Mastering Microservices with Java

Third Edition

Build enterprise microservices with Spring Boot 2.0, Spring Cloud, and Angular

Packt

www.packkt.com

Sourabh Sharma

Mastering Microservices with Java

Third Edition

Build enterprise microservices with Spring Boot 2.0, Spring Cloud, and Angular

Sourabh Sharma

Packt

BIRMINGHAM - MUMBAI

Mastering Microservices with Java
Third Edition

Copyright © 2019 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Commissioning Editor: Aaron Lazar
Acquisition Editor: Denim Pinto
Content Development Editor: Zeeyan Pinheiro
Technical Editor: Romy Dias
Copy Editor: Safis Editing
Project Coordinator: Vaidehi Sawant
Proofreader: Safis Editing
Indexer: Mariammal Chettiyar
Graphics: Alishon Mendonsa
Production Coordinator: Deepika Naik

First published: June 2016
Second edition: December 2017
Third edition: February 2019

Production reference: 1220219

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78953-072-8

`www.packtpub.com`

To my adored wife, Vanaja, and son, Sanmaya, for their unquestioning faith, support, and love.
To my parents, Mrs. Asha and Mr. Ramswaroop, for their blessings.

mapt.io

Mapt is an online digital library that gives you full access to over 5,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

Packt.com

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

At www.packt.com, 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 Packt books and eBooks.

Contributors

About the author

Sourabh Sharma has over 16 years of experience in product/application development. His expertise lies in designing, developing, deploying, and testing N-tier web applications and leading teams. He loves to troubleshoot complex problems and develop innovative ways to solve problems. Sourabh believes in continuous learning and sharing your knowledge.

I would like to thank Zeeyan, Romy, and the reviewers for their hard work and critical review feedback. I also would like to thank Packt Publishing and Denim for providing me with the opportunity to write this edition.

About the reviewer

Aristides Villarreal Bravo is a Java developer, member of the NetBeans Dream Team and the Java User Groups community, and a developer of the jmoordb framework. He is currently residing in Panama. He has organized and participated in various conferences and seminars related to Java, Java EE, NetBeans, the NetBeans Platform, open source software, and mobile devices, both nationally and internationally. He is a writer of tutorials and blogs for web developers about Java and NetBeans. He has participated in several interviews on sites including NetBeans, NetBeans Dzone, and JavaHispano. He is a developer of plugins for NetBeans.

> *I want to thank my parents and brothers for their unconditional support (Nivia, Aristides, Secundino, and Victor).*

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.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.

Table of Contents

Preface 1

Section 1: Section 1: Fundamentals

Chapter 1: A Solution Approach 9
- **Services and SOA** 10
  - Monolithic architecture overview 11
  - Limitations of monolithic architectures versus its solution with microservices architectures 11
    - Traditional monolithic design 12
    - Monolithic design with services 13
- **Microservices, nanoservices, teraservices, and serverless** 13
  - One-dimension scalability 15
  - Release rollback in case of failure 16
  - Problems in adopting new technologies 16
  - Alignment with agile practices 17
  - Ease of development – could be done better 18
  - Nanoservices 20
  - Teraservices 20
  - Serverless 20
- **Deployment and maintenance** 21
  - Microservices build pipeline 21
  - Deployment using a containerization engine such as Docker 22
    - Containers 22
    - Docker 23
      - Docker's architecture 24
- **Deployment** 25
- **Summary** 25

Chapter 2: Environment Setup 26
- **Spring Boot** 27
  - Adding Spring Boot to our main project 28
- **REST** 31
  - Writing the REST controller class 35
    - The @RestController annotation 35
    - The @RequestMapping annotation 36
    - The @RequestParam annotation 36
    - The @PathVariable annotation 37
  - Making a sample RESTapplication executable 40
- **An embedded web server** 41
- **Maven build** 42

Running the Maven build from IDE 42
Maven build from the Command Prompt 43
Testing using Postman 44
Some more positive test scenarios 47
Negative test scenarios 48
Summary 49
Further reading 50

Chapter 3: Domain-Driven Design 51
Domain-driven design (DDD) fundamentals 52
The fundamentals of DDD 53
Building blocks 53
Ubiquitous language 53
Multilayered architecture 54
Presentation layer 55
Application layer 56
Domain layer 56
Infrastructure layer 56
Artifacts of DDD 56
Entities 57
Value objects 58
Services 60
Aggregates 61
Repository 63
Factory 64
Modules 66
Strategic design and principles 66
Bounded context 66
Continuous integration 68
Context map 68
Shared kernel 70
Customer-supplier 70
Conformist 71
Anti-corruption layer 71
Separate ways 71
Open Host Service 72
Distillation 72
Sample domain service 73
Entity implementation 73
Repository implementation 76
Service implementation 78
Summary 82

