRING CLOUD ROUTING
Provides a simple, yet effective way to route to APIs and provide cross cutting concerns to them such as security, monitoring/metrics, and resiliency.

OAuth2 Client SECURITY
Spring Boot integration for Spring Security's OAuth2/OpenID Connect client features.

Spring Security SECURITY
Highly customizable authentication and access-control framework for Spring applications.

Figure 7.24: Spring Boot initial project setup

Configure OAuth2-related properties in `application.properties`. Here, we will first define the Spring Cloud Gateway property **TokenRelay** to **default-filters** and **routes** to the remote `GET /callme` API which is hosted on another microservice **spring-gateway-oauth2-keycloak-server** on localhost port 8080. We will add a filter to remove the request header cookie; it tells the gateway to remove the users `Cookie` header from the request during the routing operation because downstream services don't need this; all they need is the JWT.

Next, we need to add Keycloak OAuth2 configuration for both the clients. Most of the configurations are default values. We need to provide the Keycloak server address and port where it's running. Get client-secret of **user-with-test-scope** from the **Credentials** tab under the **Clients** menu:

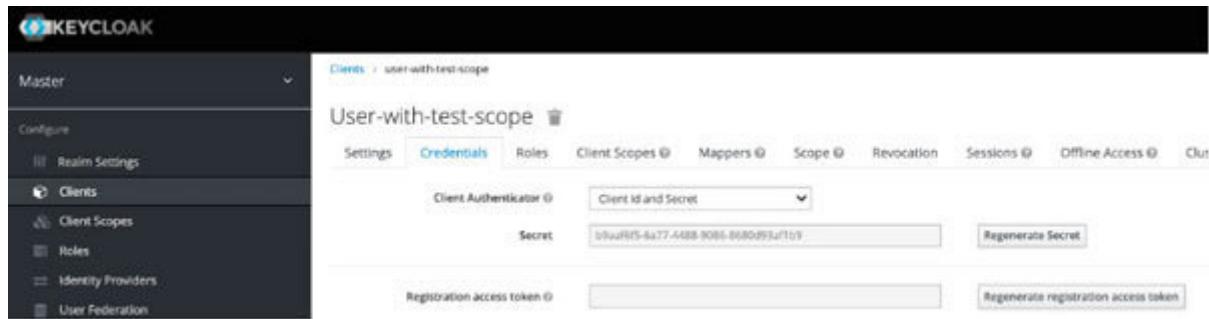


Figure 7.25: Clients credential page

spring:

application:

name: spring-gateway-oauth2-keycloak

cloud:

gateway:

- TokenRelay

routes:

- id: callme-service

uri:

predicates:

- Path=/callme/**

filters:

- RemoveRequestHeader=Cookie

security:

oauth2:

client:

provider:

keycloak:

token-uri:

authorization-uri:

user-info-uri:

user-name-attribute: preferred_username

registration:

keycloak-with-test-scope:

provider: keycloak

client-id: user-with-test-scope

client-secret:

authorization-grant-type: authorization_code

redirect-uri: "{baseUrl}/login/oauth2/code/keycloak"

keycloak-without-test-scope:

provider: keycloak

client-id: user-without-test-scope

client-secret:

authorization-grant-type: authorization_code

redirect-uri: "{baseUrl}/login/oauth2/code/keycloak"

server:

port : 9020

Add the Spring security configuration class with the **@EnableWebFluxSecurity** annotation for OAuth2 with default values. Here, we can disable the CSRF security filter:

```
1. @EnableWebFluxSecurity
2. SecurityConfig {
3.
4. @Bean
5. public SecurityWebFilterChain springSecurityFilterChain
(ServerHttpSecurity http) {
6.         http.authorizeExchange(exchanges -> exchanges.
anyExchange().authenticated())
7.                 .oauth2Login(withDefaults());
8.         http.csrf().disable();
9.         return http.build();
10.    }
11. }
```

Create a controller class with two REST APIs:

GET It will return the JWT access token to the client.

GET It will return the session ID.

```
1. @RestController
2. ClientController {
3.
```

```
4.     Logger LOGGER =
5.
6.     @GetMapping(value =
7.         public Mono getHome authorizedClient) {
8.             API
9.             return Mono.just(authorizedClient.getAccessToken
() getTokenValue());
10.        }
11.
12.
13.        public Mono index(WebSession session) {
14.            / Mono
15.            return Mono.just(session.getId());
16.        }
17. }
```

Now, create another **spring-gateway-oauth2-keycloak-server** microservice which will host business logic-specific REST endpoints APIs. Create this Spring Boot project using Spring Initializr:

<https://start.spring.io/>

Add dependencies of **Spring Spring OAuth2 Resource Spring** and



Project

Maven Project Gradle Project

Language

Java Kotlin Groovy

Spring Boot

2.5.0 (SNAPSHOT) 2.4.2 (SNAPSHOT) 2.4.1
 2.3.8 (SNAPSHOT) 2.3.7

Project Metadata

Group: com.online.store.demo

Artifact: spring-gateway-oauth2-keycloak-server

Name: spring-gateway-oauth2-keycloak-server

Description: Demo project for Spring Cloud Gateway with OAuth2 and Keycloak integration.

Package name: com.online.store.demo.spring-gateway-oauth2-keycloak

Packaging: Jar War

Java: 15 11 8

Dependencies

Spring Web WEB
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Spring Security SECURITY
Highly customizable authentication and access-control framework for Spring applications.

OAuth2 Resource Server SECURITY
Spring Boot integration for Spring Security's OAuth2 resource server features.

ADD DEPENDENCIES... ⌘ + B

Figure 7.26: Spring Boot initial project setup

Add the application and OAuth2 Keycloak configuration in the **application.yaml** file. Here, mainly we define the OAuth2 resource server JWT authentication provider URI, which will be the Keycloak server's default **master** realm:

spring:

application:

```
name: spring-gateway-oauth2-keycloak-server
```

security:

oauth2:

```
resourceserver:
```

```
    jwt:
```

```
        issuer-uri:
```

```
server:
```

```
    port : 9021
```

Add the Spring security configuration with the following security annotations and OAuth2 JWT. It means all incoming requests to the **GET /callme** REST API must be authenticated with the JWT token:

```
@Configuration
```

```
@EnableWebSecurity
```

```
@EnableGlobalMethodSecurity(prePostEnabled =
```

```
SecurityConfig extends WebSecurityConfigurerAdapter {
```

```
    configure(HttpSecurity http) throws Exception {
```

```
    http.authorizeRequests(authorize ->
        authorize.anyRequest().authenticated())
        .oauth2ResourceServer(OAuth2ResourceServerConfigurer::jwt);
    }
}
```

Create a controller and expose the **GET /callme** REST API. It will authorize client requests through the Keycloak client who has a **TEST** role and allow it to access:

- 2.
3. DemoController {
- 4.

5. Logger LOGGER =
- 6.
- 7.
- 8.
9. public String ping() {
- 10.
11. SecurityContext context =
SecurityContextHolder.getContext();
12. Authentication authentication =
context.getAuthentication();
: " + authentication.getAuthorities();
14. }

15. }

Let's verify this SSO. Let's verify this SSO. Login to the **spring-gateway-oauth2-keycloak** Spring gateway authentication microservice app on It will give two client options to log in which we have configured in Keycloak:

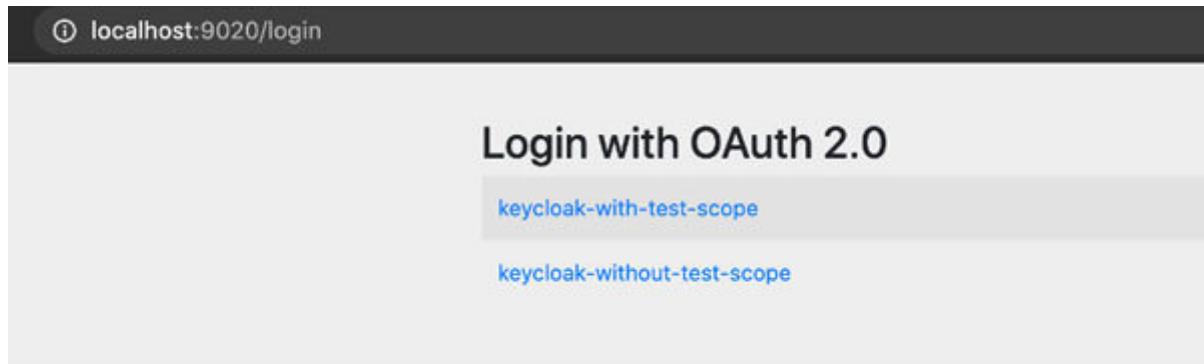


Figure 7.27: SSO login page of Spring Boot app

This **GET /login** API will forward the request to the Keycloak server login page, where you can give the same credentials which you have provided initially by default:

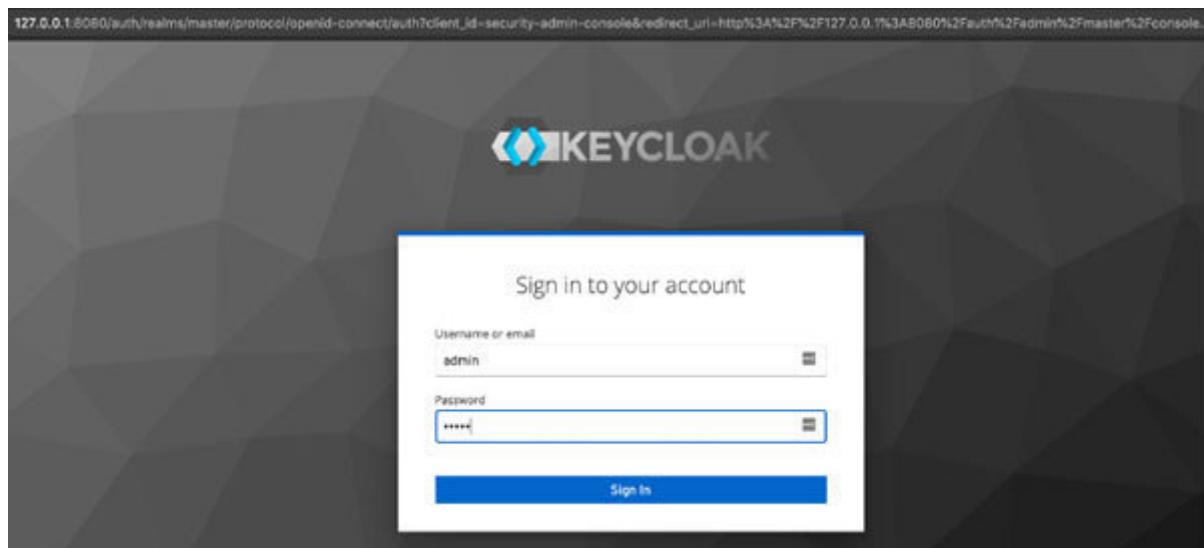


Figure 7.28: SSO page redirection to Keycloak server

It will return the session ID on your browser as the response.

Hit **GET /callme/ping** with the same session ID, which will return all the available scopes from the Keycloak server where we added a custom scope

```
$curl -H "Cookie: SESSION=f4ba6eco-30f5-4212-a458-oeaddofacef7"
```

Scopes: [SCOPE_profile, SCOPE_email, SCOPE_TEST]

Now, hit the **/token** API of the auth server microservice by adding this session ID. It will return the JWT secure token, which will be used by clients to access other APIs of the provider microservice:

```
curl -H "Cookie: SESSION=f4ba6eco-30f5-4212-a458-oeaddofacef7"
```

```
eyJhbGciOiJSUzI1NilsInR5cClgOiAiSldeiwia2lkli  
A6ICJPTjZ2WWptVXNIZkhDZmJ3dGRrdmRPeVFiUEF  
pUnJZNnB3WVM2RE5RQ2NVIno.eyJleHAiOjE2MTAxMjk5MDUsImhlhdC  
I6MTYx  
MDEyOTgoNSwiYXVoaF9oaW1ljoxNjEwMTI5MjkzLCJqdGkiOiIxNGY  
1ZTA3Yy1hNzIxLTRmYjEtOTBjMyozZmQoMWM4MDE1YjliLCJpc3Mi  
OijodHRwOi8vbG9jYWxob3NoOjgwODAvYXVoaC9yZWFSbXMvbWFzd  
G VylowiYXVkljbpbIm1hc3Rlc1yZWFSbSIsImFjY291bnQiXSwick3Vi  
IjoiNGQwYjJiZjktYzk1YyooYWEE1LWEwNWQtYWNhYjdjNDM5NmQ2liwi
```

dHlwIjoiQmVhcmVylowiYXpwljoidXNlci13aXRoLXRlc3
Qtc2NvcGUILCJzZXNzaW9uX3NoYXR
IljoiOTc2YWlzYWltM2NkNSooMTQwLTg1ZTYtNDBiMTJlMGFkN
jUoliwiYWNYljoimSIsInJlYWxtX2FjY2Vzcyl6eyJyb2xlcyl6WyJ
jcmVhdGUtcmVhbGoiLCJvZmZsaW5lX2FjY2VzcylsImFkbWluIwid
W1hX2F1dGhvcmI6YXRpb24iXXosInJlc291cmNIx2FjY2Vzcyl6ey
JtYXNoZXItcmVhbGoiOnsicm9sZXMiOlsidmldy1pZGVudGloes
1wcm92aWRlcnMiLCJ2aWV3LXJlYWxtIiwibWFuYWdlLWlkZW5oaXR
5LXByb3ZpZGVycylsImItcGVyc29uYXRpb24iLCJjcmVhdGUtY2xpZ
W50liwibWFuYWdlLXVzZXJzliwicXVlcnktcmVhbG1zliwidmldy1hdXRo
b3JpemFoaW9uliwicXVlcnktY2xpZW5ocylsInF1ZXJ
5LXVzZXJzliwibWFuYWdlLWV2ZW5ocylsIm1hbmFnZS1yZWFSbSIsInZp
ZXctZXZlbnRzliwidmldy1c2VycylsInZpZXctY2xpZW5ocylsIm1hbmFn
ZS1hdXRob3JpemFoaW9uliwibWFuYWdlLWNsaWVudHMiLCJxdWVye
S1ncm91cHMiXXosImFjY291bnQiOnsicm9sZXMiOlsibWFuYWdlLWF
jY291bnQiLCJtYW5hZ2UtYWNjb3VudC1saW5rcylsInZpZXctcHJ
vZmlsZSJdfXosInNjb3BlljoicHJvZmlsZSURVNUIGVtYWlsliwiZW1haW
xfdmVyaWZpZWQiOmZhBNILCJwcmVmZXJyZWRfdXNlcm5hbWUiOij
hZG 1pbiJ9.PJhpo-
Gl5YDxFWJYaeQHfVgJtMa1EhMomOh4wjo5q6ue6rkEVKONjljp1ep3R6Tj
V6434lIrdn3jArNBXhjE-eQ-
tkJdieh5iOy4452JM1TmhzM29MW9Xe2qZW6N8MqYwvaiuO8cq2uQq
rKXjZpTmozk4glowReWnh6TqMMdU_o2dJmxRWIYiiIqM2L7HsMLxvw
ghRgyd76FlG3Nn4oW7oAN9exv984fPek1Co7_N-
cuIDYD9zxKWsnOMyr3Z34ZbHfoAS-zGhRUcF4_EkACMFvY6sUF-
mJR2Q2oUj
dHcTa3FkW5eEoSzwMhS8Up_XnUTc2UM4QeoYQ4bcQbpCvLvWQ%

You can decode any JWT token on This online portal will show embedded data like user, role, and their authentication detail. Here is the sample API token translated response:

& TOKEN TYPE

```
{  
  "alg": "RS256",  
  "typ": "JWT",  
  "kid": "ON6vYjmUsHfHCfbwtdkvdOyQbPAiRrY6pwYS6DNQCcU"  
}
```

PAYLOAD:DATA

```
{  
  "exp": 1610129905,  
  
  "iat": 1610129845,  
  "auth_time": 1610129293,  
  "jti": "14f5eo7c-a721-4fb1-90c3-3fd41c8015b2",  
  "iss": "http://localhost:8080/auth/realms/master",  
  "aud": [  
    "master-realm",  
    "account"  
  ],  
  "sub": "4dob2bf9-c95c-4aa5-ao5d-acab7c4396d6",  
  "typ": "Bearer",  
  "azp": "user-with-test-scope",  
  "session_state": "976ab3ab-3cd5-4140-85e6-40b12eoad654",  
  "acr": "1",  
  "realm_access": {  
    "roles": [  
      "create-realm",  
      "offline_access",  
      "admin",  
      "uma_authorization"  
    ]  
  },  
},
```

```
"resource_access": {  
    "master-realm": {  
        "roles": [  
            "«view-identity-providers",  
            "«view-realm",  
            "«manage-identity-providers",  
            "«impersonation",  
  
            "«create-client",  
            "«manage-users",  
            "«query-realms",  
            "«view-authorization",  
            "«query-clients",  
            "«query-users",  
            "«manage-events",  
            "«manage-realm",  
            "«view-events",  
            "«view-users",  
            "«view-clients",  
            "«manage-authorization",  
            "«manage-clients",  
            "«query-groups"  
        ]  
    },  
    "account": {  
        "roles": [  
            "«manage-account",  
            "«manage-account-links",  
            "«view-profile"  
        ]  
    },  
},
```

```
"scope": "profile TEST email",
"email_verified": false,
"preferred_username": "admin"
}
```

VERIFY SIGNATURE

```
RSASHA256(
base64UrlEncode(header) + ":" +
base64UrlEncode(payload),
,
)
```

Conclusion

In this chapter, we discussed the API gateway features, advantages, and best practices. We also discussed API security and its best practices. We did hands-on code exercise and implemented how to route API traffic of microservices with the Spring Cloud Gateway, rate limiting, and distributed caching with Redis. We also implemented API A&A security using SSO with the help of OAuth2, Spring Security, Keycloak, and Spring Cloud Gateway libraries.

In the next chapter, we will discuss important aspects of testing distributed microservices to complete end-to-end API testing. We will cover an important aspect of testing with the **Continuous Integration** pipeline. We will also discuss how to mock the REST API to develop and test distributed cloud native microservices.

Points to Remember

JWT is an industry known secure token API authorization mechanism.

Internally, the Spring Cloud Gateway runs on the Netty non-blocking web server which provides asynchronous request processing for faster non-blocking processing of client requests.

Performance can be improved by caching frequently used API response, for example, most frequently searched product metadata, and so on.

All leaders' nodes talk to all followers through the gossip protocol.

Redis is an open-source data structure store to cache data in key-value pairs. It's used as a database, cache, and message broker. We can configure Redis to choose the mode to write and read from. It is recommended to serve writes through the Redis leader and reads through Redis followers.

Cached data must have an eviction policy to remove data from cache after a certain period. Also, cached data should be updated/refreshed periodically, and so on.

When a client sends too many requests, API rate limiting can throttle client connections instead of disconnecting them

immediately. Throttling allows clients still use your services while still protecting the API. There is always a risk of the API request timing out, open connections also raise the risk of DDoS attacks.

Spring Security added OAuth support for WebFlux starting with the 5.1.x GA. OAuth2 login configuration for Webflux is similar to the one for a standard web MVC application.

Key terms

Denial of Services

Distributed Denial of Services

Static Dynamic Random-Access Memory

Java Web Token

Backends for Frontends

OAuth 2.0 is the industry-standard protocol for authorization

References

Spring Cloud

Survey

CHAPTER 8

Microservices Testing and API Mocking

In the modern microservices era, most of the modern apps are developed using the latest cloud native-based microservice architecture. Based on my personal work experience, most of the organizations face challenges to plan a test strategy which can be integrated with **continuous delivery (CI)** pipeline and automate microservices testing. A test strategy is important for those testing teams, who have recently moved from monolithic-based testing to microservices testing style.

The microservices architecture has a ripple effect on development and testing of microservice due to its modular approach and separate deployments. It requires a new modern test plan approach for microservices testing. In this chapter, we will make test managers and engineers life easy by following better test strategy, tools, methodologies, and best practices.

This chapter will also cover important aspects of testing microservices to complete end-to-end testing of business use cases across multiple microservices.

Let's get started!

Structure

In this chapter, we will cover the following topics:

Microservice testing challenges

Microservice testing benefits

Black box versus white box testing

Manual versus automation testing

Microservice testing strategy and testing pyramid

Unit testing

Component testing

Contract testing

Integration testing

End-to-End (E2E) testing

Microservice CI/CD DevOps pipeline

Best practices of microservices testing

Microservices testing types – a quick reference

Functional testing

Non-functional testing

Need for microservices integration testing

Testing microservices integration using **Behavioral Driven Development (BDD)**

Given-when-then test strategy

Advantages of BDD

BDD with Cucumber-pros and cons

Implementing integration testing with BDD and Cucumber

Unit testing versus TDD versus BDD

Testing microservices tools

REST API mocking with WireMock

Need of WireMock

WireMock scope

Mocking existing active APIs (through WireMock recording and playback for new API)

Mocking the new API when the API is not available

Assumptions and limitations

Objectives

After studying this chapter, you should be able to understand the need for microservices integration testing, microservice testing overview and benefits, various testing strategies, and the testing pyramid. This chapter will help you to understand different kinds of functional and non-functional testing, difference between white box vs black box testing and manual vs automation testing. This chapter will help you to design the microservice CI/CD DevOps pipeline for Kubernetes containers and implementation of BDD integration testing using the **Cucumber** tool. It will also help you to refer to the best practices of microservices testing and how to mock REST APIs using

Microservice testing challenges

At present, many organizations have already adopted digital transformation which is based on microservices-based cloud native modern applications development and testing. Testing microservices apps is very challenging for companies, test engineers, test managers, and so on.

These are a few important challenges based on the industry and my personal experience during app development and testing with multiple agile microservices teams:

Microservice architecture-related challenges:

Functional and cyclic dependencies of microservices.

Testing of all microservices is not in parallel. Mostly, they are being developed separately and released phase-wise.

End-to-end integration testing is a nightmare in reality when they are dependent on other microservices and other microservices are not ready to test.

Every microservice will have their own security mechanism and test data.

They all have separate source code and are deployed on separate containers.

They have their own independent databases and database schema.

Hard to find failover of other microservices when they are dependent on each other. Microservices apps are developed in different languages based on use cases, so testing the framework won't work for all the apps.

Business requirement and design change

Frequent changes of business requirements in the Agile development methodology.

Frequent changes of architectural design.

Test database

Databases can have different combinations of SQL/RDBMS and NoSQL like Redis, MongoDB, Cassandra, and so on.

Environment challenges:

Environment challenges like inconsistent test data across all the staging and prod servers for the same microservice apps.

Integration

Integration challenges with third party or external services for different test environments because every environment should be able to access different external staging and PROD services.

Team coordination

Coordination challenges between the development and testing microservices teams to sync with various distributed microservices teams.

Test matrices and tracing challenges such as collecting application logs from distributed microservices log files across different container instances.

API tracing challenges when multiple microservices talk to each other remotely or across the different clusters.

Operational

Sometimes, the operational cost is high because it requires test engineers with different skill sets.

Load testing

Load or performance testing is a major challenge to determine end-to-end integration testing for a feature which has been tested by multiple sets of microservices.

Sometimes, it gets more challenging when microservices are based on async event-driven design, where quick responses will not be calculated. It doesn't give accurate **Transaction Per Second** for end-to-end integration testing.