Chapter 4: Implementing a Microservice 83
OTRS overview 84
Developing and implementing microservices 86
Restaurant microservice 87
OTRS implementation 88
Restaurant service implementation 90
Controller class 92

API versioning 92
Service classes 94
Repository classes 96
Entity classes 99
Booking and user services 105
Execution 105
Testing 105
Microservice deployment using containers 112
Installation and configuration 112
Docker Machine with 4 GB of memory 113
Building Docker images with Maven 113
Running Docker using Maven 118
Integration testing with Docker 118
Managing Docker containers 119
Executing Docker Compose 120
Summary 122

Section 2: Section 2: Microservice Patterns, Security, and UI

Chapter 5: Microservice Patterns - Part 1 124
Service discovery and registration 125
Spring Cloud Netflix Eureka Server 126
Implementation 127
Spring Cloud Netflix Eureka client 129
Centralized configuration 133
Spring Cloud Config Server 134
Spring Cloud Config client 138
Execution and testing of the containerized OTRS app 142
Summary 145
References 146

Chapter 6: Microservice Patterns - Part 2 147
The overall architecture 148
Edge server and API gateway 151
Implementation 153
Demo execution 157
Circuit breaker 159
Implementing Hystrix's fallback method 161
Demo execution 165
Centralized monitoring 166
Enabling monitoring 167
Prometheus 173
Architecture 173
Integration with api-service 174
Grafana 179
Summary 189
Further reading 189

Chapter 7: Securing Microservices 190
Secure Socket Layer 190
Authentication and authorization 192
OAuth 2.0 193
Uses of OAuth 193
OAuth 2.0 specification – concise details 194
OAuth 2.0 roles 196
Resource owner 196
Resource server 197
Client 197
Authorization server 197
OAuth 2.0 client registration 197
Client types 198
Client profiles 199
Client identifier 201
Client authentication 202
OAuth 2.0 protocol endpoints 202
Authorization endpoint 202
Token endpoint 203
Redirection endpoint 203
OAuth 2.0 grant types 205
Authorization code grant 205
Implicit grant 209
Resource owner password credentials grant 211
Client credentials grant 212
OAuth implementation using Spring Security 213
Security microservice 213
API Gateway as a resource server 221
Authorization code grant 223
Using the access token to access the APIs 226
Implicit grant 227
Resource owner password credential grant 227
Client credentials grant 229
Summary 230
Further reading 230
Chapter 8: Consuming Services Using the Angular App 231
Setting up a UI application 232
Angular framework overview 235
MVC and MVVM 236
Angular architecture 236
Modules (NgModules) 237
Components 239
Services and dependency injection (DI) 240
Routing 240
Directives 241
Guard 243
Developing OTRS features 244
The home page 244

src/app.module.ts (AppModule) 246
src/app-routing.module.ts (the routing module) 247
src/rest.service.ts (the REST client service) 248
src/auth.guard.ts (Auth Guard) 251
app.component.ts (the root component) 251
app.component.html (the root component HTML template) 252
Restaurants list page 254
src/restaurants/restaurants.component.ts (the restaurants list script) 254
src/restaurants/restaurants.component.html (the restaurants list HTML template) 255
Searching for restaurants 256
Login page 257
login.component.html (login template) 258
login.component.ts 259
Restaurant details with a reservation option 260
restaurant.component.ts (the restaurant details and reservation page) 262
restaurant.component.html (restaurant details and reservation HTML template) 263
Reservation confirmation 265
Summary 266
Further reading 266

Section 3: Section 3: Inter-Process Communication

Chapter 9: Inter-Process Communication Using REST 268
REST and inter-process communication 269
Load balanced calls and RestTemplate implementation 270
RestTemplate implementation 272
OpenFeign client implementation 276
Java 11 HTTPClient 279
Wrapping it up 282
Summary 283
Further reading 283

Chapter 10: Inter-Process Communication Using gRPC 284
An overview of gRPC 284
gRPC features 285
REST versus gRPC 286
Can I call gRPC server from UI apps? 286
gRPC framework overview 287
Protocol Buffer 288
The gRPC-based server 291
Basic setup 291
Service interface and implementation 295
The gRPC server 298
The gRPC-based client 299
Summary 302
Further reading 302

Chapter 11: Inter-Process Communication Using Events 303
An overview of the event-based microservice architecture 303
Responsive 305
Resilient 305
Elastic 305
Message driven 305
Implementing event-based microservices 306
Producing an event 306
Consuming the event 313
Summary 318
Further reading 319