Microservice testing benefits

Properly tested microservices operate in a consistent and dependable manner on actual production servers, which generate customer trust and more revenue. A software system with no tests, or tests that aren't thorough enough, will suffer from outages that disappoint customers and suffer loss of revenue.

These are some important benefits of microservices testing:

High quality code delivery

Wow, customer experience

Easy maintenance of microservices apps

Low-cost operational cost/OPEX

Better scaling of hardware infrastructure

Reduce production issues

Team optimization with the help of automation testing using a testing framework

End-to-end functional and system testing across all business features and infrastructure

Test individual functionality and with integrated services

Improve performance of apps

Improve development and test engineer productivity

Easy to identify regression changes if any new feature change or fix is applied

Black box versus white box testing

Black box is a testing method when the internal structure or technicality of an application is behind the wall in a black box, so that testing can be done only on the overall system at high level. Testers without any technical background can test apps. The objective of black box testing is to test the expected behavior of microservices apps. It's focused for end users testing, for example, E2E testing, regression testing, and so on.

In **white box** the tester knows the internal structure and source code of the system. It is slower because the test engineer writes test cases and scripts by looking at the internal source code functionality like the unit, component, contract, and integration testing. White box testing focuses on code structure, configurations, business logic, and so on.

Manual versus automation testing

Software testing is a huge domain; it can be broadly categorized into the following two domains:

Manual testing

Automated testing

Let's understand its difference:

difference: difference:

difference: difference: difference: difference: difference: difference: difference: difference: difference: difference: difference: difference: difference: difference: difference: difference: difference: difference: difference: difference:
--

difference: difference: difference:

difference: difference: difference: difference: difference: difference: difference: difference: difference:

Table 8.1: Manual testing vs automated testing

From this, we conclude that automation testing of microservices is the way to go!

Manual testing can be applied on pre-prod servers to test UI-based functional testing.

Microservice testing strategy and testing pyramid

There are many testing strategies for microservices. Here, we will talk about more convenient and systematic processes which we have followed in many projects.

Microservices are built for deployment on small Kubernetes containers in cloud native modern apps methodology. This combination of a microservice architectural and container-based infrastructure requires a systematic testing strategy. A microservice architecture is dependent on multiple dependent microservices. Your test strategy should take care of all the microservices challenges and should be automated with the CI/CD pipeline. Choose your testing techniques with a perspective on time to market, cost, and risk.

There are various testing strategies for microservices. Here, we will discuss popular strategies which are followed in most of the organizations.

The goal of having multiple layers of the following test pyramid is to catch different types of issues at the beginning of testing levels, so that at the end, you will have very few issues. Each type of test focuses on a different layer of the overall software system and verifies the behavior within that scope. For a distributed microservices based cloud native system with a client and backend, the tests can be organized into the following layers:

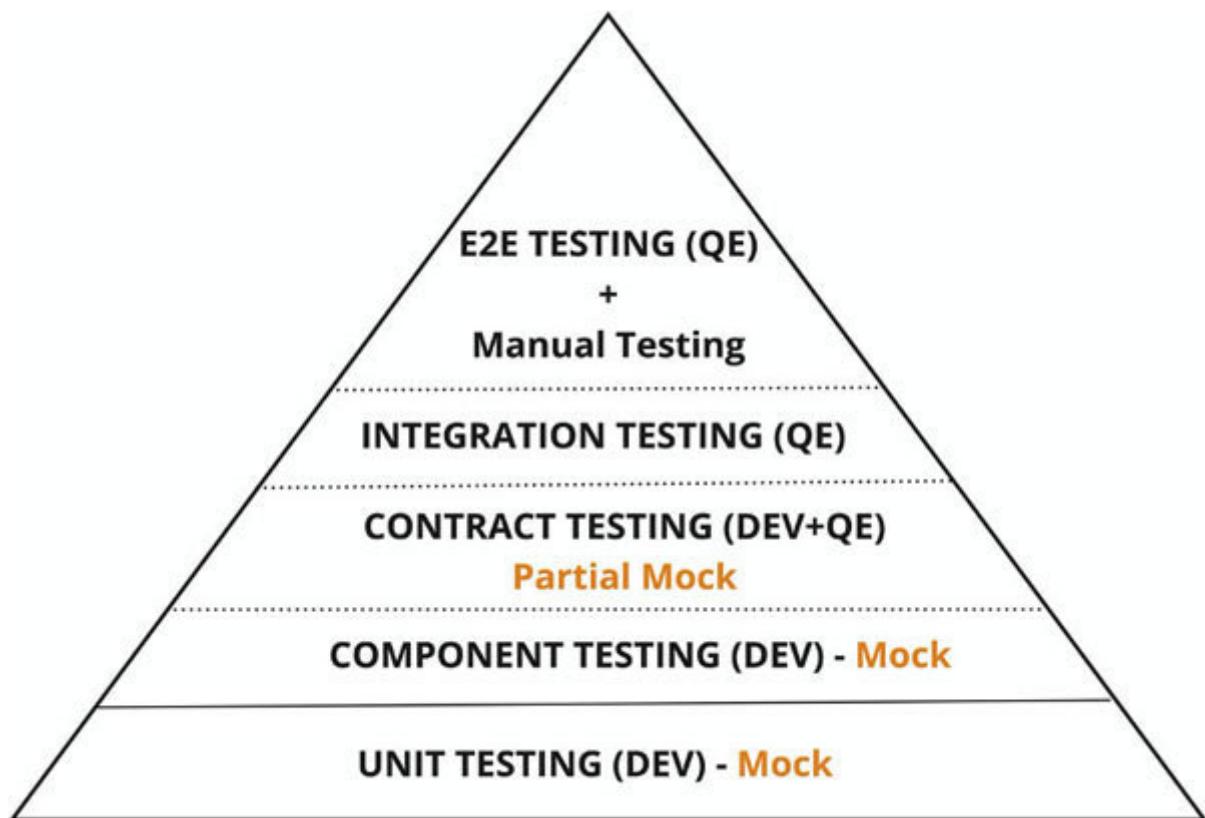


Figure 8.1: Test triangle

It's based on this testing pyramid of five principles:

Unit testing

It's a **level 1** This should be the starting point of the microservices testing pipeline/pyramid. It tests a small unit of functionality and verifies the behavior of source code methods or functions inside a microservice by stubbing and mocking dependent modules and test data.

It's a code level white box testing. Application developers write unit test cases for a small unit of code (independent functions/methods) by using different test data and analyze output independently without impacting other parts of the code.

It is suggested to go with **Test Driven Development** where unit test cases are written first and then make them fail, and write the source code to pass test cases. It's a preferred testing practice adopted by many organizations.

The biggest benefit is that there is no need to write separate unit test cases after writing the source code like a traditional way of testing.

We recommend using the **TestContainers** unit test library for testing lightweight microservices on container platforms.

Testcontainers is a Java library that supports JUnit tests, providing lightweight instances of common databases, Selenium web

browsers, and so on that can run in a Docker container. Refer to the following official document for more details:

<https://www.testcontainers.org/>

Testcontainers make the following kinds of tests:

Data access layer integration It uses a containerized instance of a MySQL, PostgreSQL, Oracle, and other databases to test data access (DAO/DTO) code on lightweight docker containers.

Application integration It runs microservices applications in a short-lived test mode with dependencies such as databases, message queues, or web servers.

UI/acceptance It containerizes web browsers on Docker containers, compatible with Selenium, for conducting automated UI tests. Each test can get a fresh instance of the browser, with no browser state. It does video recording of each test session, or just each session where tests failed for reference.

Component testing

It's a **level 2** It tests a small part of the entire system, for example, testing catalogue microservices of the e-commerce ecosystem. In this component testing, dependent microservices and database responses are mocked or stubbed. In this testing approach, all microservices APIs are tested with multiple sets of test data.

After unit testing, we need to test entire microservices functionalities and APIs independently in isolation. By writing the component test at the granularity of the microservices layer, the API behavior is driven through tests from the client or consumer perspective. Component tests will test the interaction between microservices with the database, messaging queues, external, and third-party outbound services all as one unit.

Contract testing

It's a **level 3** It tests and verifies agreed contracts between microservices. There are contracts defined before development of microservices in the API designing for example, expected response for the given client request or query. If any change happens, then the contract has to be revised.

For example, if any new feature change is deployed, then it must be exposed with a separate version */v2* API request, and we need to make sure that the older */v1* version still supports client requests for backward compatibility.

It tests a small part of integration between the following:

Microservice to its connected databases.

API calls between two microservices

Integration testing

It's a **level 4** It's part of the functional testing phase. It's next level of contract testing where integration testing verifies an entire functionality by testing all related microservices. It covers testing of a bigger part of the system.

For example, login functionality or buying a product from an online e-commerce portal which involves multiple microservices integrations. It tests interactions of microservices API and event-driven components for a given functionality.

We will cover it with an **integration testing automation framework based on BDD** later in this chapter.

It also helps to trace microservices interactions using tracing logs by using **open Spring** and so on:

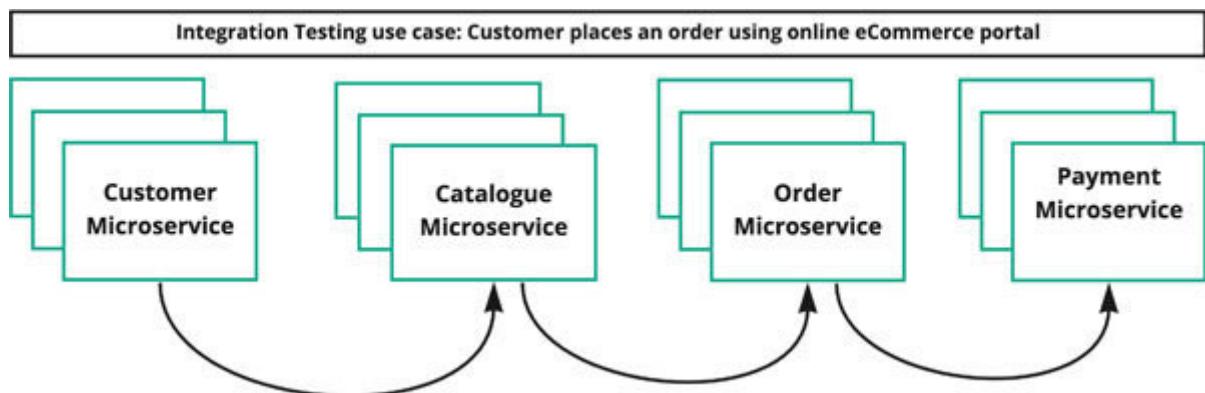


Figure 8.2: Integration testing use case

According to Martin an integration test *exercises communication paths through the subsystem to check for any incorrect assumptions each module has about how to interact with its*

Progression Testing (PT) should also be covered in this phase if we have to test incremental changes on every release.

End-to-End (E2E) testing

It's the final level of end-to-end usability testing like black box testing. E2E testing verifies that the entire system as a whole meets business functional goals from user or client prospective. E2E testing performed on the external front-end **user interface** or APIs client calls with the help of the REST client or CURL, and so on. Also, microservices tracing can be verified and shared with developers for any bugs.

Microservice CI/CD DevOps pipeline on Kubernetes

This is a reference microservices testing pipeline with the CI/CD pipeline:

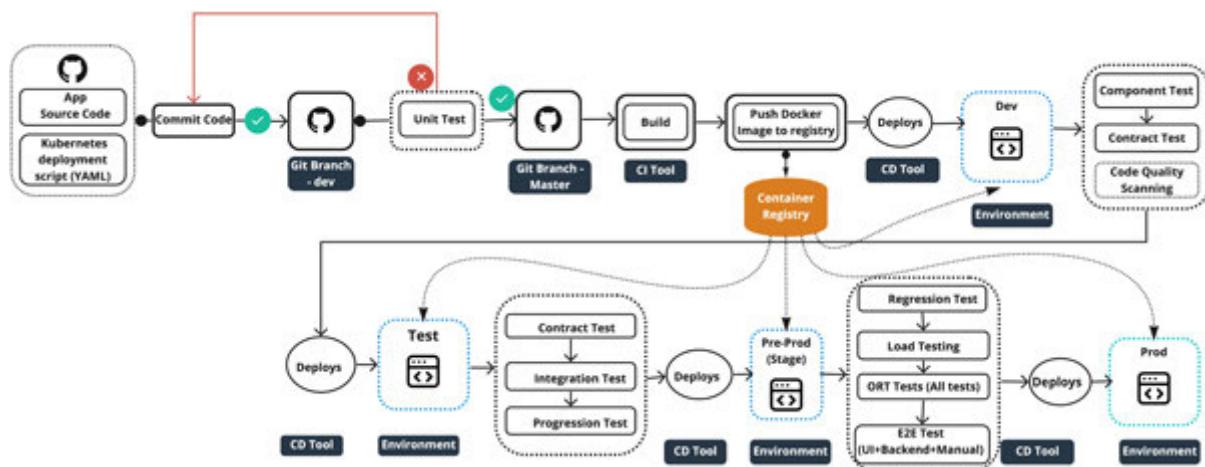


Figure 8.3: Reference CI/CD pipeline for app deployment on container

Best practices of microservices testing

These are a few best practices of microservices testing:

Test the individual method or function as unit testing and always stub dependent method or function test data.

Test smaller units to fail individually and impact any other test cases or functionality.

Prefer chaos engineering to kill apps and databases services and monitor resilient and high availability for platform testing. There are many chaos engineering tools available like **Chaos Mesh**, and so on. Refer to all chaos engineering tools here:

https://en.wikipedia.org/wiki/Chaos_engineering

Integrate testing as part of the CI/CD pipeline preferably with

Integration testing between microservices for end-to-end functionality, which may involve testing of a single feature across testing of multiple microservices.

Always prefer an automated testing framework over manual testing.

Use dummy data for local testing. We would suggest mocking your dependent API response on the mocking server.

Use a common data set across multiple microservices testing teams to avoid any data inconsistency.

Development and testing should go in parallel with agile sprints.

Share application logs with the testing teams to verify and share with the development teams.

Use API tracing for communication logs between microservices.

The testing framework should support all popular browsers and mobile frameworks.

Use in-memory or docker container-based **Testcontainers** databases rather than real databases to test your database dependent functionalities on container. It saves a lot of time and there will not be any dependency on database issues.

Use the test container API to test docker images for that particular microservice CI pipeline.

Testing should be done on a local machine/laptop before going to a shared test server.

Also, test new changes of microservices for backwards compatibility with services already deployed on production.

Use modern application observability, alerting, and monitoring tools like APM tools and so on to analyze test results and issues.

Microservices testing types – a quick reference

There are broadly two categories of testing types:

Functional

Non-functional

Functional testing

These are a few mostly used functional testing types:

Unit Testing It's a modular approach of testing a small unit of functionality independently by providing mock data.

Progression Testing It tests incremental features after every release of code.

Integration Testing It tests dependent microservices for a complete feature like login or get catalogue test use cases. It ensures that each microservice works as expected independently and with the integration of other dependent microservices. It's also called System Integration Testing (SIT).

Smoke/sanity It's a quick testing performed by test engineers on every new released build from developers. It runs a standard routine test case if that build is working fine before deploying to test servers.

It should be done on any new build. It tests and verifies the most important functionality of the system like login and so on. The objective is not to perform exhaustive end-to-end testing, but just to verify that the critical functionality of the system is working fine.

User Acceptance Testing It's the last phase of microservices testing before releasing the app to real end users. It performs functional testing by naive and business users to verify if it meets the real user's expectations.

User Interface Testing It tests and verifies front-end apps like web applications and Android/iOS interface. There are automated UI testing tools available for web and mobile, which records all UI actions and runs it frequently on every change. For example, Selenium is a popular and trusted tool which can be used for web testing and Appium for iOS/Android.

Non-functional testing

These are a few most commonly used functional testing types:

Load or performance testing It evaluates microservices infra responsiveness and threshold for expected client requests load. It verifies and measures **transaction per second** for microservices API interfaces endpoints. This testing decides high availability, non-functional feasibility to scale the environment. There are multiple load testing tools available like and so on.

Operational Readiness Testing (ORT/OAT/OT):

It's a testing strategy which is performed in the final stage during the go-live stage after all testing activities are done. It tests operational readiness of apps before deploying to production. It ensures system and component compliance and smooth system operation in its **Standard Operating Environment**. It's also called **Operational Acceptance Testing (OAT)** and **Operational Testing**.

These are some important types of ORT testing:

Database backup and recovery

App installation and configuration testing

Load and performance test operation

App backup and restore testing

Security testing

Failover testing

Rollback testing

Disaster recovery testing on different data centers and **Availability Zones**

End-to-end test environment operational testing

Operational documentation review

System monitoring and observability testing

Performance testing

Regression testing

Maintainability testing

Component testing: It tests a small part of the entire system, for example, testing microservices of the e-commerce ecosystem. In

this testing, dependent microservices and database responses are mocked or stubbed. In this testing approach, all microservices APIs are tested with multiple sets of test data.

System Testing It tests and verifies a complete integrated system by combining all microservices, databases, environments, and so on to evaluate the compliance of the system with the corresponding requirements.

It validates the complete and fully integrated apps. The purpose of a system test is to evaluate the end-to-end system specifications. It's a black box testing. It's being performed after integration testing. It comes under ORT.

Cross-platform It performs testing across various browsers and hardware devices like how apps are running on different devices like laptop, mobile tab, and so on. Usually, the same microservices based app can be accessed through different devices, client interfaces like web browsers, and various mobile platforms like Android, iOS, and so on.

It also tests different hardware with different screen resolutions and different browsers like Chrome, Firefox, Internet Explorer, and so on.

Contract It tests and verifies agreed contracts between microservices. There are contracts defined before development of microservices. If any changes happen, then the contract has to be revisited and revised.

End-to-End (E2E) E2E testing verifies that the entire system meets business functional goals for the end users or actors.

Regression Testing It verifies that a small code or configuration change in the microservice app doesn't impact the existing functionality.

Need for microservices integration testing

In cloud native patterns, when we have multiple microservices, which are deployed on multiple clusters in a multi-cloud environment, then testing those REST API-based microservices is a nightmare because they are talking to each other synchronously or asynchronously.

Integration testing is a great solution to test use cases end to end by testing integrated and interdependent microservices like login authentication, browse and add an item in the catalogue, end-to-end order placement, and payment on an online e-commerce portal.

Integration test cases are expected to be executed after deployment of microservices on test environments. This phase comes after the development and unit testing phases.

Testing microservices integration using Behavioral Driven Development (BDD)

Behavioral Driven Development is a methodology for developing software through continuous interaction between developers, QAs and BAs within the agile team.

BDD covers integration testing of multiple integrated microservices during test phase. TDD tests a single microservice's features during the development phase.

We can create a separate microservice integration BDD test module to run integration test cases of any microservices and execute it on the QA/test environment. It can be configured and integrated with the DevOps **Continuous Integration** build pipeline using Jenkins, and so on. When the build will be passed after BDD integration on QA, then it should be promoted to the staging/pre-prod environment to make sure that all the REST APIs are intact.

BDD has the following major components:

Feature It contains feature info like scenarios, steps, and examples (test data). It's written in the **Gherkin** language. It's a plain text file with the **.feature** extension.

Every feature can have multiple positive and negative test scenarios: for example, login with the wrong password, login with the correct login credentials, and so on.

Step Every scenario contains a list of steps.

Reference:

Given-When-Then test strategy

BDD is based on the following three major pillars:

Precondition

Test execution

Acceptance and assertions

Reference: Cucumber doesn't technically distinguish between these three kinds of steps.

Given

The purpose of *Given* is to put the system in a known state before the user (or external system) starts interacting with the system (in the *When* steps). Avoid talking about user interaction in If you were creating use cases, *Givens* would be your preconditions.

Setting up initial data

Setting up the initial configuration

Creating model instances

When

The purpose of *When* step is to describe the key action the user performs such as interacting with a web page. It actually calls the business logic or actual APIs.

Then

The purpose of *Then* steps is to observe outcomes like **JUnit**. The observations should be related to the business value/benefit in your feature description. The observations should also be on some kind of output (calculated value, report, user interface, message).

Testing REST APIs or test execution.

Verifying that something related to the *Given + When* is (or is not) in the output.

Checking whether some external system has received the expected message.

Additional And, But keywords

If you have several or you can write as follows:

Scenario: Multiple Givens

Given one thing

Given another thing

Given yet another thing

When I open my eyes

Then I see something

Then I don't see something **else**

Or you can make it read more fluently by writing:

Scenario: Multiple Givens

Given one thing

And another thing

And yet another thing

When I open my eyes

Then I see something

But I don't see something **else**

Advantages of BDD

Here are a few advantages of BDD:

Communication between business and development is extremely focused as a result of a common English type language.

Simplicity, no-technical syntax of features files. Features files are written in plain English, which can be linked with Agile stories.

The code is easier to maintain, flexible, and extendable.

The code is self-documenting with the examples.

Test data can be changed only in the features file, not in the code.

Stories are easier to *groom* – and

There is more visibility into team progress and status using reports. Cucumber reports can be shared with top-level management, integrated with Jenkins, and configured with email notifications. It can also be integrated with automated build and deployment tools like email plugins.

BDD with Cucumber: pros and cons

Cucumber is a very powerful framework for BDD testing. It has many useful features like testing by example (data tables, which can be part of the test cases) or parameters, which can be passed directly from feature file(s). Multiple sets of tests can be sent to BDD test cases. Test data can be passed from the feature files without touching code or making changes in the properties resource files. Features files and the related code looks readable and maintainable. Additionally, Cucumber supports many different languages and platforms like Ruby, Java, or .NET.

Please refer to this comparison reference with other BDD tools

Implementing integration testing with BDD and Cucumber

Let's do some hands-on coding. In this section, we will integrate the testing framework and apply integration testing on microservices. We will add two scenarios for integration testing for pass and fail:

Order microservice, which will call ***catalogue-service*** and ***customer-management-service*** microservices internally. Make sure when you run this integration test suite, both these services are running on a localhost server.

The following diagram depicts the integration of microservices for this test case:

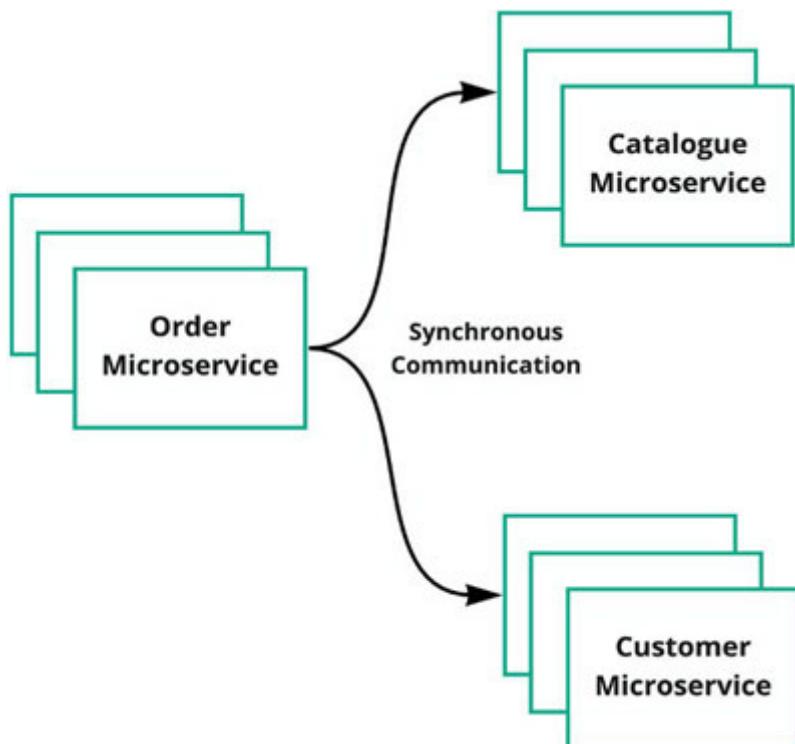


Figure 8.4: Microservices synchronous integration

Create orders using It will be a POST API call and will *FAIL* with **400** errors. It's just a sample for reporting and identifying failed test cases.

Prerequisite

This BDD framework to test any microservices with **Cucumber Java Spring** and so on. These are basic installation requirements:

Knowledge of BDD fundamental, basic Java, and Spring Boot framework

Java 8.x

Spring Boot v2.2.7.RELEASE+ (web and test starter)

Cucumber JVM 1.2.5+

RestAssured v4.3.3+

Google GSON v1.1+

AssertJ

Source code reference is as follows:

<https://github.com/rajivmca2004/bdd-integration-tests-framework>

Here are the steps:

Let's create a Spring Boot project ***bdd-integration-tests-framework*** and add these Spring Boot starters, development tools, Cucumber JVM, and other automation test tools Maven dependencies:

Add the following plugin to run **Cucumber** test cases from the command line CLI (optional):

Add the following plugin to generate **Cucumber BDD HTML reports** with nice tables and graphs:

Add integration test profile in your Maven and add the **CucumberRunner** class which we have already created for this framework:

org.apache.maven.plugins

maven-failsafe-plugin

none

CucumberRunner

integration-test

integration-test

Add **profiles** for different test environments. Here, we have added two **dev** and **default** profiles for simplification:

application-default.properties

application-dev.properties

Add Spring Boot application configuration. We have added the **DEBUG** level for granular logs and activated the **default** profile, which will load the **application-default.properties** configuration:

logging.level.org.springframework: DEBUG

spring.profiles.active=default

Create the **CucumberRunner** class with the required annotations **@RunWith(Cucumber.class)** and **@CucumberOptions** with the following configurations:

Classpath locations of all feature files.

Configure pretty plugin to generate report in HTML and JSON formats. Also, specify the target file location where you want to place the generated HTML/JSON files.

Add the tag for integration test case execution customization. In this scenario, add tilde sign ~ before the `@ignore` annotation, which can be added to features/scenarios to skip/ignore during execution.

Add the package name for Cucumber and other Java classes.

If you want to skip undefined steps from execution, you can specify the `false` value to the `strict` parameter. The default option for `strict` is

Used for dry run of test cases without hitting actual microservices APIs:

```
@CucumberOptions(features = },
```

```
    plugin = },
```

```
    tags = },
```

```
    glue =
```

```
    strict = false
```

```
//,dryRun=true
```

```
)  
  
@ContextConfiguration(classes =  
  
CucumberRunner {  
  
SpringClassRule SPRING_CLASS_RULE = new SpringClassRule();  
  
}  
  
}
```

Create the main Spring boot class and add the environment-specific two profiles – You can add more for **test** and other environments:

```
/**  
  
* This class contains main Spring Boot class  
  
* @author rajiv.srivastava
```

```
*/  
  
@SpringBootApplication  
  
Application {
```

```
Resource[] DEV_PROPERTIES = new ClassPathResource[] {
```

```
    new ClassPathResource(ApiEnum.DEV_ENV.value()) };
```

```
Resource[] LOCAL_DEFAULT_PROPERTIES = new  
ClassPathResource[] {
```

```
    new ClassPathResource(ApiEnum.LOCAL_ENV.value()) };
```

```
main(String[] args) {
```

```
    args);
```

```
}
```

```
/*
```

```
* Environment profiling
```

```
*/
```

```
DevConfig {
```

```
    @Bean
```

```
    PropertySourcesPlaceholderConfigurer
```

```
    propertySourcesPlaceholderConfigurer() {
```

```
        PropertySourcesPlaceholderConfigurer pspc = new  
PropertySourcesPlaceholderConfigurer();  
  
        pspc.setLocations(DEV_PROPERTIES);  
  
    }  
  
    return pspc;  

```

```
ProdConfig {
```

```
    @Bean
```

```
    PropertySourcesPlaceholderConfigurer  
    propertySourcesPlaceholderConfigurer() {
```

```
        PropertySourcesPlaceholderConfigurer pspc = new  
PropertySourcesPlaceholderConfigurer();
```

```
        pspc.setLocations(LOCAL_DEFAULT_PROPERTIES);
```

```
    return pspc;
```

```
}
```

```
}
```

}Add an **enum** to hold all generic key/value microservices APIs endpoints. We can add any app specific properties and access anywhere in the code:

```
ApiEnum {
```

```
    private String value;
```

```
    ApiEnum(String value) {
```

```
        = value;
```

```
}
```

```
    public String value() {
```

```
        return value;
```

```
 }  
  
}
```

Add an **abstract** class for common framework methods which can be inherited and used by any class. We have added a method **buildHeaders()** to add headers with API calls:

```
@ContextConfiguration(classes = })
```

```
BaseTestingStep {
```

```
    HttpHeaders buildHeaders();
```

```
}
```

Create the feature file using the Gherkin script language using **@orderService** which will call **customer-management-service** microservices. You need to comment **@ignore** to execute this test case. If you want to ignore this test case, then uncomment it:

```
@orderService
```

```
Feature: Test Order, Catalogue, customer management  
Microservices Integration Test
```

```
#@ignore
```

Scenario: Fetch order

Given I Set GET order service api endpoint

When fetch order service will be called

Then receive valid HTTP response code **200**

Create the step class for the preceding **@orderService** feature script file. In this class, we will use BDD annotations and **@Then** test steps. We will set up required headers and query parameters in the **@Given** method, call microservices APIs using **RestAssured**, and match actual response/result with the expected one by using the **AssertJ** assertions.

After creating the feature file, we will build this integration project using Java IDE or command line **mvn clean**. It will log skeleton methods for steps files in application logs. Just use it and paste it in step class and write the method definition with the body. We will use the JSON payload in this example:

```
OrderFeatureSteps extends BaseTestingStep {
```

```
    private String ordersHost;
```

```
    Response response;
```

```
String jsonString;  
  
HttpHeaders headers;  
  
/*  
  
 * 1. Scenario: This Order microservice will call both  
catalogue and  
 * customer-management microservice and return aggregated  
result  
 */
```

Set GET order service api

```
i_Set_GET_order_service_api_endpoint() throws Throwable {
```

```
    RequestSpecification request = RestAssured.given();  
  
    RestAssured.baseURI =  
ordersHost.concat(ApiEnum.ORDERS_API.value());
```

```
    request = RestAssured.given();
```

```
    request = request.headers(buildHeaders());
```

```
}
```

order service will be

```
fetch_order_service_will_be_called() throws Throwable {
```

```
    RequestSpecification request = RestAssured.given();
```

```
    response = request.given().get();
```

```
}
```

valid HTTP response code

```
ok) throws Throwable {
```

```
// Added Assert if orders is more than o
```

```
    jsonString = response.asString();
```

```
    ListString>> orders = JsonPath.from(jsonString).get();
```

```
    Assert.assertTrue(orders.size() >
```

```
        assertThat(response.getStatusCode()).isEqualTo(ok);
```

```
}
```

```
@Override

protected HttpHeaders buildHeaders() {

    headers = new HttpHeaders();

    headers.set(HttpHeaders.CONTENT_TYPE,
MediaType.APPLICATION_JSON_VALUE);

    return headers;

}

}
```

Write the **@catalogueService** feature file, which will create a catalogue by calling the **catalogue-service** microservice:

```
@catalogueService
```

```
Feature: Test Catalogue Microservices Integration Test
```

```
#@ignore
```

```
Scenario Outline: Create catalogue in catalogue microservice
```

Given create catalogue data with request body ""

When create catalogues service will be called

Then catalogue created with valid HTTP response code 201

Examples:

```
| body |
```

```
|     src/test/resources/catalogues/catalogues.json |
```

Create the feature file for the **@catalogueService** feature file:

```
CatalogueFeatureSteps extends BaseTestingStep {
```

```
    private String cataloguesHost;
```

```
    Response response;
```

```
    String jsonString;
```

```
    HttpHeaders headers;
```

```
/*
```

* 1. Scenario: This Catalogue microservice will create order

*/

catalogue data with request body

```
create_catalogue_data_with_request_body(String fileBody) throws  
Throwable {
```

```
    RestAssured.baseURI= cataloguesHost;
```

```
    RequestSpecification request = RestAssured.given();
```

```
    file = Files.readAllBytes(Paths.get(fileBody));
```

```
    request =  
    request.contentType(MediaType.JSON).headers(buildHeaders()).body  
(file);
```

```
}
```

service will be

```
create_catalogues_service_will_be_called() throws Throwable {
```

```
    RequestSpecification request = RestAssured.given();
```

```
        response =
request.when().post(ApiEnum.CATALOGUES_API.value());
}


```

created with valid HTTP response code

```
ok) throws Throwable {
```

```
    jsonString = response.asString();
```

```
    System.out.println(jsonString);
```

```
    assertThat(response.getStatusCode()).isEqualTo(ok);
```

```
}
```

@Override

```
protected HttpHeaders buildHeaders() {
```

```
    headers = new HttpHeaders();
```

```
    headers.set(HttpHeaders.CONTENT_TYPE,
MediaType.APPLICATION_JSON_VALUE);
```

```
    return headers;
```

```
}
```

```
}
```

Run integration testing by running the **CucumberRunner** class. It will generate pretty reporting HTML pages:

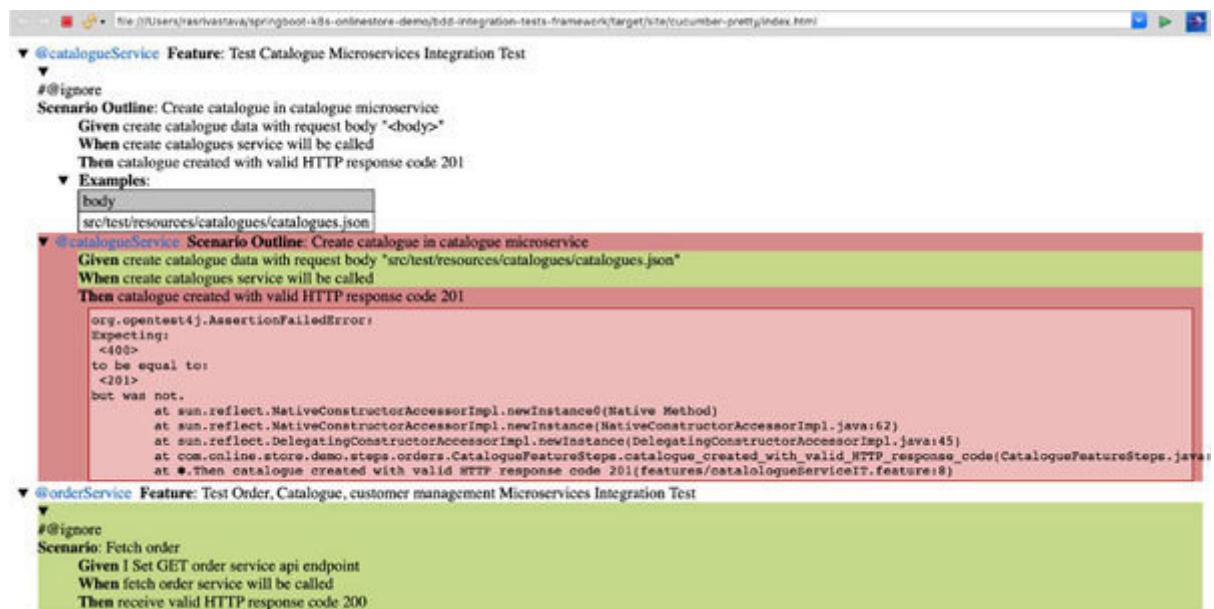


Figure 8.5: BDD Cucumber HTML report

We have added an advanced BDD reporting plugin with Maven. It will create an extensive HTML dashboard with tables and graphs:

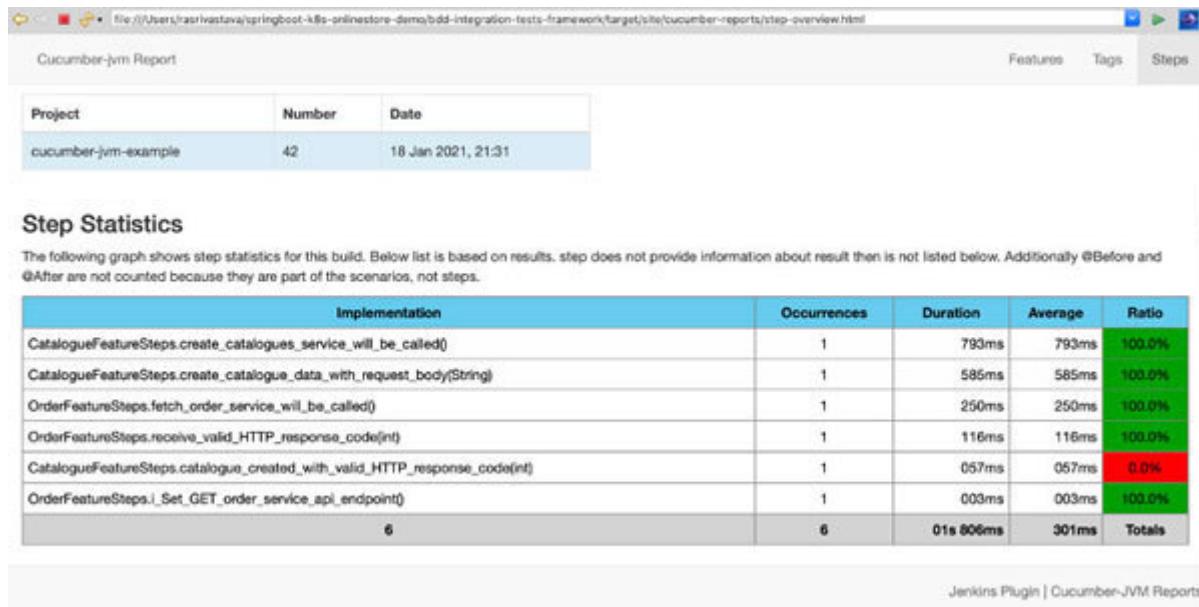


Figure 8.6: Cucumber report for each BDD steps

When you run test cases in the Eclipse-based IDE, it will show the run status report as shown in the following screenshot:

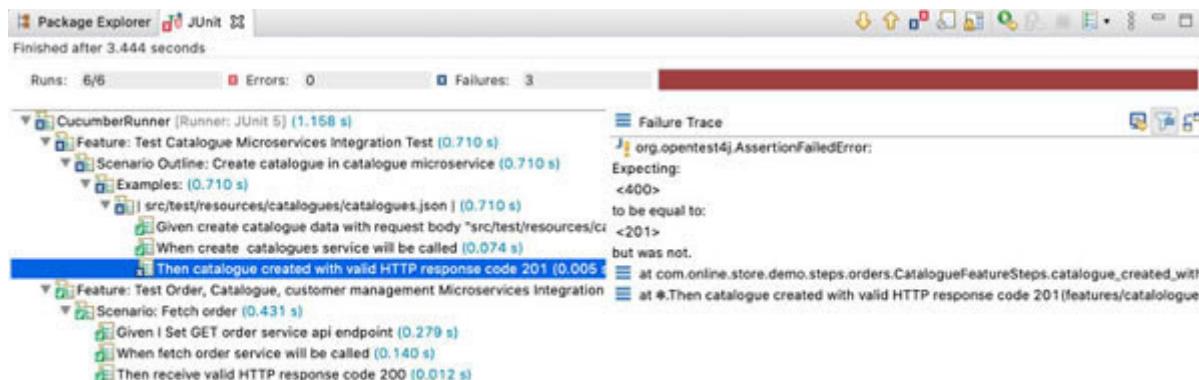


Figure 8.7: Eclipse test report status

Here are the tag reports:

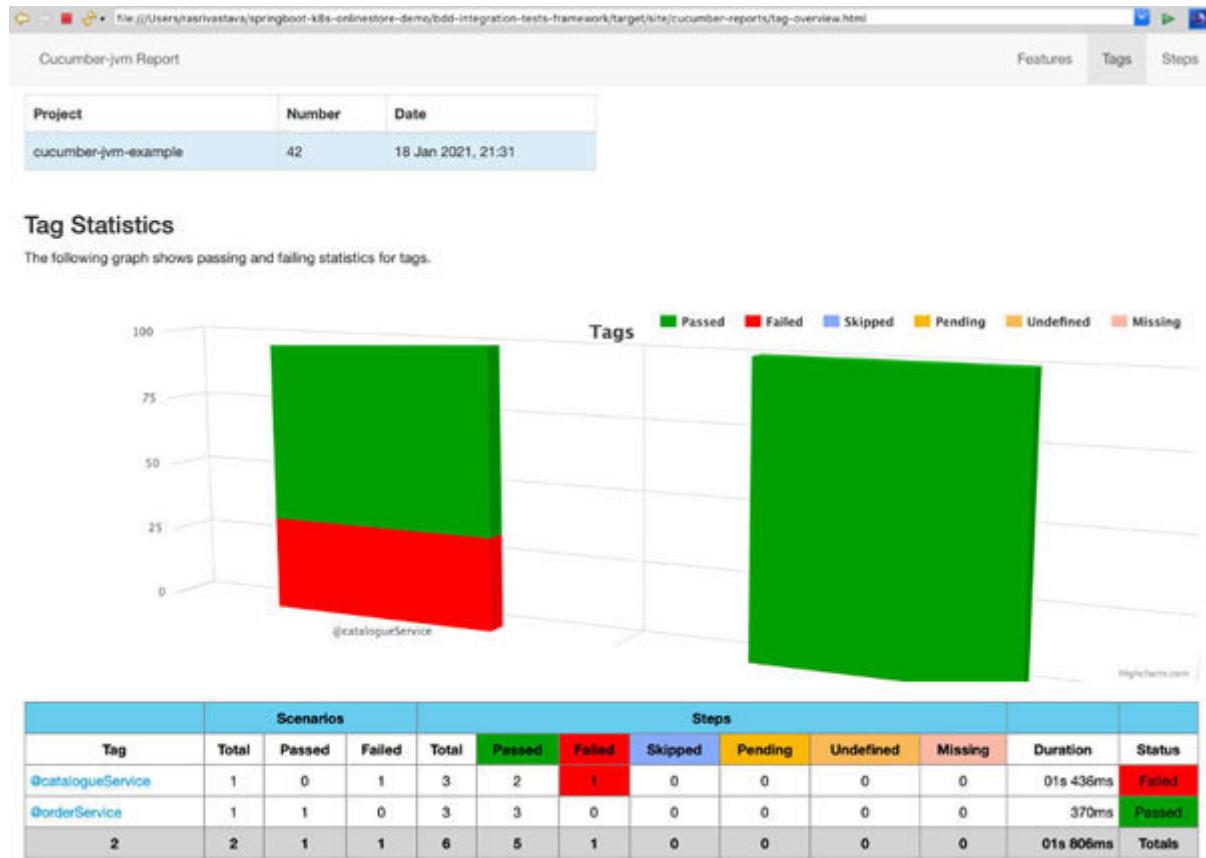


Figure 8.8: Cucumber tag reports

Here is the feature statistics report:

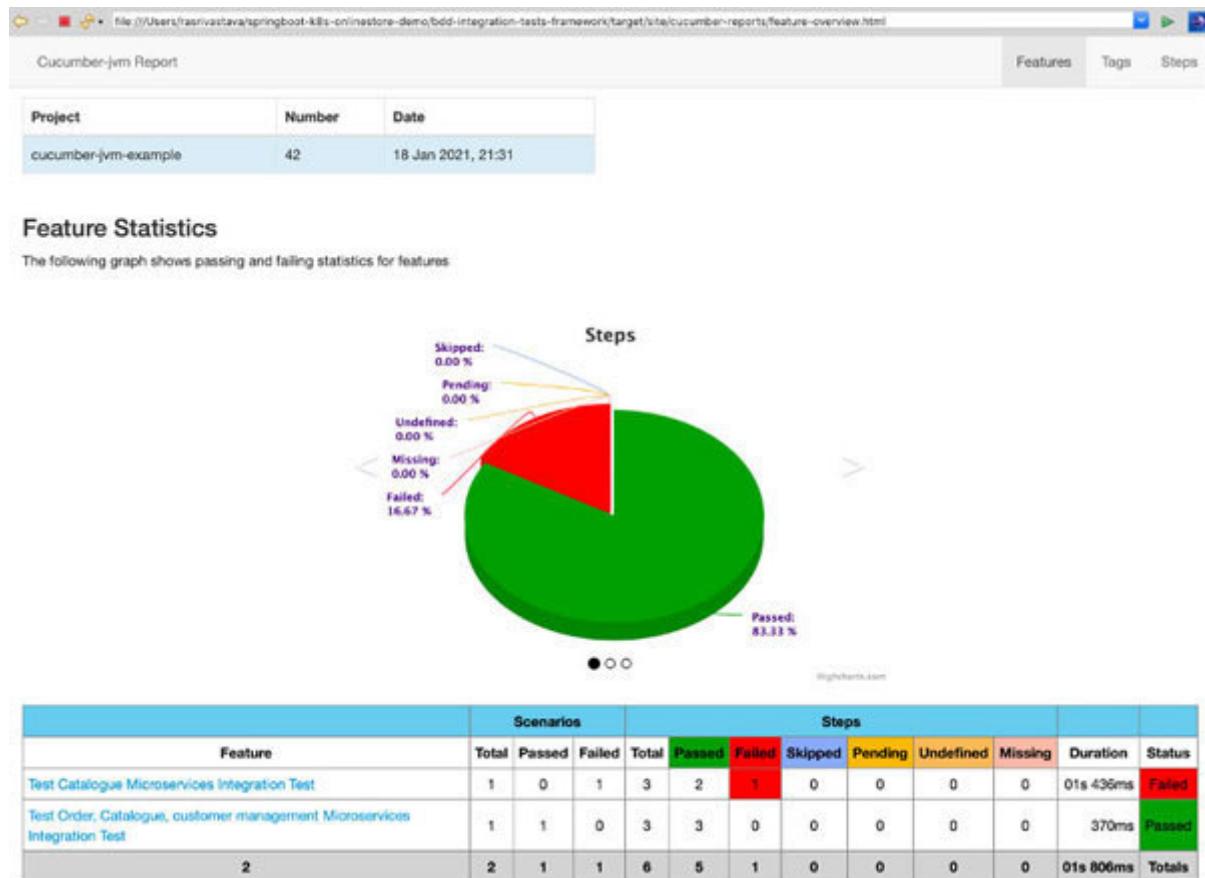


Figure 8.9: Cucumber feature stats report

Cucumber has a feature to group features/scenarios as a group, which is called a It can be annotated in the feature file by using for example, These test suites can be run individually based on your requirements:

```
# Run a Cucumber test suite
```

```
$ mvn clean test -Dcucumber.options="--tags @orderService"
```

```
-Dspring.profiles.active=dev
```

```
# Run multiple Cucumber test suites

$ mvn clean test -Dcucumber.options="--tags @orderService,@catalogueService

# Run a Cucumber test suites and also generate detail report
with

graphs and jar/class files

$ mvn clean install -Dcucumber.options="--tags @orderService "
-Dspring.profiles.active=dev

#Run all test suites and also generate detail report with graphs
and jar/class files

$ mvn clean install
```

**You can use WireMock with BDD-based integration testing to
mock those APIs which are still not available for testing.**

Unit testing versus TDD versus BDD

Unit testing is for testing individual modules, whereas **Test Driven Design** is based on failing test cases first and then writing code to make that pass.

BDD is based on behavioral testing based on real scenarios (which can't be tested in TDD).

Testing microservices tools

There are many tools that can help you with your testing microservices strategy. Some of the best recommended tool options:

Chaos Resiliency tool from Netflix will help you with managing and preventing the consequences of random services instance failure and test the behavior of each service. **Istio** service mesh open source can also be used for fault injection.

JMeter, Locust, Some of the most common performance and load testing tools.

Pact and Spring Cloud To facilitate your contract testing.

Cucumber JVM, AssertJ, Junit, Mocha, Jest, Supertest, or any other popular integration and unit testing tools.

REST API mocking with WireMock

The objective of using WireMock is to use it as a backup mock server in the failover/outage of the actual REST APIs and test them with dummy sample data.

These are major use cases, when we need to mock APIs and test with dummy or cached response data:

After designing and documenting APIs, they should be tested and verified with mock data to ensure API rules and security and governance are applied much before the actual development.

Currently, Development and QA environments are impacted by frequent third-party APIs outages and other service environment issues. It affects Development/QA teams' productivity.

Mocking API server is required, which will sync with the dependent microservices or third-party servers (service providers) and cache the API response periodically on the mock server for Development/QA environment. So, when third-party REST API services are down, then development and testing work won't be impacted on Development/QA servers.

This is a work flow diagram of WireMock:

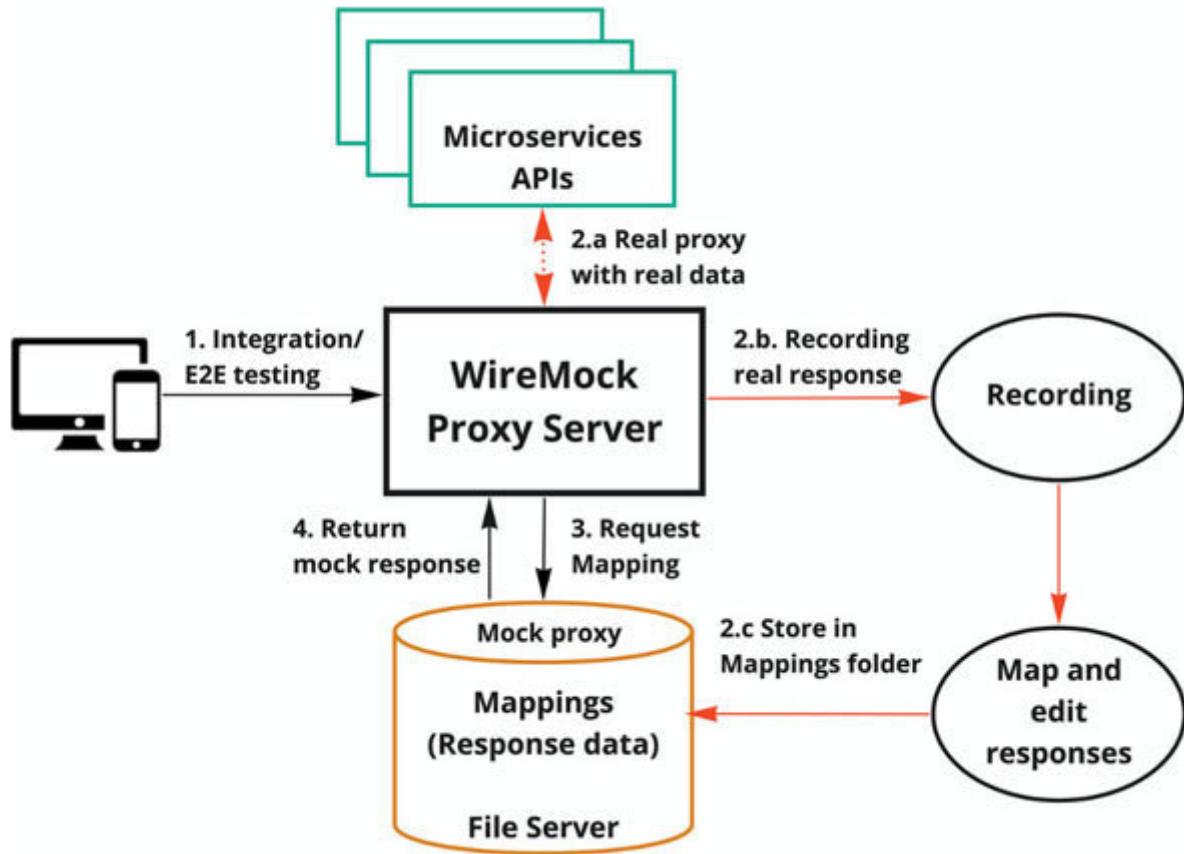


Figure 8.10: WireMock server work-flow

The step-by-step WireMock server workflow is as follows:

Development client initiates API testing. It goes to the **WireMock** proxy server.

WireMock recording and mapping steps:

WireMock hits real microservices APIs when it's not cached at mapping the file server at the WireMock end.

It records real API responses, maps, and edit responses.

It caches API responses in WireMock's mock proxy file server.

In this step, WireMock checks whether the API response is already available with the WireMock file server storage.

File Server returns the mock API response to the client.

Need of WireMock

Here are a few important needs of WireMock:

WireMock is a simulator for HTTP-based APIs, which provides a service virtualization tool or a mock server. It enables you to stay productive when an API you depend on is incomplete or down.

It supports testing of edge cases and failure modes that the real API won't reliably produce.

It's faster and can reduce your build and test time from hours down to minutes.

Real API's requests and responses can be cached and used as a mock server in the absence of the real server using recording features.

It can run on the recording mode by capturing outside third-party traffic to and from an existing running API. It captures REST API responses to the WireMock proxy server and stores them in the **mappings** folder.

It provides a provision to create the requests and corresponding responses in form of JSON request/response objects.

WireMock scope

Here are a few WireMock scopes:

WireMock will be used as a mocking server. The scope of this mocking server is to run it as an independent server or at the developer's local machine for REST API mocking. Also, it should be deployed on the remote non-production and production servers for QA testing.

It will sync up actual third-party servers and record all the tested API requests and responses.

Additionally, mock JSON files can be created for the custom scenarios for the different sets of the request.

The same JSON will be used by web clients internally for local testing during the development phase.

One WireMock instance (JVM) can be only configured for a single API server.

[Mocking existing active APIs \(through WireMock recording and playback for new API\)](#)

It caches all the API responses when it hits the API for the first time, and the next time, WireMock will pick up the response from the local cache where WireMock is deployed and returned to the client. To flush off the WireMock cache and get it updated, you need to use the admin API to refresh the API mapping cache.

Mocking the new API when the API is not available

We will test for the different sets of test data and URI parameters like query and path parameters. The WireMock caches with its unique UUID key will be different for the different request data/query.

WireMock can be run in the following two modes:

Auto-switching Seamless development and testing by pointing to the WireMock server when the main server is up/down with its continuous recording mode feature. In this mode, the client will always point to the WireMock server only, and the WireMock server mocks all the responses. In this case, the client doesn't have to change the URL of the server in their build. In this mode, the recording will be always on, we need to the admin API to refresh the API mapping cache.

Manual switching The client has to change the URL of the server in their build when the main server is down and switching is required for the WireMock server. In this mode, the recording will be always on.

Assumptions and limitations

Only one environment can be configured and mocked with one instance of WireMock, that is, the configured IP address while starting WireMock will be mocked through that instance.

For starting WireMock, we will run the following command so only the API hitting to <http://thirdpartyservices.com> will be mocked. So, if the client is connecting to more than one server, then multiple WireMock instances have to run on the different ports and each WireMock will point to one proxy server:

```
$ -java -jar wiremock-standalone.jar --port 9000 --proxy-all="http://thirdpartyservices.com" --record-mappings
```

Conclusion

In this chapter, we discussed microservice testing overview, benefits, different testing strategies, and testing pyramid. We covered a quick introduction of functional and non-functional testing. We also discussed microservice CI/CD DevOps pipeline practices for Kubernetes deployment. We also covered a detailed overview of integration testing and best practices of microservices testing and how to implement using BDD. We learned an integration testing framework with a lab code exercise. We also covered how to mock REST APIs using WireMock and a few testing tools.

Software and test engineers must develop a testing culture that incorporates automated testing into all aspects of the development and release pipeline. This includes writing unit tests for classes and functions, preferably following TDD, BDD, integration tests for services, end-to-end tests and ORT tests for entire systems, load/performance testing, and UI tests to exercise both the client and the backend together.

In the next chapter, we will cover observability and monitoring techniques of microservices with Spring Micrometer health APIs and Wavefront APM in detail. We will cover the need of APM like performance and telemetry monitoring tools for distributed microservices and integration with Prometheus and Grafana.

Points to Remember

Manual testing can be applied on pre-prod/prod servers to test UI-based functional testing.

Use cloud native modern observability, alerting, and monitoring tools like APM tools Wavefront, Dynatrace, CloudWatch, and so on to analyze test results and issues.

One WireMock instance (JVM) can be only configured for a single API server.

Key terms

Behavioral Driven Development

Test Driven Development

CI/ Continuous Integration/Continuous Delivery

Unit Testing

Progression Testing

Integration Testing

User Acceptance Testing

User Interface Testing

Load/Performance Testing

Operational Readiness Testing

Operational Acceptance Testing

Operational Testing

System Testing

System Integration Testing

End-to-End Testing

Regression Testing

Standard Operating Environment

References

BDD:

Comparison reference with other BDD

CHAPTER 9

Microservices Observability

Observability is a critical requirement of managing any distributed microservices based system. To understand this, we need to compare with traditional monolithic systems, deployed as a single deployable unit, might have different points of integration, failure, and dependencies. Just imagine, complexity of monitoring various microservices in a distributed environment.

Let's try to understand the difference between observability and see how it's different from monitoring, logging, and metrics. Observability determines the behavior of the entire system from the system's output sensors like application logs and metrics. Metrics are not the same as observability. Just think sensors are monitoring tools which collect metrics and logs from the entire application and infrastructure, which are continuously observed by observability tools. These tools do more than just monitoring like analyzing the application and infra failure patterns, logs, and usage, infrastructure resources like memory, CPU, and storage utilization. Observability also provides alerting and self-diagnostic features using advanced algorithms and artificial intelligence technologies. In a nutshell, monitoring tells when something is wrong with the system, while observability observes and finds out *why that wrong happened?* **Monitoring** is a subset of and key action for observability. You can only monitor a system that's observable.

We will cover observability and monitoring techniques of microservices with Spring Micrometer health APIs and Wavefront **Application Performance Management** tools in detail. We will also discuss how to trace multiple microservices in a distributed environment, need of APM and telemetry monitoring tools, and integration with Prometheus and Grafana. This chapter will also cover sample source code examples of these monitoring tools.

Let's get started!

Structure

In this chapter, we will cover the following topics:

Application logging overview

Logging levels

Best practices of logging and tracing

Simple logging using Spring Boot

Log aggregation of microservices

Log aggregation using **Elasticsearch, Fluentd, and Kibana** on the Kubernetes container

Prerequisite

Need for monitoring distributed microservices

API health check using Spring Micrometer

Monitoring apps and infra with the Wavefront APM tool

Microservices debugging using tracing and telemetry

Distributed tracing with Spring Cloud Sleuth

Prerequisite

Microservices monitoring

Monitoring microservices with Prometheus and Grafana

Prerequisite

Objectives

After studying this chapter, you should be able to learn microservices observability, monitoring techniques, best practices, simple logging, and log aggregation using Spring Boot. This chapter will help you to understand monitoring distributed microservices using Spring Boot's and **Grafana** tools and debugging using microservices tracing using **Spring Cloud**

Application logging overview

Logging is a requirement for any cloud native applications. It's one of the strong pillar of the **twelve-factor** application principles. It helps to log all application transactions and helps to identify issues. It's an essential part of application design and development. It makes app monitoring easy and organized.

Every microservice logs its own application logs in their log files or sends to log exporter tools like **EFK** and so on.

Logging is an important need of all applications and brings benefits to developers, operators, and business people. Applications need to capture relevant log data to help us diagnose and fix problems and measure business metrics.

There are various testing frameworks and tools available with Java and Spring-based applications. Spring Boot uses the **Logback** logging API by default with the **SLF4j** logging framework. You can customize it and change to other logging implementations like **Log4j**.

It is recommended that you use SLF4j on any logging tool to have an abstraction layer. It will help to change the underlying logging framework tools.

Logging levels

The following are the important logging levels which we should consider on different production and non-production environments:

Its logs fatal and serious app issues. Your application is about to abort to prevent some kind of corruption. It may terminate the app.

An error is a serious issue and represents the failure of applications like heap memory, third party, and database connection issues.

It warns for unusual situations or unexpected code. It may not terminate the app. It gives a warning to investigate and follow the best code configuration or code quality issues.

It logs normal application behavior which helps to understand the business flow, request/response, and data. It indicates if the service has started or stopped. It's required for developers and test engineers to identify expected results.

It logs more granular low level and diagnostic information. It provides detailed diagnostic information for developers, sysadmins, and so on.

It captures every detail about the app's behavior.

It logs absolutely everything, including any custom logging levels that someone has defined.

It doesn't log anything at all. It's not recommended.

Best practices of logging and tracing

We hope the following best practices of logging and tracing will help in real-microservices projects:

Correlation

Add a correlation ID in all logs and append the microservice app name or ID with the timestamp.

Correlation ID should be added in all REST API requests and responses calls.

- o Add a common and consistent correlation ID for every client request which will track the entire interaction request/response end to end.

Log

Always store microservices logs in a centralized location.

Don't store logs on containers because containers are ephemeral and can be crashed and restarted.

Have a separate **Kubernetes** cluster for monitoring and storing logs of microservices deployed on different K8s clusters.

Some logs can be stored on the local storage. Sometimes, it saves the network cost by sending logs to other central locations. It's based on log frequency.

Use a log appender which automatically pulls app logs from log files of microservices containers and sends them to the central location.

Unique

Add a unique ID in API response to distinguish multiple responses for the same requests.

Async

Archive old logs on a backup server. Enable auto-backup of the log file to a persistent location.

Split log

Split log files in small chunks by configuring the max log file size limit. Logging tools are capable of splitting and creating new log files after reaching that size limit.

Log

Use configuring the log level with a **Command Line Interface** parameter, so that logs can be generated on-demand by running

apps by developers for reproducing issues at the local system.

Set the log level to **INFO** by default to avoid access logging on production servers.

Always disable log levels on production servers because they take huge memory and CPU time. It makes microservices apps slow.

Enable the log only during debugging on any non-prod and prod servers.

Set package level logs for APIs mainly for Spring and Java-based apps to avoid extra logging, which could impact the performance of apps and occupy huge storage.

Log

Filter logs based on actual requirements to avoid performance issues. Like the filter or mask API logs which are not needed for debugging any issue.

Log

Mask sensitive confidential data in logs like personal data or credit card financial information because it may create security issues.

Log

Always add audit logs information for audit and compliance purpose.

Log

Logs should be formatted before persisting. If logs are coming from different kinds of apps and databases, they should have a single format.

You can refine logs from different sources and re-structure them with a common standard format when you store in a common external location.

Prefer to log structured and meaningful log data.

Environment-based

Customize logs for different environments based on the need like a test server should have more logs than production.

Log

Add log tracing wherever needed for the transaction sequence.

Log Externalize all microservices logs from Kubernetes containers to an external central location for log aggregation and analysis.

Persist externalized aggregated logs in search engines like Elasticsearch/Solar for faster search.

Simple logging using Spring Boot

Spring Boot uses **Logback** by default. We need to make the following configuration changes to add the Spring Boot testing framework:

Add the following Spring test dependency in the Maven's **pom.xml** file:

We need to configure app logging configuration in the application properties file. Logging level is **INFO** by default:

`logging.level.org.springframework: DEBUG`

Log aggregation of microservices

There are multiple microservices in cloud native distributed environments. It's easy to log in a single monolithic app. Nowadays, apps are moving from monolith towards microservices, which makes it very tedious and complex to track sequences of logs when multiple services are logging into different microservices apps for the same use cases. For example, ordering a product from an online shopping portal which internally calls multiple microservices and completes an end-to-end (browsing product catalogue to the final payment).

It's very challenging to read logs from all apps and trace the end-to-end business flow.

To overcome this challenge of distributed logging, we need a mechanism to aggregate all logs with a central location and search the entire sequential log flow quickly by using log aggregation tools and visualize in log monitoring tools like and other APM tools like **AWS CloudWatch**, **GCP Cloud Monitoring**, **VMware CloudHealth**, **VMware vRealize** and so on:

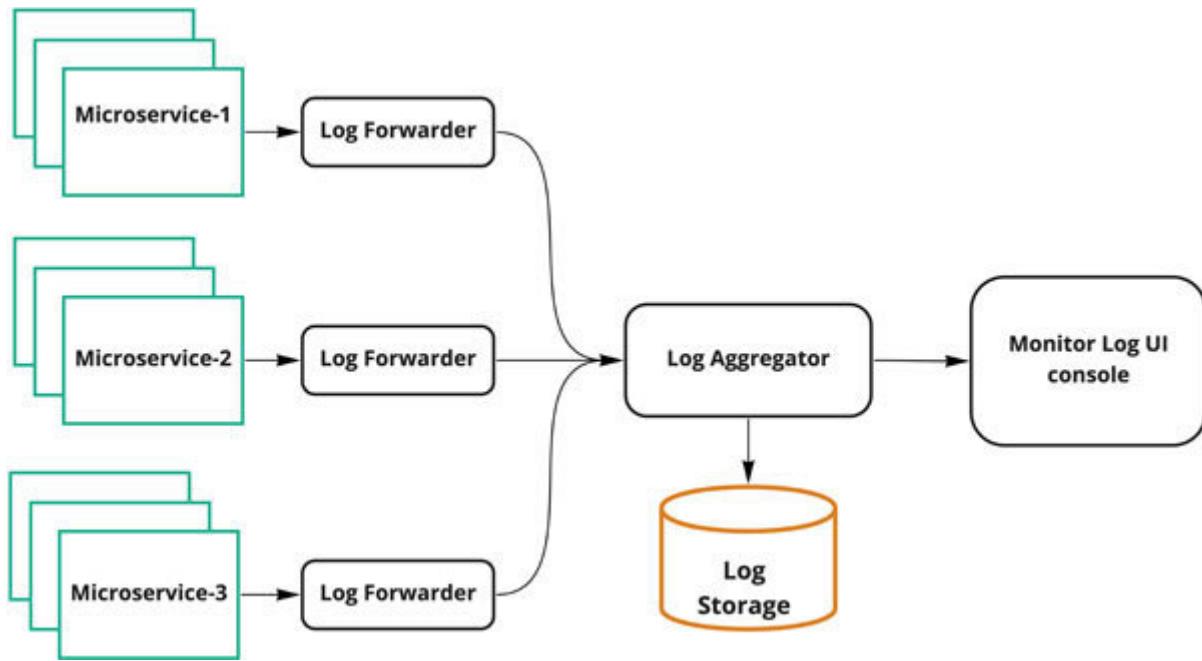


Figure 9.1: Log aggregation

Microservices run on multiple hosts and containers on the Kubernetes container platform. Microservices apps should send all the generated logs across the hosts to an external, centralized place from where it's aggregated.

A single correlation ID can be used across all microservices instructions. In this approach, a correlation ID is generated when the first microservice call is initiated and passes that same ID to downstream services and logs this ID across all microservice calls. By looking up this correlation ID, entire end-to-end logs can be searched and used for diagnostic and other purposes.

There are various log aggregation tools available for log aggregation. Log aggregation is divided in the following three modules:

Aggregated log This module is responsible to pull or push logs to a centralized server or container to aggregate logs. and Splunk log forwarder are popular log aggregator tools.

Aggregated log search: This module is responsible to store logs and search quickly. It supports advanced-level queries also, where logs can be searched quickly based on the correlation ID and other log messages. It's a storage or database where all logs are stored and searched. In the modern microservices apps development, Elasticsearch and Splunk search engine is a good choice where logs are stored in the in-memory database and persistence both, and search quickly. Search engines store these application logs based on the **Lucene** file search mechanism, which provides faster search on metadata.

Log This module fetches logs from the log storage location and visualize on a web portal with logging and monitoring web dashboard and graphs. Kibana and Splunk are very popular visualization tools.

Log aggregation using EFK on the Kubernetes container

The EFK stack usually refers to **Elasticsearch**, **Fluentd**, and **Let's explore what are the EFK stacks:**

It's a NoSQL database or data structure based on the Lucene search engine. It's a search engine for fast querying log data. It stores data/metadata and searches much faster than file systems and databases. It can ingest large volumes of data, store it efficiently, and execute queries quickly.

It's a great choice for large volumes of data. In the EFK stack, Elasticsearch etc. is used for log storage and receives log data from the **Fluentd** log forwarder. The log data is stored in an Elasticsearch index in-memory and is queried by the Kibana visualization tool. The Elasticsearch storage in Kubernetes container's **Persistent Volumes** stores the indexed log data. In on-premise control planes, the data is stored on the physical storage.

Fluentd/Fluent Fluentd is a flexible log data collector. It supports various inputs like log files or syslogs and supports many outputs such as Elasticsearch. It's an open-source log forwarder to forward logs from app servers/containers to target Elasticsearch or a similar database or file system. It has a variety of connectors to forward to different storage options. Fluent Bit is lighter and faster. Fluent Bit can read the Kubernetes container or Docker log

files from the file system or through the **Systemd** journal, enrich logs with Kubernetes metadata, and forward data to Elasticsearch or any other database. We will deploy Fluent Bit as a daemon set to all nodes on Kubernetes clusters. A **DaemonSet** ensures that a certain pod is scheduled to each **kubelet** exactly once. The Fluent Bit pods on each node mount the container logs for the host which gives us access to logs for all containers across the cluster. The Fluentd pod mounts the **/var/lib/containers/host** volume to access the logs of all pods scheduled to that kubelet as well as a host volume for a Fluentd position file.

Kibana provides a visual web dashboard. It provides a query dashboard for querying the logs in Elasticsearch. We can run an instance of Kibana in each control plane and we need them all to be kept in sync with the same base configuration. This includes setting the index pattern in Kibana.

Fluentd is a log collector, processor, and aggregator. Fluent Bit is a lightweight subset of the Fluentd log collector and processor. It doesn't have strong aggregation features such as Fluentd. You can use Fluentd for advance level aggregation and configuration.

In this section, we will implement log aggregation using the EFK technology stack on the VM. You can also install the EFK stack on Kubernetes:

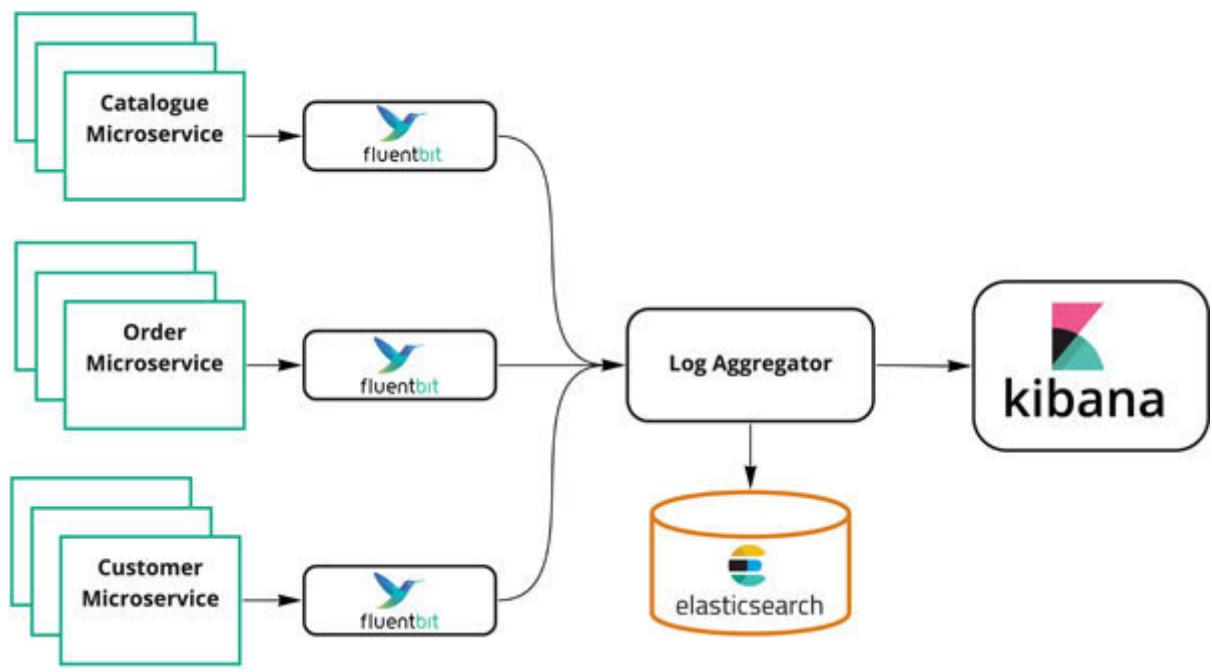


Figure 9.2: Log aggregation using EFK

Prerequisite

The following are basic installation requirements:

Java 8.x

Spring Boot v2.2.7.RELEASE+

Elasticsearch

Fluent Bit

Kibana

Homebrew package installer

Source code reference is as follows:

<https://github.com/rajivmca2004/catalogue-service>

Let's refer to the same Spring Boot project **catalogue-service** which we have already created and referred to in the previous [chapter](#). Let's get started for this lab code exercise:

We will make only a few changes in the same Spring Boot application configuration file to enable logging into files. Fluent Bit or Fluentd open-source log aggregation tools will read the same file from the configuration file, push to Elasticsearch search database and display on Kibana.

Add this logging configuration in the **application.yml** file. In this file, we have added a separate logging section with logging file details like file path and name. Also, set the logging level to different levels like **ERROR** and

Set the logging level to Java's package level. Also ignore Spring's library specific logs, which are not very meaningful for debugging application issues.

Spring by default support the Logback logging API; we can configure to any other logging framework.

We also recommend using the **SLF4j** logging abstraction layer, where backend logging libraries can be changed anytime to **Logback** or **Log4j** and so on:

spring:

application:

name: catalogue-service

jpa:

hibernate:

ddl-auto: update

logging:

file:

name: /Users/rasrivastava/spring-logging/application-backend-dev.log

level:

org.springframework: ERROR

com.online.store.demo: INFO

Now, we need to install and configure the EFK stack. We will first install Fluent Bit, which will read the application logs files which have been generated from **catalogue-service** Spring applications and stored in a given file path.

Install Fluent Bit using this reference link. We have used MacOS installation by downloading the **dmg** file, you can choose based on your operating system:

<https://docs.fluentd.org/installation/install-by-dmg>

We need to install which is another flavour of Fluent Bit with more features. You can refer to the preceding link to see the comparison. In managed cloud Kubernetes like GKE and EKS, we don't need to externally configure Fluent Bit or the ELK stack - container logs sent to **stdout** because they are part of CloudOps. Public cloud providers take care out of the box.

Now, launch **td-agent** by running the following commands:

#Load

```
sudo launchctl load /Library/LaunchDaemons/td-agent.plist
```

#Check recent logs

```
$ tail -f /var/log/td-agent/td-agent.log
```

#Unload

```
launchctl unload /Library/LaunchDaemons/td-agent.plist
```

Elasticsearch is a search engine based on the Lucene file-based search. It stores searchable logs in small tokens or chunks to search quickly. It stores logging data in-memory to search faster. It's a recommended approach to search plenty of logs from different microservices at a centralized place.

Refer to the following installation link to install the latest version of Elasticsearch on MacOS with Homebrew:

Tap the elastic homebrew repository and install Elasticsearch using the **brew** CLI command. It will install the latest version. We need to run the following commands:

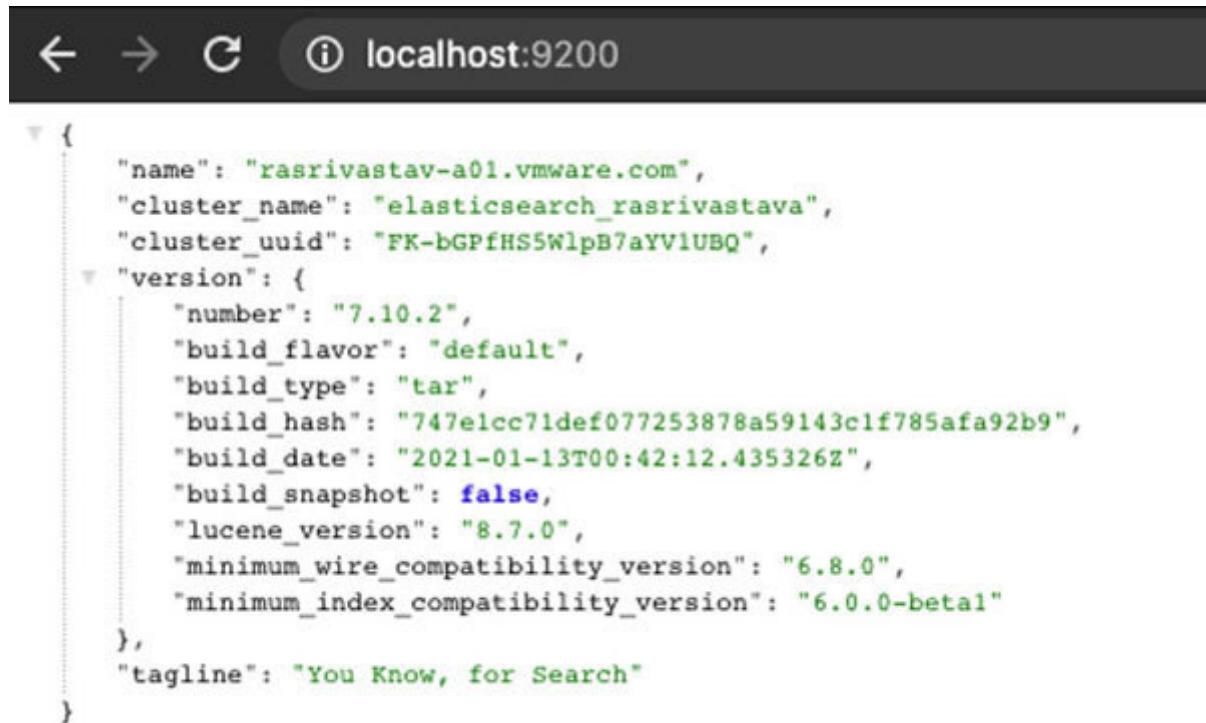
```
$ brew tap elastic/tap
```

```
$ brew install elastic/tap/elasticsearch-full
```

```
#Start
```

```
$ elasticsearch
```

Now, we can access Elasticsearch using the default port **9200**:



```
{
  "name": "rasrivastav-a01.vmware.com",
  "cluster_name": "elasticsearch_rasrivastava",
  "cluster_uuid": "FK-bGPfHS5WlpB7aYV1UBQ",
  "version": {
    "number": "7.10.2",
    "build_flavor": "default",
    "build_type": "tar",
    "build_hash": "747e1cc71def077253878a59143c1f785afa92b9",
    "build_date": "2021-01-13T00:42:12.435326Z",
    "build_snapshot": false,
    "lucene_version": "8.7.0",
    "minimum_wire_compatibility_version": "6.8.0",
    "minimum_index_compatibility_version": "6.0.0-beta1"
  },
  "tagline": "You Know, for Search"
}
```

Figure 9.3: Elasticsearch health check

We can see all Elasticsearch instances by clicking on this URL or using CURL command



```
green open .apm-custom-link 0Qz1fc81QWScT5eeazED8Q 1 0 0 0 208b 208b
green open .kibana_task_manager_1 DXE805x1Skundk4JXaO1Ag 1 0 5 263 143.9kb 143.9kb
green open .apm-agent-configuration BUX2xYkJR-u3oAdnnHbFJg 1 0 0 0 208b 208b
green open .kibana-event-log-7.10.2-000001 iVRteV2_SfODN-qyGrZXWQ 1 0 7 0 38kb 38kb
green open .kibana_1 V1ZhX8hMS52cHiZbdKauDg 1 0 40 17 2.1mb 2.1mb
```

Figure 9.4: Elasticsearch indexes

Configure Fluent Bit to forward Spring Boot Java app logs to Elasticsearch:

```
vi /etc/td-agent/td-agent.conf
```

Add this Fluent Bit configuration in this **td-agent.conf** file:

```
@type tail
```

```
host localhost
```

```
path /Users/rasrivastava/spring-logging/application-backend-dev.log
```

```
pos_file /Users/rasrivastava/spring-logging/application-backend-dev-
multiline.pos
```

```
tag spring-boot-logs
```

```
flush_interval 5s
```

```
@type elasticsearch
```

```
host localhost
```

```
port 9200
```

```
logstash_format true
```

```
logstash_prefix spring-boot-logs
```

```
flush_interval 1s
```

We need to load Fluent Bit after making changes.

Install the latest version of Kibana using the following reference link:

<https://www.elastic.co/guide/en/kibana/current/brew.html>

```
#Install
```

```
$ brew tap elastic/tap
```

```
$ brew install elastic/tap/kibana-full
```

```
#Start
```

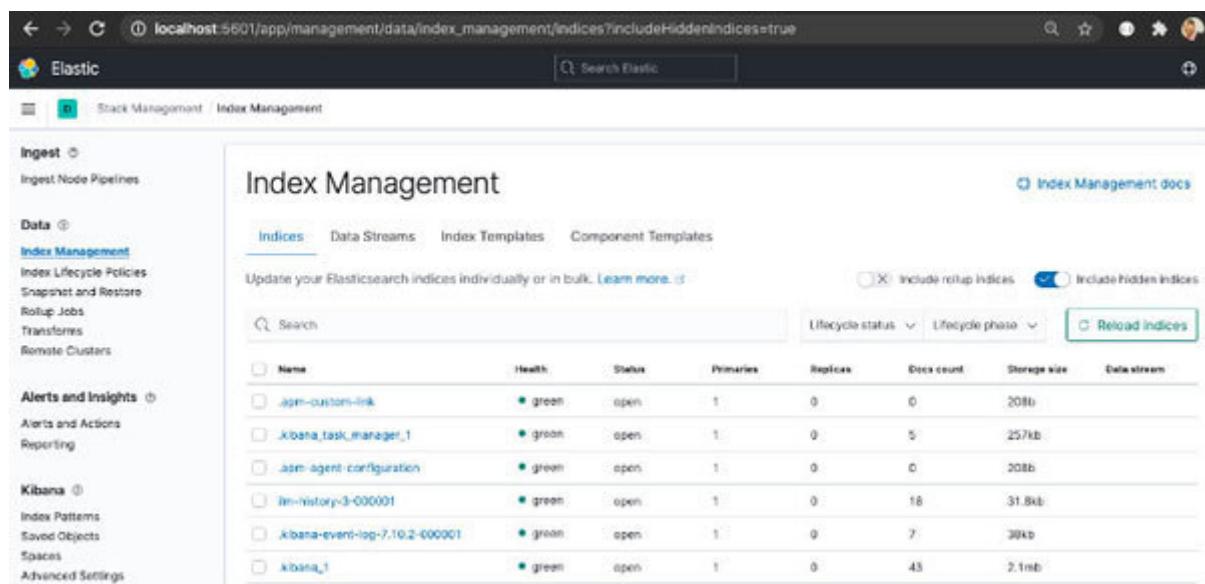
```
$ kibana
```

By default, Kibana connects to the Elasticsearch instance running on the localhost on Elasticsearch port To connect to a different Elasticsearch instance, modify the Elasticsearch host URL in the

`/usr/local/etc/kibana/kibana.yml` configuration file and restart Kibana. You can refer to the following link for more information:

<https://www.elastic.co/guide/en/kibana/6.8/connect-to-elasticsearch.html>

In the final step, we need to run Kibana on its default port **5601** on the browser and add this new Elasticsearch index for visualization on the Kibana dashboard. Configure the Kibana dashboard for different kinds of create logs and metrics reporting visual panels:



The screenshot shows the Elasticsearch management interface at `localhost:5601/app/management/data/index_management/indices?includeHiddenIndices=true`. The left sidebar includes sections for Ingest, Data (with Index Management selected), Alerts and Insights, and Kibana. The main area is titled "Index Management" and contains tabs for Indices, Data Streams, Index Templates, and Component Templates. A search bar and filters for "Lifecycle status" and "Lifecycle phase" are present. A "Reload indices" button is also visible. The table lists several indices with their details:

Name	Health	Status	Primaries	Replicas	Docs count	Storage size	Data stream
apm-custom-link	green	open	1	0	0	208b	
kibana_task_manager_1	green	open	1	0	5	257kb	
asm-agent-configuration	green	open	1	0	0	208b	
lm-history-3-00001	green	open	1	0	18	31.8kb	
kibana-event-log-7.10.2-00001	green	open	1	0	7	38kb	
kibana_1	green	open	1	0	43	2.1mb	

Figure 9.5: Kibana dashboard

This is how we can set up the EFK stack for centralized logging for all our Spring Boot-based microservices and visualize on the Kibana web dashboard.

Need for monitoring distributed microservices

Observability tools monitor the logs in real time to gather important metrics useful for both business and operations. It can also be configured to raise alarms when these metrics exceed specific thresholds. Developers use logs for debugging and tracing and even to capture important events to build and test runs in CI/CD pipelines.

In the modern microservices architecture, there are many microservices apps that talk to each other. It makes it very challenging to track microservices API communications across different K8s containers. Every client request contains request data like query, parameters, and header configuration. Similarly, microservices also generate response data with the API response code like HTTP status code, and so on.

When client requests go high, there are many microservices, which talk to each other to serve these requests. It's a nightmare to track each and every request in log files for any bug. This is a big problem, that is, traceability especially when sync, async, and reactive flows are mixed together. Developers face difficulties by spending a lot of time to track these client request and response traces.

We need tracing tools to solve these practical challenges of microservices apps and developers. There are many API tracing open-source and enterprise tools available which have visual

dashboards and reporting features also such as Opentracing, Sleuth etc. They provide search engine features to store traces logs and search faster.

API health check using the Spring Micrometer

Spring Boot has its internal health check libraries like **Spring** integration with Prometheus and Grafana libraries, and so on.

The Micrometer provides an abstraction level over the instrumentation clients for a number of popular monitoring systems like and **Prometheus** monitoring systems. Spring Boot Actuator provides its own metrics support and does not interact with the micrometer directly. So, the metrics collections happen independently. The Actuator provides metrics like health check and other monitoring details.

We can configure this by adding the following configurations:

management:

endpoint:

health:

enabled: true

show-details: always

endpoints:

web:

exposure:

include: <*>

jmx:

exposure:

include: <*>

This will expose the inbuilt actuator and micrometer APIs which can be used to check the health of microservices apps on Kubernetes containers or VMs. For example: **/health** API will return the app response:

`http://localhost:8080/actuator/health`

```
{  
    "status": "UP",  
    "components": {  
        "db": {  
            "status": "UP",  
            "details": {  
                "database": "H2",  
                "result": 1,  
                "validationQuery": "SELECT 1"  
            }  
        },  
        "diskSpace": {  
            "status": "UP",  
            "details": {  
                "total": 499963174912,  
                "free": 259620347904,  
                "threshold": 10485760  
            }  
        },  
        "ping": {  
            "status": "UP"  
        }  
    }  
}
```

Figure 9.6: Spring Boot actuator health check

You can refer to the following link for more configuration and APIs:

<https://micrometer.io/docs/ref/spring/1.5>

You can also integrate micrometer with prometheus and expose their APIs to Grafana.

[Monitoring apps and infra with the Wavefront APM tool](#)

Wavefront is an APM observability tool, which provides more monitoring features and also provides AIML-based reporting and pro-active monitoring.

The **VMware Tanzu Observability** by the Wavefront platform is mainly built to handle requirements of modern applications and multi-clouds at high scale. It's a unified solution with analytics (including AI) that ingests visualizes and analyses metrics, traces, histograms, and span logs. So, you can resolve incidents faster across cloud applications, which are correlated with the cloud infrastructures views.

Please refer to the link for more information:

Wavefront pushes Spring Boot microservice app logs to the SaaS cloud server using the Wavefront proxy forwarder:

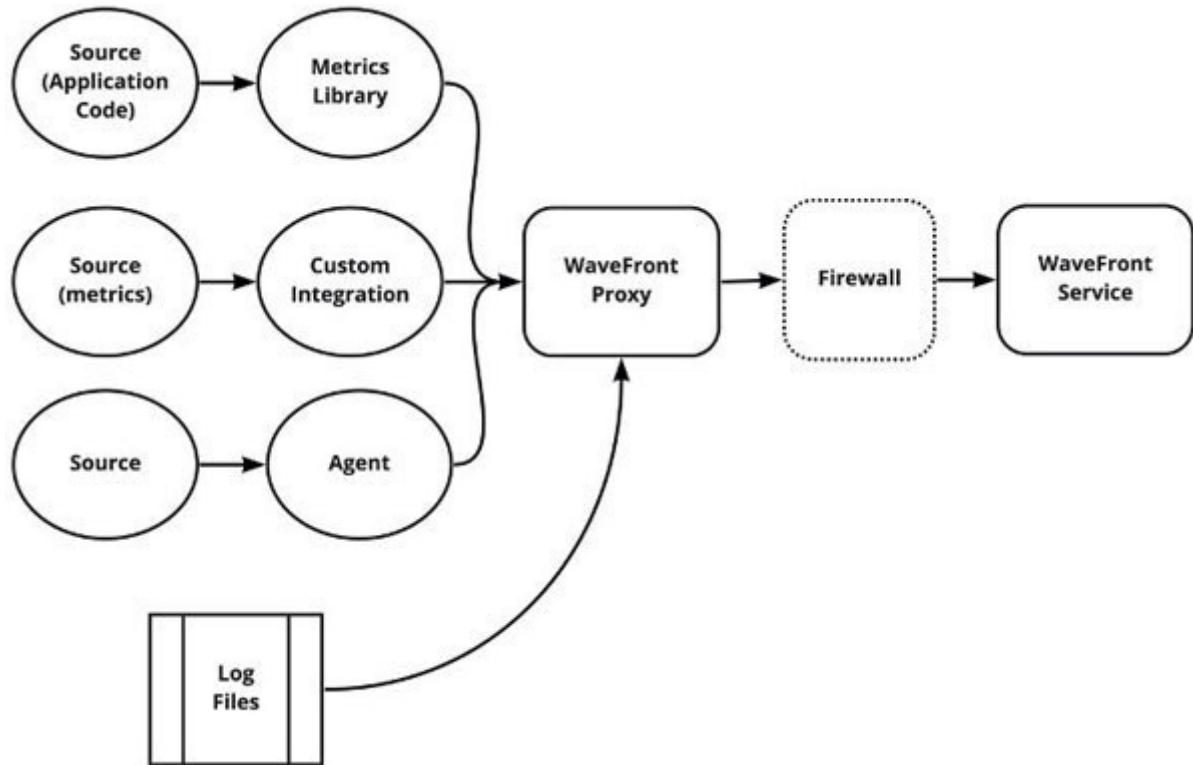


Figure 9.7: Wavefront proxy

Wavefront is secured by design. It only uses the following monitoring data from the organization's on-premise data center/cloud **Availability Zones**

Metrices

Traces and spans

Histogram

There are multiple ways to protect the privacy of data on SaaS cloud when data is transmitted from applications and infrastructure servers to the cloud. It's safe to use!

Spring Boot provides inbuilt integration with Wavefront APM. It configures the Wavefront admin observability console by adding a minimal configuration. Let's follow this step by step to integrate Spring Boot microservice with WaveFront.

We can add this configuration in Spring Boot microservice Maven's configuration **pom.xml** file as follows:

Add the following dependency to send the trace data to Wavefront using **Spring Cloud Sleuth** or Pick either Spring Cloud Sleuth or OpenTracing and use it across all your Spring Boot-based microservices. We will use Spring Cloud Sleuth here:

Add Wavefront Maven dependencies:

Build your source code with the Maven build using the following command:

```
mvn clean install
```

This will add a Wavefront admin console automatically. When we run the Spring Boot microservice app after adding this configuration, the Wavefront URL will be in application logs:

```
2021-02-10 00:18:58.678  INFO 16037 --- [restartedMain]  
c.o.s.demo.CatalogServiceApplication : Started  
CatalogServiceApplication in 7.742 seconds (JVM running for  
8.372)
```

Your existing Wavefront account information has been restored from disk.

To share this account, make sure the following is added to your configuration:

Connect to your Wavefront dashboard using this one-time use link:

<https://wavefront.surf/us/l5pvtdXrf5>

API token for Wavefront SaaS

Wavefront SaaS URL

It returns this information. Add the following configuration to the Spring Boot **application.yml** file:

To quickly test and verify, connect to your Wavefront dashboard using this one-time use link. It can be different for different machines or environments. The following URL is just a reference:

<https://wavefront.surf/us/l5pvtdXrf5>

The screenshot shows the Wavefront web UI dashboard. On the left, there is a sidebar with navigation links for 'Saved Searches', 'Integrations', 'Tag Paths', 'Tags', and 'Updated By'. The main area is titled 'Dashboards' and contains three listed dashboards:

Name	Charts	Favorites	Last Updated	Views	Access
Tour: Anomaly Detection	13	0	05/14/2020	0 today 0 week 0 month	View
Tour: Cluster Metrics Exploration	43	0	05/14/2020	0 today 0 week 0 month	View
Tour: Single Host Health Exploration	37	0	05/14/2020	0 today 0 week 0 month	View

Figure 9.8: Wavefront web UI dashboard

Check out Honeycomb, AppDynamics, Datadog APMs enterprise level observability, and monitoring vendor tools.

Microservices debugging using tracing and telemetry

Tracing of multiple microservices is very essential to identify their interaction during debugging for any production bug. One of the major problems microservices developers encounter when microservice apps grow is its nightmare to identify their interactions and tracing details.

Telemetry is an important pillar of observability of microservices systems. It's the automated communication processes from multiple data monitoring sources. Telemetry data is used to take proactive measures, scale infrastructure, and improve customer experience by increasing the performance of the application and monitoring security, application health, and quality. Modern cloud native apps use telemetry from service mesh technologies like **Istio** or These tools have out-of-the-box capability to trace microservices interaction using side-car proxy on a separate Kubernetes container along with the main application container on the same POD. We will discuss more about the service mesh in the last chapter.

The Spring Cloud Sleuth project solves tracing-related use cases. It provides unique tracing IDs to app logging which are common and consistent between microservice calls. It adds two types of IDs to your logging: one called a **trace ID** and the other called a **span** The span ID represents a basic unit of microservice; for example, sending an HTTP request from the front-end web application to back-end REST APIs. The trace ID contains a set of

span IDs, forming a tree-like structure. The trace ID will remain the same between microservice calls.

It helps developers to find how a single client/user request travels from one microservice to the next. In this way, end-to-end tracing can be captured, analyzed, and visualized on the web console. We need to integrate with other tools with Spring Sleuth to achieve this.

Spring Boot provides many supported APIs/libraries. Let's discuss a few important monitoring tools:

Spring Cloud Log tracing framework which adds trace IDs and span IDs.

Collection of tools, APIs, and SDKs. You use it to instrument, generate, collect, and export telemetry data (metrics, logs, and traces) for analysis in order to understand your software's performance and behavior.

Visual representation of microservices traces. It's a distributed tracing system. It helps gather timing-based trace data needed to troubleshoot latency problems in microservices architectures.

We can use **OpenZipkin**, which is the fully open-source version of Zipkin, a project that originated in Twitter in 2010 and is based on the Google Dapper papers.

All this additional information in your logs is great but making sense of it all can be quite cumbersome. Using something like the EFK stack to collect and analyze the logs from your microservices can be quite helpful.

Distributed tracing with Spring Cloud Sleuth

In this coding exercise, we will add correlation IDs (trace ID and span ID) using Spring Cloud Sleuth using Spring Boot microservices.

Prerequisite

The following are basic installation requirements:

Java 8.x

Spring Boot v2.2.7.RELEASE+

Spring Cloud Sleuth 2.2.7.RELEASE+

Source code reference is as follows:

<https://github.com/rajivmca2004/catalogue-service>

Let's refer to the same Spring Boot project **catalogue-service** which we have already created. We will only update its Maven **pom.xml** file and **application.yml** files:

Add the following Maven dependencies of Spring Cloud Sleuth:

Run your Spring Boot app after auto loading of Sleuth JAR dependencies.

Now, access the application by following REST API URL:

```
curl http://localhost:8010/catalogue
```

The moment this rest API is called to the back-end **Catalogue** microservice, it will print these logs with trace ID and span ID in the log files:

```
2021-02-12 23:37:21.677  INFO [catalogue-service, 120a49f10bd14d8a, 120a49f10bd14d8a, false] 1124 ---  
[nio-10-1-exec-1] c.o.s.d.controller.CatalogueController.
```

The first value is the **microservice name** and the second value is the **trace** The third value is the **span** Finally, the last value indicates whether the span should be exported to Zipkin.

Microservices monitoring

Microservices-based applications have different monitoring requirements. The business logic applied to a process such as a online eCommerce application in a distributed system where many microservices interact with each other. To track these interconnected microservices apps, we need a common correlating data.

The following are some of the best considerations before designing the monitoring policy:

Why and where distributed microservices should be monitored?

What monitoring data should be gathered when monitoring? It could be logs and matrices, and so on.

What open source and enterprise tools are available for publishing, gathering, and storing monitoring data?

How to effectively monitor and generate alerts pro-actively before app or infra crash?

Failure is the most important reason why monitoring is important. IT systems are not just up or down, they should be also healthy and generate metrics to decide the workload status of the CPU, memory, and storage usages. Smart and modern monitoring

systems are capable of alerting proactively before any failure. They also do predictive analysis to procure and scale the IT infrastructure.

Service providers are bound with their customer/clients with **Service Level Agreement**. It's not possible to monitor the SLA without continuous monitoring whether apps are deployed on public, on-premise, or hybrid cloud. Monitoring systems over time produces valuable data that can be used to improve the service performance. Failure and performance data can be analysed to look for patterns in system failures, which can be correlated with events.

There are many **Application Performance Monitoring** tools available which works on **Artificial Intelligence**. There are also many open-source tools available like **Grafana** and **Prometheus** to pull and visualize monitoring data of microservices apps. These tools are used to publish, gather, and store metrics data. The storage tier is specifically designed to handle time-series data. There are various supported databases to store time-series data and faster writing like Cassandra, Elasticsearch, and so on.

These tools are also capable to monitor the performance of front-end, REST APIs, and overall systems.

Reactive microservices will be monitored the same as non-reactive. In many ways, monitoring sensors will pull logs async way and forward to observability APM tools.

Monitoring microservices with Prometheus and Grafana

In this section, we will do a hands-on exercise of setting up Prometheus with Spring Boot microservices which pulls the metrics information from apps and exposes through APIs. These metrics can be used by any visual web dashboard tools like Grafana, which pulls all the metrics and displays on their console. These monitoring APM tools can also create visual reports and alerts also.

Prometheus is a popular and widely used open source for event monitoring and alerting.

Let's start the lab exercise!

Prerequisite

The following are basic installation requirements:

Java 8.x

Spring Boot v2.2.7.RELEASE+

Prometheus

Grafana

Source code reference is as follows:

<https://github.com/rajivmca2004/catalogue-service>

Let's refer to the same Spring Boot project **catalogue-service** which we have already created. We need to follow these steps:

Add the following Prometheus dependencies in this microservice Spring Boot app. The Spring Actuator module contains the actuator endpoints and the actuator endpoints allow you to monitor and interact with your application:

Refer to all APIs which you can access to monitor your microservice app:

<https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-features.html#production-ready-endpoints-exposing-endpoints>

Prometheus dependency exposed app matrices through Add the following configurations to expose using JMX and API:

management:

endpoint:

health:

enabled: `true`

show-details: always

endpoints:

web:

exposure:

include: '*'

jmx:

exposure:

include: '*'

Now, go to this URL in your browser or access using CURL command:

It will return all app matrices like CPU usage and memory information:

```
# HELP system_cpu_usage The "recent cpu" the whole system
```

```
# TYPE system_cpu_usage gauge
```

```
system_cpu_usage 0.18282619907168643
```

```
# HELP jvm_memory_used_bytes The amount of used memory  
  
# TYPE jvm_memory_used_bytes gauge
```

Survivor

Old

Eden

Class 9956992.0

Install Grafana using the Docker image. It can be deployed on your machine/VM or Kubernetes container:

```
docker grafana/grafana:7.4
```

You can refer to the latest version and installation details from here:

<https://grafana.com/docs/grafana/latest/installation/docker/>

Run Grafana on user port

```
docker run -d -p grafana/Grafana:7.4.0
```

To open the Grafana dashboard, go to the following URL on your web browser: It will ask for a username and password for the first time. You can provide the default credentials. It will also ask you to change your password first time. There is an option to set the administration user and password in It will display the home page as shown in the following screenshot:

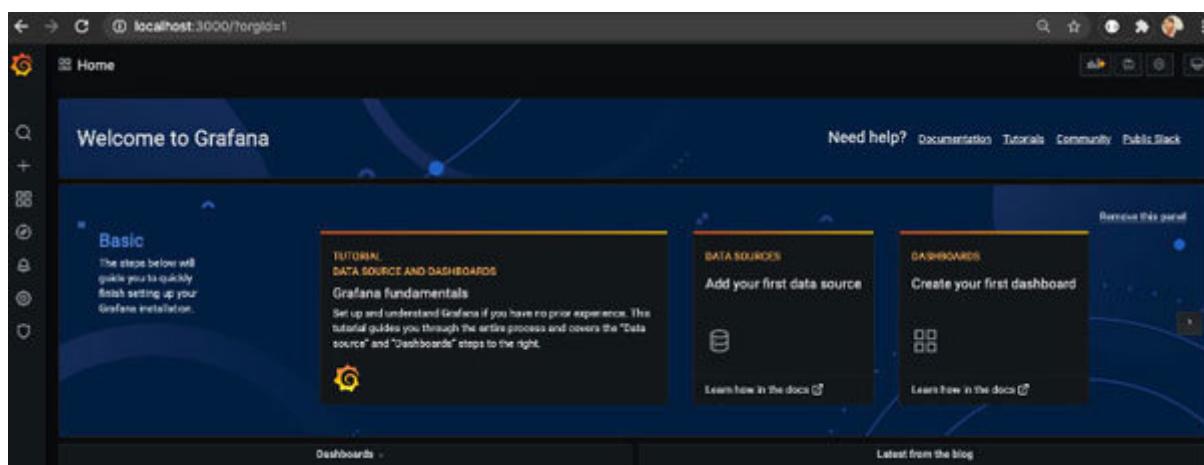


Figure 9.9: Grafana home page

Now, add the Prometheus plug-in using the **DATA SOURCES** option from the home page. You can select various other data sources:

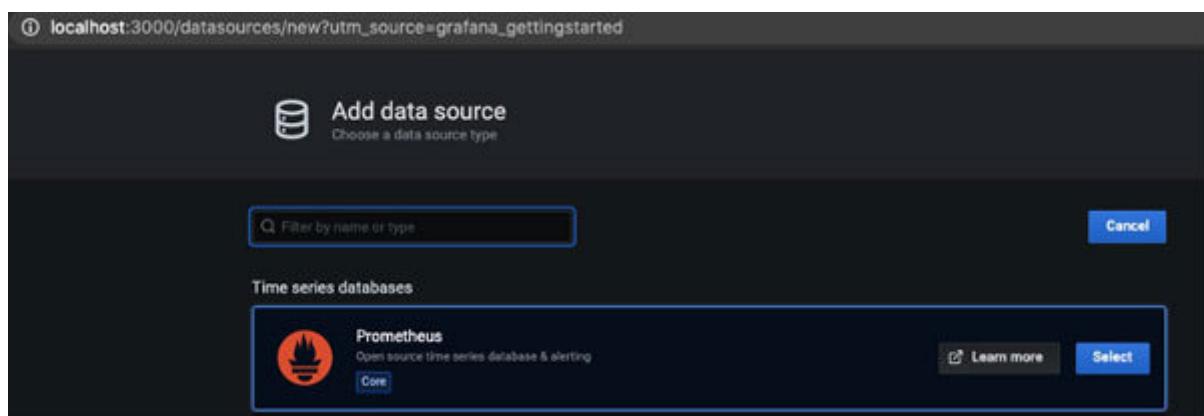


Figure 9.10: Grafana add source page

Add the Prometheus server URL and other configuration details like credential, auth type, and so on:

The screenshot shows the 'Data Sources / Prometheus' configuration page in Grafana. At the top, there's a header with a Prometheus icon and the text 'Data Sources / Prometheus' and 'Type: Prometheus'. Below the header, there are two tabs: 'Settings' (which is active) and 'Dashboards'. A sub-header says 'Configure your Prometheus data source below' with a close button ('X'). Below this, a note encourages users to skip effort and get Prometheus (and Loki) as fully managed, scalable and hosted data sources from Grafana Labs with the [free-forever Grafana Cloud plan](#).

The main configuration area is divided into several sections:

- HTTP**: Contains fields for 'URL' (set to `http://localhost:8010/actuator/prometheus`), 'Access' (set to 'Server (default)'), and 'Whitelisted Cookies'.
- Auth**: Contains toggle switches for 'Basic auth', 'With Credentials', 'TLS Client Auth', 'With CA Cert', 'Skip TLS Verify', and 'Forward OAuth Identity'.
- Custom HTTP Headers**: A section with a '+ Add header' button.
- Misc**: Contains a 'Disable metrics lookup' toggle switch and a 'Custom query parameters' input field with an example: `max_source_resolution=5m&timeout=10`.

Figure 9.11: Grafana Prometheus config page

Import a set of default dashboards to start displaying useful information:

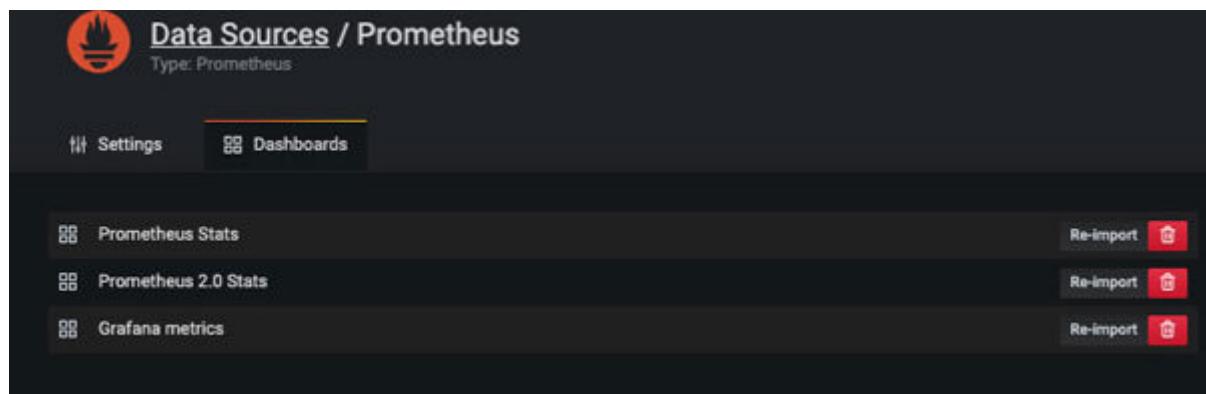


Figure 9.12: Grafana default dashboards

Create a new dashboard with a graph:

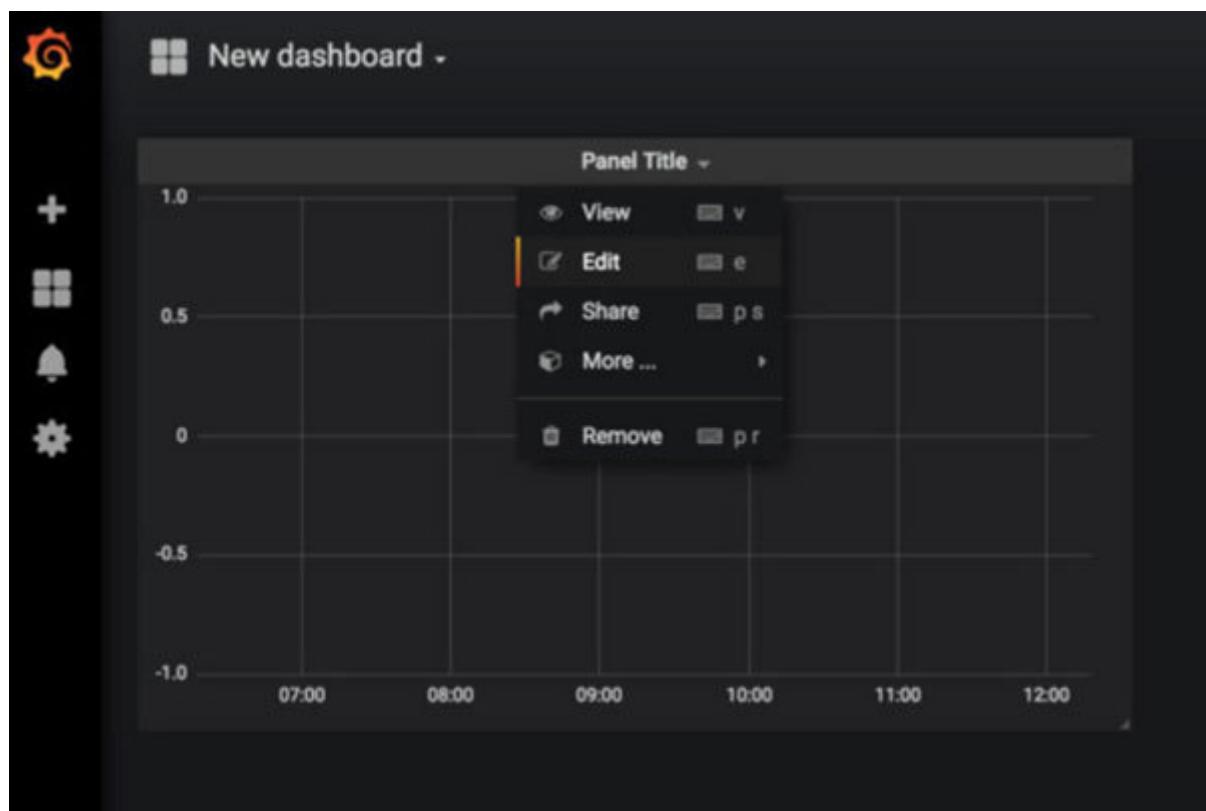


Figure 9.13: Grafana new dashboard create page

Add a Prometheus query expression in Grafana's query editor as shown in the following screenshot:

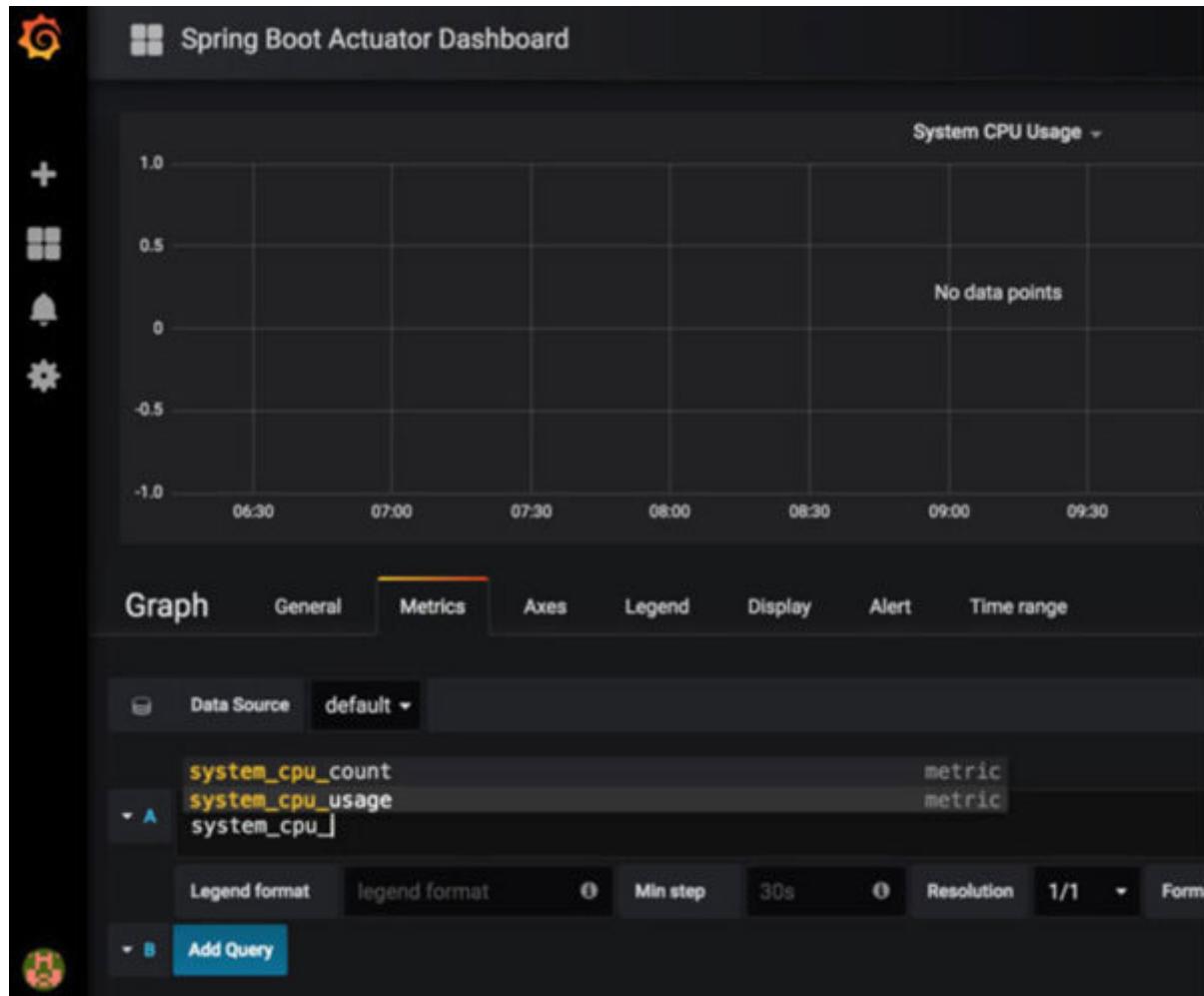


Figure 9.14: Grafana query expression page

Visualize metrics from Grafana's dashboard:



Figure 9.15: Grafana matrices and graphs

Conclusion

In this chapter, we discussed observability and monitoring techniques, their needs, and best practices of microservices in detail. We discussed and implemented a few monitoring tools like Spring Micrometer health APIs, Prometheus, and Wavefront APM. We understood microservices simple logging and log aggregation using Spring Boot. We also discussed how to trace multiple microservices in a distributed environment. We covered the needs of APM-like performance and telemetry monitoring tools for distributed microservices and integration with Prometheus and Grafana.

In the next chapter, we will discuss an overview of Kubernetes orchestration tool, its architecture and advantages. We will also cover how to manage Kubernetes clusters using different tools and techniques.

Points to Remember

It is recommended that you use SLF4j on any logging abstraction layer. It will help to change underlying logging framework tools.

Use the micrometer abstraction layer on other monitoring tools in apps.

Fluentd is a log collector, processor, and aggregator. Fluent Bit is a lightweight subset of Fluentd log collector and processor. It doesn't have strong aggregation features such as Fluentd.

Spring's provides Logback logging API integration by default. However, we can choose any other supported logging framework or tools.

EFK is faster and has better advantages over legacy Logstash based ELK for Kubernetes platform. FluentD seems the ideal candidate because it uses Docker logging driver and parser, which is suitable for Kubernetes container platform.

Key terms

Elasticsearch Fluent Bit Kibana

Elasticsearch Logstash Kibana

Application Performance Management

Command Line Interface

Kubernetes

Service Level Agreement

Artificial Intelligence

Java Management Extensions

References

Spring

Wavefront

Prometheus open source: <https://prometheus.io/>

Grafana open source: <https://grafana.com/>

Elasticsearch open source: <https://www.elastic.co/>

FluentD open source: <https://www.fluentd.org/>

Fluent Bit open source: <https://fluentbit.io/>

Kibana open source: <https://www.elastic.co/kibana>

CHAPTER 10

Containers and Kubernetes Overview and Architecture

Kubernetes is a new revolution and new **Infrastructure as a Service (IaaS)** for cloud native modern apps and infra. It's a container orchestration tool. It's also known as **K8s** , a popular open source system for automating deployment, scaling, and management of containerized applications. It provides containers to deploy modern applications and provides a containerized highly available and resilient infrastructure environment. It's a platform to build other platforms and primarily provides container orchestration and other resources to provide a new API driven infrastructure and platform layer on top of VMs.

This chapter will cover containers, Docker, apps containerization, Kubernetes overview, and architecture at a higher level. We will discuss image container registry and how to persist Docker images in the image registry.

We will discuss Kubernetes overview, principles, and advantages. We will discuss fundamental concepts of Kubernetes pods, Docker, Docker desktop, and process to build Docker images using Java, Spring framework, Maven, and other related technologies. Finally, we will cover a quick introduction of Kubernetes resources.

Let's get started!

Structure

In this chapter, we will cover the following topics:

Container overview

Containerization benefits

Comparison of Container and **Virtual Machine**

Container image registry

The Docker container

Docker Engine

OCI container image using Buildpacks

Pack

kpack

Difference between pack and kpack

Install and configure Docker Desktop

Components of Dockerfile

Dockerfile example

Build Dockerfile

Inspect the Docker image

Run Dockerfile

Build and run Docker image for Java app

Public repositories using Docker Hub

Private repositories using Harbor

Install Harbor Image Registry

Introduction of Kubernetes

Kubernetes architecture

Master control plane/Management cluster

Worker cluster

Need of Kubernetes

Kubernetes resources

Namespace

Service

Ingress

Ingress controller

Deployment

ConfigMap

Secret

Taints and tolerations

RBAC

Network policies

Storage

CronJob

ReplicaSet

StatefulSet

Objective

After studying this chapter, you will be able to learn about containers, containerization benefits, Docker container, Docker Engine, container image registries, **pack** and **kpack** OCI Docker image build tools. It will help to understand Docker components, how to build Docker images, and run on Docker. It will also cover Kubernetes architecture and quick introduction of a few important Kubernetes resources.

Container overview

It's a small software unit that packages source code, their dependencies, and OS configuration. A container includes an application binaries and all its runtime dependencies. Each container has its own set of processes and services.

Container is a new VM for modern microservices-based apps. It provides smooth portability of microservices apps to build anywhere and run anywhere because it contains **Operating System** specific configuration on top OS kernel:

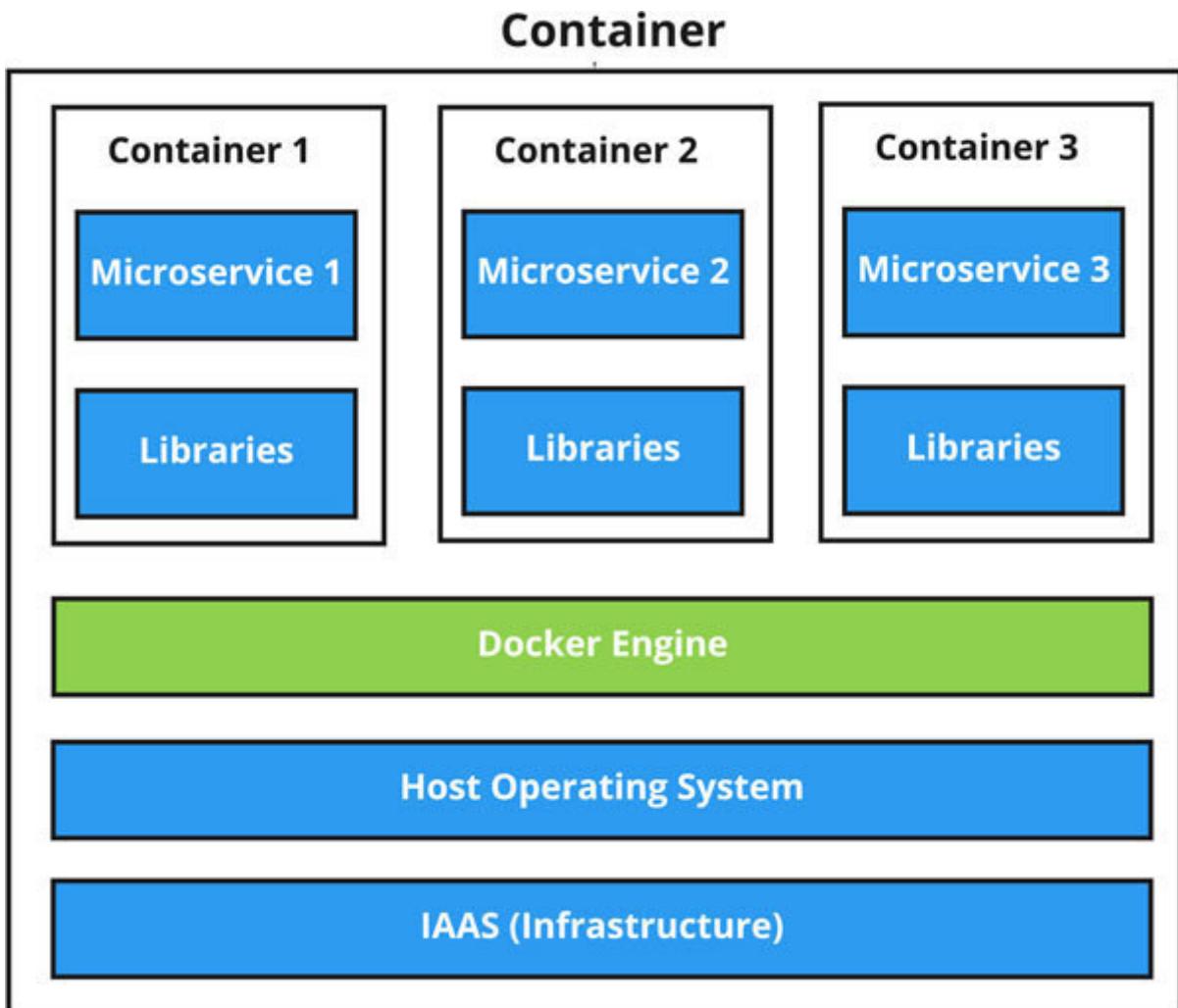


Figure 10.1: Container architecture

Container can be installed and run apps on any container environment without any environmental glitch. Internally, it uses Docker images. These Docker images run on containers using a runtime environment such as **Java** and These Docker images are stored in image registry like Harbor, AWS **Elastic Container Registry** Google **Container Registry** and so on.

Container shares the same host OS on the IaaS infrastructure. There is a Docker container that uses docker LXC runtime. It's a proper implementation of the twelve-factor principle. It provides an

immutable artifact for promotion across various non-production and production environments.

Kubernetes is deprecating Docker as a container runtime after Kubernetes v1.20.

Newer versions of Kubernetes used **ContainerD** runtime where apps run on containers. Kubernetes is deprecating Docker as a container runtime after Kubernetes v1.20. This doesn't mean the death of Docker, and it doesn't mean you can't, or shouldn't use Docker as a development tool anymore. Docker is still a useful tool for building containers, and the images that result from running Docker build can still run in your Kubernetes cluster.

Containers decouple applications from the underlying host OS infrastructure. This makes deployment easier in different cloud, IaaS, or OS environments.

Comparison of container and virtual machines

These are a few differences of container and VM:

Container has faster performance than VM.

Container uses less CPU and memory utilization than VM.

The hypervisor allows multiple VMs to run on a single machine. VM has their own guest OS, however container shares host OS kernel.

Containers are lighter and auto expandable than VM.

VMs are an abstraction of physical hardware turning one server into many servers. Containers and VMs used together provide a great deal of flexibility in deploying and managing app.

Container provides advanced isolation over VM because mostly single microservice apps run on a single container. VM is useful when a strong security boundary is critical.

VM provides load balancing to move running VMs workload to other servers in a failover cluster. Containers themselves don't move. On the other hand, a container orchestrator can

automatically start or stop containers on cluster nodes to manage changes in load and availability:

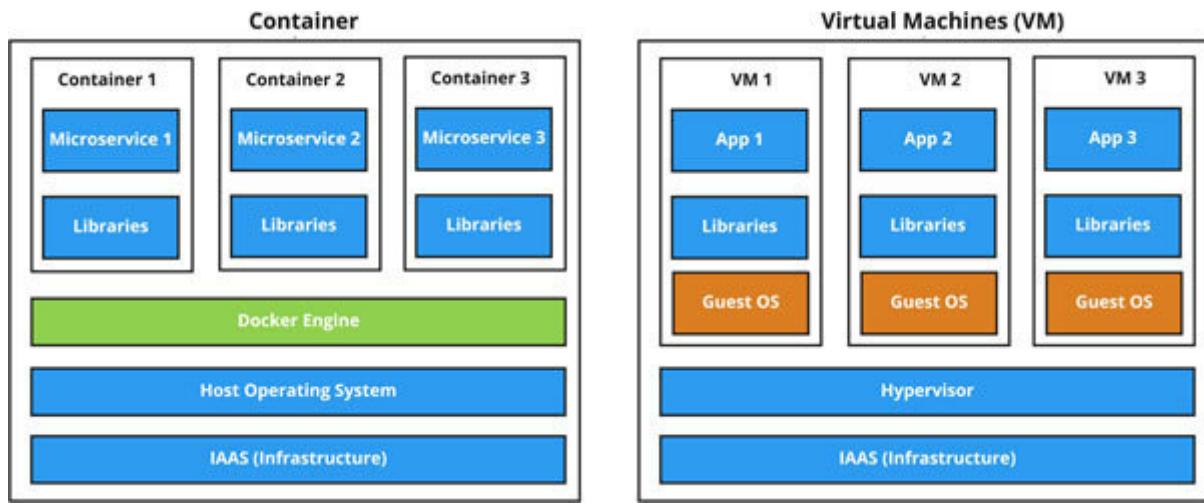


Figure 10.2: Container versus VM

Containerization benefits

These are some major benefits of application containerization:

Compute

Containerized microservices apps are lightweighted, they can run on lesser memory and CPU.

CI/CD integration

Easy integration with the CI/CD pipeline.

Provides faster deployment using various CI/CD DevOps tools.

Easy shipping of microservices apps to any environment.

Supports Linux and Windows based workload apps.

Smooth portability for different on-premises and cloud platforms and OS.

Provide enhanced security by running microservices apps in isolated containers with their own security restrictions and policies.

Business continuity

Provide business continuity by providing resiliency. If one container fails, it won't impact entire apps.

Easy No need to configure dev env from scratch for new on-boarded developers.

Easy customization and

Faster customization of individual container configuration.

Provide flexibility to run on **Bare Metal** and VMs.

Easy day 2 operation and

Quick and easy upgrades and patching.

Easier auto-scaling.

Higher

Higher productivity by supporting microservices app development and deployment. Containers allow developers to change and track the changes in the platform's source code, thus enhancing productivity. By reducing the dependency on the main server or

OS resources, they simplify and save time while installing applications on different platforms.

Easier infra

Easier infrastructure management using Kubernetes install, update and upgrade.

Container image registry

A container image registry is a software service that stores container images, and is hosted either by a third-party or as a public/private registry such as **Docker Hub** and These image registries are backed up by large storage on VM/BM.

It's a repository to persist docker images, and provides tools for image scanning for security vulnerabilities. DevOps team can integrate this image registry with the CI/CD pipeline. **Continuous Integration** part builds Docker container image and pushes to container image registry. **Continuous Delivery** part pulls these container images and deploys on Docker Engine/Docker Desktop or Kubernetes containers.

The following diagram shows the role of the container registry between CI/CD pipelines:

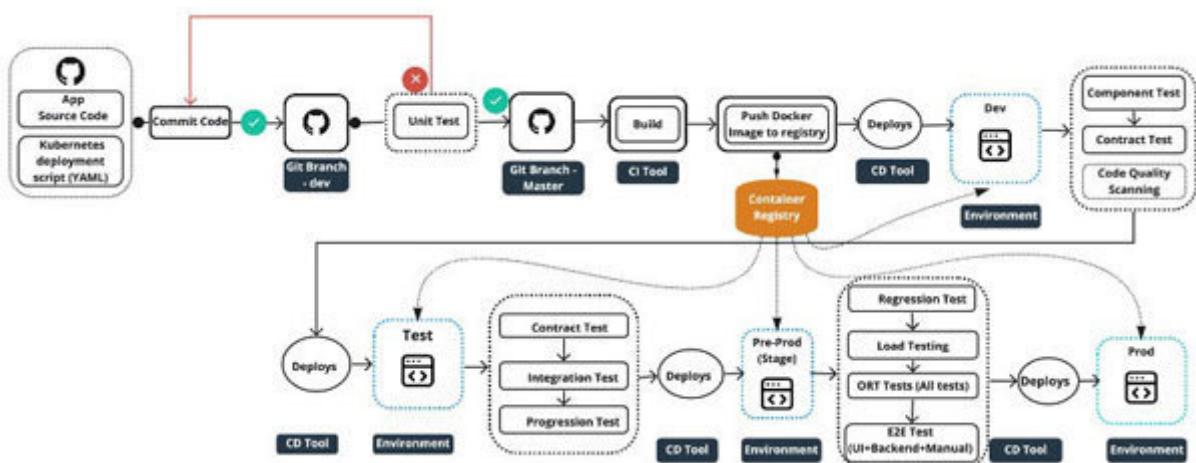


Figure 10.3: CI/CD architecture with container registry

This image scanning process comes under **DevSecOps** principal.

These are some popular image registries:

Good for on-prem registry repository.

Docker Public cloud for development and testing purpose.

Store secure K8s images.

Enterprise for artifact management, supporting all package types.

Store and manage Docker and provide a single source of truth.

AWS Elastic Container Registry.

Google Container Registry.

It builds, analyzes, and distributes container images.

Docker container

Docker is a runtime container image that contains source code, dependent libraries, and OS base layer configuration. It provides a portable container that can be deployed and run on any platform. Docker images run on containers like Kubernetes containers.

Docker is a first citizen for the Kubernetes containers. It's a tool for developers to package all the deployable source code, dependencies and environment dependencies. DevOps can use this as a tool to deploy on Kubernetes containers.

Docker is more suitable to package microservices and run on any private, public, and hybrid Kubernetes clusters.

Dockerization is a process to convert any source code to docker portable image. Docker container images are in a container image registry. It works on "build once and run anywhere" principal.

Docker Engine

Docker Engine is a containerization technology for building and containerizing your applications. It provides a runtime environment where applications run on containers. The Docker Engine is made up of three core components:

A server with a long-running daemon process

APIs which specify interfaces that programs can use to talk to and instruct the Docker daemon.

A **Command Line Interface** Docker client.

OCI container image using Buildpacks

The **Open Container Initiative** is a standard authority to standardized Docker as a runtime container. It provides industry standard container image formats and runtimes to run faster with ease.

The Cloud Native Buildpacks project was initiated by Pivotal (VMware) and Heroku in January 2018 and joined the Cloud Native Computing Foundation in October 2018. The project aims to unify the buildpack ecosystems with a platform-to-buildpack contract that is well-defined and that incorporates learnings from maintaining production-grade buildpacks for years at both Pivotal and Heroku.

Cloud Native Buildpacks embrace modern container standards, such as the OCI image format. They take advantage of the latest capabilities of these standards, such as cross-repository blob mounting and image layer "rebasing" on Docker API v2 registries. Cloud Native Buildpacks transform your application source code into images that can run on any cloud or Docker engine. Another big advantage is that we get more control of the base container image and all the dependencies and developers don't have to write giant YAML files. Cloud Native Buildpacks embrace modern container standards such as the OCI image format. They take advantage of the latest capabilities of these standards. Also, Buildpacks supports modern programming languages such as Java, Python, Node.js, .NET, and PHP. It auto-detects programming

language and their configuration like Maven/Gradle configuration and injects dependent libraries into the Docker image with the source code, which makes images to easily ship and run on any environment.

There are other options to create Docker images using these Buildpacks open sources tools.

Pack

Pack is an open-source tool maintained by the Cloud Native Buildpacks project to support the use of Buildpacks, which we have discussed in the last section.

It provides these functionalities:

Build a Docker image using application source code and inject dependent libraries using Buildpacks images.

Rebase application images created using Buildpacks.

Creation of various dependent components used within the ecosystem.

Pack works both as a CLI and a Go library. You can simply install on MacOS using the Homebrew CLI command:

```
$ brew install buildpacks/tap/pack
```

You can refer this link for other installation options:

<https://buildpacks.io/docs/tools/pack/>

kpack

kpack is a Kubernetes-native build service that builds container images on Kubernetes using Cloud Native Buildpacks. It takes source code Git repositories like GitHub, builds the code into a Docker container image, and uploads it to the container registry of your choice like Docker Hub or Harbor.

It provides an automated way to listen to Git repositories and builds Docker images using Git code for any commits. It also provides a manual way to trigger build. Internally, it uses Buildpacks images and Pack library.

Refer to this link for more information:

<https://github.com/pivotal/kpack>

Difference between Pack and kpack

The major difference is Pack just builds container images manually using Buildpacks, means the developer has to build image and push to image registry or add these in the CI/CD script. On the other hand, kpack is a Kubernetes-native build service that builds container images on Kubernetes using Cloud Native Buildpacks and push to desired image registry automatically on every code commit on the Git server. Pack doesn't listen to Git server code changes.

Install and configure Docker Desktop

Install Docker Desktop that comes with Docker CLI. The Docker CLI requires the Docker daemon, so you'll need to have that installed and running locally on your machine or bootstrap host.

Create a Docker Hub account in the following link and get set go to push and pull images:

<https://hub.docker.com/>

You can create Docker images locally, then you have a choice to push images to Docker Hub cloud SaaS or set a local on-premise Harbor private container image repository for security and privacy reasons.

Your Docker Desktop should always be running when you work with Docker containers such as building, packaging, running, and persisting in image registry. You can use the same Docker Hub login credentials for Docker Desktop on your local machine, which runs in the background as a daemon service.

Components of Dockerfile

These are important sections of Dockerfile:

It's a prerequisite to create Dockerfiles. In our example, it should have dependency on the OpenJDK image.

Set the environment variable for Dockerfile.

Create a work directory and all commands will be run in the context of the given directory.

It provides defaults when executing a container.

It's used to copy one or many local files or folders from the source and add them to the filesystem of the containers at the destination path.

It's used to add one or many local files or folders from the source and adds them to the filesystem of the containers at the destination path.

It will run commands on the current image and commit the result, which can be used for the next step in the Dockerfile.

It informs the Docker that the container listens on the specified network ports at runtime. We can specify whether the port listens on TCP/UDP, default is TCP if the protocol is not specified.

It will create a mount point with the specified volume name, for example,

Variable that users can pass at build-time to the builder with the Docker **build** command.

It allows you to configure a container that will run as an executable. It executes commands before starting the app on the container. If the **ENTRYPOINT** isn't specified, Docker will use **/bin/sh -c** as the default executor.

Dockerfile example

In this following Docker file, we will define a sequence of commands to pick up a parent or independent images using create a volume mount point with file path, pass an argument for the JAR file, add files to the Docker image, and then finally define a set of commands, which will execute during container bootup. It should be saved with the file name **Dockerfile** and placed inside home folder/directoty of the project source code binaries.

We need to define these commands in a sequence like the following:

```
# Docker Image name: catalogue-service-app-demo
```

```
# Use the official image as a parent image.

FROM openjdk:8-jdk-alpine

# Creates a mount point with the specified name

VOLUME /tmp

# variable that users can pass at build-time to the builder with
the docker build command

ARG JAR_FILE

# Add files into docker image

ADD ${JAR_FILE} app.jar

# Allows you to configure a container that will run as an
executable

ENTRYPOINT
```

Build Dockerfile

This command builds Docker file using Docker CLI:

```
$ docker image build -itsrajivsrivastava/ .
```

There are multiple ways to use docker files, one of them is listed here. Every programming language also has their own tools and libraries:

Maven Install Maven CLI and libraries and run this command:

```
$ mvn clean install dockerfile:build
```

Cloud Native Buildpacks Pack CLI Install Pack library by using this link:

<https://buildpacks.io/docs/tools/pack/>

1. **# Install BuildPack Pack CLI**
2. **\$ brew install buildpacks/tap/pack**
- 3.
4. **# Build docker image using pack CLI**
5. **\$ pack build sample-app --path samples/apps/java-maven --builder cnbs/sample-builder:bionic**

It uses Cloud Native Buildpacks. Refer to this URL:

<https://github.com/pivotal/kpack>

Tanzu Build service It's based on kpack. Refer to this URL:

<https://tanzu.vmware.com/build-service>

Inspect Docker image

Normally, Docker image file sizes are very huge ($>\sim 50$ because they contain application and OS dependent libraries along with application source code. Internally, Docker stores these contents in separate layers (logically split files). When any changes happen in source code or libraries, only impacted layers get changed and only that layer pushes to the image registry. It saves a lot of upload/download time from image registry to Docker or Kubernetes containers.

There is an open source **dive** tool, which provides these features:

Inspects Docker images layer contents with details.

It estimates image efficiency like how much wasted space that Docker image contains.

It shows image changes like which files and layers have been modified/added/removed, and so on.

It builds Docker images and inspect with a single command: **\$ dive build -t**

It also helps in analyzing images for pass/fail results based on image efficiency and helps in CI integration pipeline invocation.

You can download **Dive** tool from here:

<https://github.com/wagoodman/dive>

Let's inspect a real Spring Boot Java application using this tool. We need to run this command on CLI after installing the **dive** tool:

\$ dive

1. # Install and inspect docker image:
2. \$ brew install dive

3. \$ dive itsrajivsrivastava/spring-petclinic

It will generate a Docker inspection report like this:

Layers			Current Layer Contents			
Layer	Size	Command	Permission	UID:GID	Size	Filetree
63 MB	FROM 8bf067b107a6f74		-rwxr-xr-x	0:0	4.8 MB	bin
745 B			-rwxr-xr-x	0:0	1.1 kB	└── bash
7 B			-rwxr-xr-x	0:0	35 kB	└── bunzip2
225 B			-rwxr-xr-x	0:0	0 B	└── bzipcat => bin/bunzip2
619 kB			-rwxr-xr-x	0:0	0 B	└── bzcat => bin/bzip2
398 kB			-rwxr-xr-x	0:0	2.1 kB	└── bzcmp => bzip2
4.5 kB			-rwxr-xr-x	0:0	0 B	└── bzdiff
214 B			-rwxr-xr-x	0:0	4.9 kB	└── bzgrep => bzgrep
141 kB			-rwxr-xr-x	0:0	0 B	└── bzgrep
3.0 kB			-rwxr-xr-x	0:0	3.6 kB	└── bzgrep
10 B			-rwxr-xr-x	0:0	0 B	└── bzgrep
1.6 kB			-rwxr-xr-x	0:0	14 kB	└── bzgrep
53 kB			-rwxr-xr-x	0:0	0 B	└── bzless => bzmore
3 B			-rwxr-xr-x	0:0	1.3 kB	└── bzmore
50 kB			-rwxr-xr-x	0:0	35 kB	└── cat
2.4 kB			-rwxr-xr-x	0:0	64 kB	└── chgrp
619 kB			-rwxr-xr-x	0:0	60 kB	└── chmod
			-rwxr-xr-x	0:0	68 kB	└── chown
			-rwxr-xr-x	0:0	142 kB	└── cp
			-rwxr-xr-x	0:0	121 kB	└── dash
			-rwxr-xr-x	0:0	101 kB	└── date
			-rwxr-xr-x	0:0	76 kB	└── dd
			-rwxr-xr-x	0:0	134 kB	└── df
			-rwxr-xr-x	0:0	72 kB	└── dir
			-rwxr-xr-x	0:0	8 B	└── dmesg
			-rwxr-xr-x	0:0	0 B	└── dnsdomainname => hostname
			-rwxr-xr-x	0:0	35 kB	└── domainname => hostname
			-rwxr-xr-x	0:0	28 kB	└── echo
			-rwxr-xr-x	0:0	31 kB	└── egrep
			-rwxr-xr-x	0:0	28 kB	└── false
			-rwxr-xr-x	0:0	65 kB	└── fgrep
			-rwxr-xr-x	0:0	220 kB	└── findmnt
			-rwxr-xr-x	0:0	2.3 kB	└── grep
			-rwxr-xr-x	0:0	5.9 kB	└── gunzip
			-rwxr-xr-x	0:0	102 kB	└── gexec
			-rwxr-xr-x	0:0	18 kB	└── gzip
			-rwxr-xr-x	0:0	27 kB	└── hostname
			-rwxr-xr-x	0:0	68 kB	└── kill
			-rwxr-xr-x	0:0	53 kB	└── ln
			-rwxr-xr-x	0:0	134 kB	└── login
			-rwxr-xr-x	0:0	84 kB	└── ls
			-rwxr-xr-x	0:0	88 kB	└── lsblk
			-rwxr-xr-x	0:0	68 kB	└── mkdir
						└── mknod

Figure 10.4: Container versus VM

Run Dockerfile

The Run Dockerfile command runs the Docker image on **8010** port on localhost or where it's deployed:

```
$docker run -p 8010:8010 itsrajivsrivastava/catalogue-service:latest
```

Build and run Docker image for Java app

In this hands-on code example, we will create a Docker image of a Java/Spring Boot microservice app and store it in the Docker Hub image registry and finally how to run this app on a local Docker Desktop client.

Docker image registry is a persistent storage to store Docker images from where it can be pulled by the CI/CD pipeline or K8s deployments and deployed on a container.

Docker build services like Docker, Buildpack CNB kpack, Google Cloud Jib, VMware **Tanzu Build Service** and other plug-ins build Docker images and store to Docker registries. These docker images can be updated automatically after any code commit in the source code repositories like GitHub, Bitbucket, Gitlab etc.

Spring has recently added **GraalVM** for compiling and building native Docker images of Spring Boot applications. Currently it supports Java and Kotlin languages. These native Spring applications can be deployed as a standalone executable (no JVM installation required) and offer interesting characteristics, including almost instant start-up typically less than 100 ms, instant peak performance, and lower memory consumption at the cost of longer build times and fewer runtime optimizations than the JVM. Refer to this link for details:

<https://github.com/spring-projects-experimental/spring-native>

With simple **mvn spring-boot:build-image** or **gradle bootBuildImage** commands, you can generate an optimized container image that will contain a minimal OS layer and a small native executable that ships just the required bits from the JDK, Spring, and the dependencies that you are using in your application.

Prerequisite

These are basic installation requirements:

Docker Desktop for Mac. You can download other images/libraries of other OS like Windows and Linux servers.

Docker Hub account.

Java 8.x+.

Maven

<https://maven.apache.org/install.html>

<https://git-scm.com/book/en/v2/Getting-Started-Installing-Git>

<https://docs.brew.sh/Installation>

Source code reference:

Let's follow these steps to build and run a Docker image for Java microservices app using the Spring native:

Run Docker Desktop/Engine and login using the Docker CLI command. If your Docker Desktop is already running and you have logged on your machine, then no need to provide Docker login credentials. If you are using anything other than Docker Hub, then you must explicitly log into that image container registry using their credentials. Docker Hub provides a secret token, which is advisable to use when connecting local Docker container or during login:

1. # Docker-Hub Login:

2.

3. \$ docker login

Create a Java project using Spring Boot and add a Docker image prefix in Maven's **pom.xml** file. This prefix will be your image registry ID, in our case Docker Hub user ID:

1.

2.

3.

4.

5. itsrajivsrivastava

6.

Add the Docker Maven plug-in:

com.spotify

dockerfile-maven-plugin

1.3.7

`${docker.image.prefix}/${project.artifactId}`

`target/${project.build.finalName}.jar`

Create a Docker image descriptor configuration file in our **root** folder of application source code with the file name:

```
# Docker Image name: catalogue-service-app-demo
```

```
# Use the official image as a parent image.
```

```
FROM
```

```
# Creates a mount point with the specified name  
  
VOLUME /tmp  
  
# variable that users can pass at build-time to the builder with  
the docker build command  
  
ARG JAR_FILE  
  
# Add files into docker image  
  
ADD ${JAR_FILE} app.jar  
  
# Allows you to configure a container that will run as an  
executable
```

ENTRYPOINT

Build Docker images using Maven. If you are using Maven and Spring Boot app to build the Docker image, go to the source project folder, and run this Maven command. You need to install Maven before running the following command on Mac, Linux, and Windows:

```
$ mvn clean install dockerfile:build
```

The Maven command to push image to the current image registry (we need to be logged in on Docker Hub or Harbor on your local

system):

```
$ mvn install dockerfile:push
```

List all Docker images:

1. \$ docker image ls
- 2.

3. #Search specific docker image
- 4.
5. \$ docker image ls | grep

Show a breakdown of the various layers in the Docker image.

Docker split image in smaller chunks, which is called If any docker image is updated, only the updated layer will be pushed to docker image registry (Docker Hub in our case).

```
$ docker image history :latest
```

You can also try to build an image like this for non-Java projects. Go to the project folder of source code's home path (in this case, it's Java based Spring Boot) project and run this command:

```
$ docker image build -t itsrajivsrivastava/ .
```

Push the image to Docker/Harbor registry:

Tag your image before pushing:

1. \$ docker tag /image:tag
- 2.
3. #Docker-Hub:
4. \$ docker tag itsrajivsrivastava/catalogue-service
itsrajivsrivastava/catalogue-service:latest

Now you should be able to push it:

1. #Docker-Hub Push (When you are logged-in to Docker-Hub thru local Docker Desktop client)
- 2.
3. #Docker-Hub:
4. docker push itsrajivsrivastava/catalogue-service:latest

Pull image to Docker from Docker Hub:

1. \$ docker pull
- 2.
3. #Docker-Hub:
4. \$ docker pull itsrajivsrivastava/catalogue-service:latest

Run Docker images locally. Just run this Docker CLI command to run a container Docker image. It will run our Spring Boot microservice app **catalogue-service** on port which can be accessed using a web browser:

```
docker run -p itsrajivsrivastava/catalogue-service:latest
```

Now, test application by hitting this app URL on the web browser:

```
1.  [
2.    {
3.
4.
5.      100.5
6.    },
7.    {
8.
9.      "name":
10.     "price": 200.51
11.   },
12.   {
13.
14.
15.     "price": 600.0
16.   },
17.   {
18.
19.
20.     "price": 900.5
21.   }
22. ]
```

Public repositories using Docker Hub

Docker Hub is a Docker image registry, which is available as a SaaS service on cloud for public. They also offer a paid private image repository. It provides an easy way to start with to push and pull images from Kubernetes deployments.

This is a very convenient public cloud where anyone can create their Docker Hub account and store Docker images free.

Docker Hub also provides a private repository.

Private repositories using Harbor

There are many private repositories like Harbor, JFrog, Google Container Registry, Amazon **Elastic Container Registry** which are available on on-premises and on public cloud as a paid service.

We will cover Harbor private registry, which is open source, can be deployed locally on-premises.

Install Harbor image registry

Install Harbor by referring this link, you can choose latest version:

<https://goharbor.io/docs/1.10/install-config>

If your Docker Desktop is already running and you have logged on your machine, then no need to provide Docker login credentials.

Introduction of Kubernetes

Kubernetes is an open-source orchestration tool created by Google. It is written in the GO language. Kubernetes is also known as With Kubernetes, you can run any Linux container across on-premises and multi-cloud environments. Kubernetes provides some powerful features such as ingress/ egress load balancer, service discovery, auto-scaling of container, security, config map to store key-value pairs, node affinity, taint and tolerations, secret to store credentials, and certificates. It can be installed on VM and **Bare Metal**

Kubernetes architecture

Let's dive deep and understand how K8s works with its internal architecture. There are multiple components of Kubernetes:

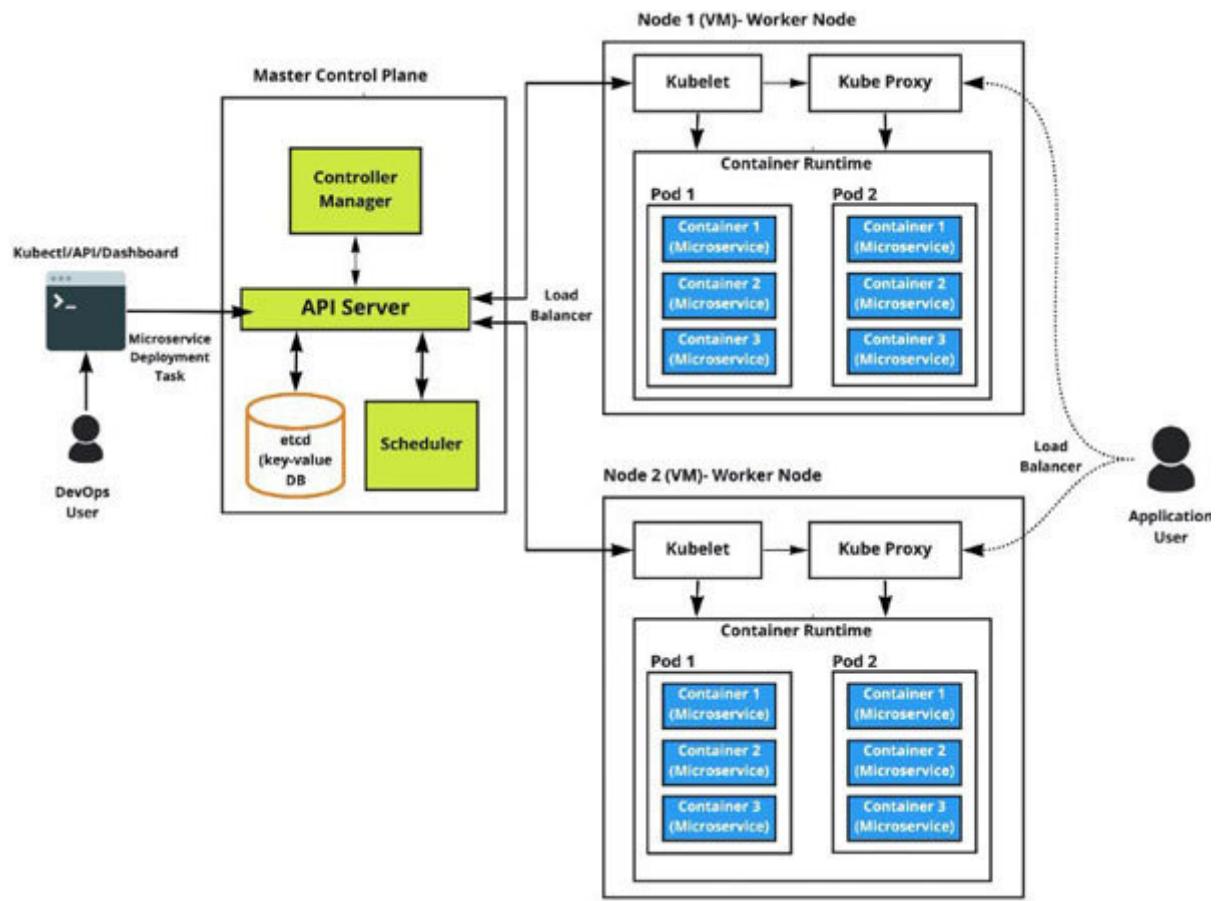


Figure 10.5: Kubernetes architecture

Before understanding the architecture, let's discuss some important components first, which is based on the preceding architecture diagram.

There are two major modules of any K8s clusters orchestration and management system.

Master control plane/management cluster

Master control plane works as a manager and administers K8s worker nodes/clusters. It has an admin and other services that help to orchestrate the worker K8s cluster.

Control place manages the K8s cluster with data about the cluster's state and configuration. It handles the important work of making sure your containers are running in sufficient numbers and with the required resources.

These are the main components of management cluster:

API It's a front-end control plane and set of APIs that serves Kubernetes functionality through a RESTful interface and stores the state of the cluster, which can be accessed by **Kubectl** or **Kubeadm** CLI tools. It also validates if the request is valid.

It checks K8s cluster health and watches API server for new pod requests. It communicates with nodes to create new pods and to assign work to nodes while allocating CPU and memory resources.

Controller It takes care actual running of the cluster. It runs controllers, includes node controller, endpoint controller, namespace controller, and so on. Controller consults the scheduler and makes sure the correct number of pods are running. If a pod

goes down, another controller notices and responds. A controller connects services to pods, so requests go to the right endpoints.

It's a persistent key-value database service that stores config data about cluster state of the K8s cluster. It's used during the K8s cluster backup in case of cluster failure.

Worker cluster

Worker cluster hosts all app workloads and exposes to real app users. These are controlled by a control plane and do their assigned job/tasks:

All containers will run in a pod. Pods abstract the network and storage away from the underlying containers. Our cloud native microservices app will run here. It's the smallest logical unit that contains one or more containers. One container per pod is recommended for microservice apps. An additional sidecar container is needed to handle cross-cutting concerns or non-functional requirements like externalize logging, filtering, service discovery using service mesh and envoy proxy, and so on.

Each compute or worker node contains a **Kubelet** agent. It's a service that communicates with the Kubernetes control plane. It registers nodes with the Kubernetes API server and watches for worker node assignments from the Kubernetes scheduler and instantiates a new pod when needed. It also makes sure containers are running in a pod.

Container runtime This is responsible for managing containers life cycle such as pull images, start/stop container, and destroying the container. It supports Docker Engine as default and supports **ContainerD** from Kubernetes version v.20. It also supports **CRI-O** and **RKT** container runtime.

Kube It's responsible for forwarding app user requests to the right pod. Each compute node also contains a network proxy for facilitating Kubernetes networking services inside or outside of the cluster.

Need of Kubernetes

Docker runtime engine is good to run apps on a single container on a single node. There are multiple challenges with Docker container deployment model like if any Docker container fails, then it must be restarted manually. Also, it doesn't provide auto scaling/high-availability and other advanced features of security, externalize configuration, managing clusters of multiple nodes, and so on.

It's a nightmare to manage microservices driven containerization such as health check, version control, scaling, and rollback mechanism. Kubernetes gives you the orchestration and management capabilities required for containers, at scale on production environment.

Kubernetes orchestration allows you to build microservices apps that spread to multiple containers, their scheduling across K8s cluster, scaling, and managing the health of those containers. It spawns new containers, pulls container images from the image registry, and deploys on container, when any container fails. Kubernetes manages containers.

Without Kubernetes, large DevOps teams must manually manage K8s deployment scripts. You don't need to create your K8s deployment script manually. It makes most of the configuration automatically, which will reduce the amount of time and operational cost.

These are some key features of Kubernetes:

Auto-scaling.

Service discovery.

Self-healing.

RBAC security.

Plugins and admission hook controllers.

Externalize configuration management.

Secret vault to store credentials.

Auto health check and self-healing.

Built-in load balancer with ingress and egress controllers.

Service discovery.

Automated rollbacks and rollouts.

Canary deployment.

Cluster API manage worker nodes by management control plane.

Service discovery and other rich K8s plug-ins support.

It automatically mounts storages with its storage orchestration.

Kubernetes resources

These are some of the important K8s objects and concepts, which are building blocks for this container platform. Let's understand with a quick and easy definition with example.

Namespace

It's a logical grouping of K8s resources. It provides multiple virtual separation on the physical cluster where apps can be deployed in separate namespaces. It's like a folder. We can also set resource limits at namespace level like CPU and memory.

Service

It's a logical grouping of pods, which can be exposed as a service name for internal and external communication. There are three kinds of services:

Load It exposes service to external public networks using cloud provider interface or load balancer, which can be connected to other third-party interfaces.

Cluster It exposes services internally within K8s cluster resources. These services can be interconnected with each other for internal microservices communication.

Node It exposes service to external public networks directly thru node's static port at node level. It's not advisable to use on production servers. These services can be directly accessed by a node's IP and port.

Ingress

Ingress acts as the entry point for K8s cluster. It consolidates routing rules into a single resource mainly HTTP as it can expose multiple services under the same IP address. It's advisable to expose microservices/services through ingress controllers and resources.

Ingress controller

For the ingress resource to work, the cluster must have an ingress controller running. An ingress controller is a tool/library that provides reverse proxy, configurable traffic routing, and TLS termination for Kubernetes services. **Istio** **Kong** and **HAproxy** are a few popular ingress controllers.

Deployment

Deployment contains K8s desired deployment configuration to deploy pods/containers and **ReplicaSets** on K8s clusters.

ConfigMap

ConfigMap stores key-value pairs and provides externalize configurations for all the apps/microservices. It stores external configuration like database host details and another configuration at the central place.

Secret

It manages sensitive key-value pairs as an external configuration in the encoded format. It stores passwords, credentials, OAuth tokens, certificate, SSH keys, and so on. It's safe and secure.

Taints and tolerations

Node affinity is a property of pods that decides on which node pod will be deployed. Taint tolerates pods creation on node based on the given condition.

[RBAC](#)

Role-Based Access Control is a method of regulating access to computer or network resources based on the roles of individual users within your organization.

Network policies

Network policies manage a network traffic at the pod level (OSI layer 3 and 4). It allows to specify how a pod can communicate with various network entities.

Storage

K8s uses the following storage volumes:

It's a file directory that is accessible to the containers in a pod which denotes physical storage.

Persistent Volumes It's a storage cluster on K8s, which can be accessed and used by pod containers. It's linked to the volume mount and allocated to pods through persistent volume claim.

Persistent Volume Claim PVC a logical claim object which bounds persistent volume. It connects to backend storage volumes. They request the storage resources that K8s deployment needs.

CronJob

CronJob runs scheduled jobs periodically on a given scheduled time. It's used for batch jobs execution or to run temporary tasks.

ReplicaSet

ReplicaSet is a K8s deployment type that ensures guarantee of running specified number of pods at any time. Even though if pod fails, it will keep on redeploying new pod on the K8s cluster.

[StatefulSets](#)

StatefulSet manages pods that are based on an identical container specification. It maintains a sticky identity for each of their pods. These pods are not interchangeable. It works well for persistent databases like MySQL, Elasticsearch, and so on.

DaemonSets

It ensures that some or all your K8s nodes run a copy of a pod, which allows them to run a daemon on every node. When a new node is added to the cluster, a pod gets added to match the nodes. Similarly, when you remove a node from your cluster, the pod is put into the trash. Deleting a DaemonSet cleans up the pods that it previously created. It's mainly used for log collection, aggregation, and monitoring.

Conclusion

In this chapter, we discussed about containers overview and containerization benefits. We discussed container image registries and Docker fundamentals. Also, gone through the process to build Docker images, OCI images, and various tools to create Docker images. We have discussed Kubernetes architecture in detail with a use case of deploying a microservice by DevOps team and access by application users. Towards the end, we covered a quick introduction of Kubernetes resources.

In the next chapter, we will discuss more about Kubernetes installation and its cluster management. We will also discuss a couple of tools to visualize and manage K8s clusters. We will cover a few real use cases with examples to build a Docker image from source code, push to container image registry, deploy on the K8s cluster, and expose to application users or front-end apps using ingress controller.

Points to Remember

Newer versions of Kubernetes used ContainerD runtime where apps run on containers. Kubernetes is deprecating Docker as a container runtime after Kubernetes v1.20.

Dockerization is a process to convert any source code to Docker portable image. Docker stores in a container image registry. It works on build once and run anywhere principal.

Your Docker Desktop should always be running when you work with Docker containers such as building, packaging, running, and persisting in container image registry. You can use the same Docker Hub login credentials for Docker Desktop, which runs in the background as a daemon service.

Docker Hub also provides a private repository.

If your Docker Desktop is already running and you have logged on your machine, then no need to provide Docker login credentials. A lot of organizations don't allow for Docker Engine or Desktop since it needs admin permissions. In this scenario, a separate boot machine/VM deployed for local container testing and Docker images management, and final golden image will be pushed to image registry. During deployment, this golden image will be picked up from image registry and deployed on non-prod/prod Kubernetes containers.

Use GraalVM to improve performance of Java microservices. It increases app throughput and reduces latency. It compiles Java apps into small self-contained libraries. Refer to this link for more information:

<https://www.graalvm.org/>

Key terms

Kubernetes.

Open Container Initiative.

Infrastructure as a Service.

Virtual Machine.

Bare Metal.

Role Based Access Control.

Continuous Integration/Continuous Delivery.

Operation System.

Command Line Interface.

Elastic Container Registry.

Persistent Volumes.

Persistent Volume Claim.

References

Docker

Tanzu Build Service

Maven

Git

Homebrew

Spring

CHAPTER 11

Run Microservices on Kubernetes

After building container images and pushing them to the image registry, we need to deploy and run them on a container orchestration platform like Kubernetes. The deployment part is usually taken care by **continuous delivery (CD)** tools like **Jenkins** , **Jenkins X** , **Argo CD** , **Skaffold** , **Concours** , **GitLab** , **Tekton** , **Google DevOps** , **AWS DevOps** , **Azure DevOps** , and so on.

In this chapter, we will discuss practical aspects of Kubernetes and its installation and configuration with monitoring and visualization tools. We will discuss how to create and manage Kubernetes clusters in detail. We will do a few code exercises by creating Docker images of Java microservices, pushing them to the Docker Hub container image registry and deploying them to Kubernetes clusters. At the end, we will expose these REST API endpoints of microservices outside of the Kubernetes cluster by using the NGINX ingress controller.

Let's get started!

Structures

In this chapter, we will cover the following topics:

Installing and running **kubectl** CLI commands

kubectl command modes

Kubernetes installation

kind tool

Minikube

Kubeadm

Enterprise Kubernetes

Kubernetes management UI tools Octant and Proxy

Set up Octant K8s dashboard UI

Set up K8s proxy dashboard UI

Kubernetes application deployment and configuration management tools:

Helm

Developer loves Helm

Difference between Helm 2 and Helm 3

Install Helm

Kubeapps

Install Kubeapps

Kubernetes operator

Skaffold

Tekton

Kustomize

Jenkins X

Spinnaker

Knative

Microservice deployment on Kubernetes container with NGINX ingress controller

Prerequisite

Objective

After studying this chapter, you should be able to learn the **kubectl** CLI overview and installation with different methods/tools. This chapter will help you to understand the various K8s management tools like Kubernetes web admin dashboard tools, application deployment, and configuration tools. You will learn how to build a container image of a microservice, store it in a container image registry and deploy it on the K8s cluster using Kubernetes **Custom Resource Definition**

Installing and running kubectl CLI commands

kubectl CLI is a very important K8s tool that internally connects to the K8s API server and sends K8s related instructions to manage clusters and deploy microservice apps. It's a must to install CLI in your environment. It will give a set of commands through which we can create K8s clusters, manage, deploy the app, create security policies, manage multiple K8s clusters, check container logs, create config map, and secret for key-value externalization and a lot more.

It can be installed on macOS, Windows, and Linux-based servers. **kubectl** provides a command-line interface to enter K8s commands and connects with K8s APIs.

It also uses K8s configuration YAML/JSON files for deployment and configuration. There are some UI K8s dashboard tools to manage K8s which we will discuss later.

In our case, we will install **kubectl** on macOS. Run this command on the macOS terminal:

```
# Install K8s using Homebrew
```

```
$ brew install kubectl
```

```
# Test kubectl installation
```

```
$ kubectl version --client
```

You all are set to install K8s and create clusters, which we will cover in the next sections.

Refer to the kubectl install reference documents at

kubectl command modes

Always keep the cheat-sheet handy as you start with Kubectl command line interface (CLI). It provides two ways to run the commands:

It provides a faster way to create and manage K8s objects using CLI imperative commands. In this approach, all operations are performed on live objects and it reflects at run-time.

An example is as follows:

```
$ kubectl create deployment --image=nginx nginx-app
```

It provides a way to create and manage declarative K8s objects using YAML/JSON configuration files. In this mode, the creation, deletion, and modification of objects are done through a single command.

An example is as follows:

```
$ kubectl apply -f
```