Section 4: Section 4: Common Problems and Best Practices

Chapter 12: Transaction Management 321
Design Iteration 321
First approach 322
Second approach 322
Two-phase commit (2PC) 322
Voting phase 323
Completion phase 323
Implementation 323
Distributed sagas and compensating transaction 324
Feral Concurrency Control 324
Distributed sagas 325
Routing slips 326
Distributed saga implementation 326
Saga reference implementations 327
Compensating transaction in the booking service 327
Booking service changes 327
Billing service changes 338
Summary 340
Further reading 340
Chapter 13: Service Orchestration 341
Choreography and orchestration 341
Choreography 342
Orchestration 342
Orchestration implementation with Netflix Conductor 343
High-level architecture 343
The Conductor client 344
Basic setup 345
Task definitions (blueprint of tasks) 347
WorkflowDef (blueprint of workflows) 349
The Conductor worker 351
Wiring input and output 353

Using Conductor system tasks such as DECISION 354
Starting workflow and providing input 355
Execution of sample workflow 356
Summary 359
Further reading 359
Chapter 14: Troubleshooting Guide 360
Logging and the ELK Stack 360
A brief overview 362
Elasticsearch 362
Logstash 363
Kibana 364
ELK Stack setup 364
Installing Elasticsearch 364
Installing Logstash 365
Installing Kibana 367
Running the ELK Stack using Docker Compose 367
Pushing logs to the ELK Stack 370
Tips for ELK Stack implementation 371
Using a correlation ID for service calls 372
Let's see how we can tackle this problem 372
Using Zipkin and Sleuth for tracking 372
Dependencies and versions 374
Cyclic dependencies and their impact 374
Analyzing dependencies while designing the system 375
Maintaining different versions 375
Let's explore more 375
Summary 376
Further reading 376
Chapter 15: Best Practices and Common Principles 377
Overview and mindset 377
Best practices and principles 379
Nanoservice, size, and monolithic 379
Continuous integration and continuous deployment (CI/CD) 381
System/end-to-end test automation 382
Self-monitoring and logging 383
A separate data store for each microservice 384
Transaction boundaries 385
Microservice frameworks and tools 386
Netflix Open Source Software (OSS) 386
Build – Nebula 387
Deployment and delivery – Spinnaker with Aminator 387
Service registration and discovery – Eureka 387
Service communication – Ribbon 388
Circuit breaker – Hystrix 388
Edge (proxy) server – Zuul 388
Operational monitoring – Atlas 389

Reliability monitoring service – Simian Army 389
AWS resource monitoring – Edda 390
On-host performance monitoring – Vector 391
Distributed configuration management – Archaius 391
Scheduler for Apache Mesos – Fenzo 392
Summary 392
Further reading 393
Chapter 16: Converting a Monolithic App to a Microservice-Based App 394
Do you need to migrate? 395
Cloud versus on-premise versus both cloud and on-premise 395
Cloud-only solution 395
On-premise only solution 396
Both cloud and on-premise solution 396
Approaches and keys to successful migration 397
Incremental migration 397
Process automation and tools setup 398
Pilot project 398
Standalone user interface applications 398
Migrating modules to microservices 400
How to accommodate a new functionality during migration 401
Summary 403
Further reading 403
Other Books You May Enjoy 404
Index 407

Preface

Presently, microservices are the de-facto way to design scalable, easy-to-maintain applications. Microservice-based systems not only make application development easier, but also offer great flexibility in utilizing various resources optimally. If you want to build an enterprise-ready implementation of a microservice architecture, then this is the book for you!

Starting off by understanding the core concepts and framework, you will then focus on the high-level design of large software projects. You will gradually move on to setting up the development environment and configuring it, before implementing continuous integration to deploy your microservice architecture. Using Spring Security, you will secure microservices and integrate sample **online table reservation system** (**OTRS**) services with an Angular-based UI app. We'll show you the best patterns, practices, and common principles of microservice design, and you'll learn to troubleshoot and debug the issues faced during development. We'll show you how to design and implement event-based and gRPC microservices. You will learn various ways to handle distributed transactions and explore choreography and orchestration of business flows. Finally, we'll show you how to migrate a monolithic application to a microservice-based application.

By the end of the book, you will know how to build smaller, lighter, and faster services that can be implemented easily in a production environment.

Who this book is for

This book is designed for Java developers who are familiar with microservice architecture and now want to effectively implement microservices at an enterprise level. A basic knowledge of Java and Spring Framework is necessary.

What this book covers

`Chapter 1`, *A Solution Approach*, starts with basic questions about the existence of microservices and how they evolve. It highlights the problems that large-scale on-premises and cloud-based products face, and how microservices deal with them. It also explains the common problems encountered during the development of enterprise or large-scale applications, and the solutions to these problems. Many of you might have experienced the pain of rolling out the whole release due to failure of one feature.

Microservices give the flexibility to roll back only those features that have failed. This is a very flexible and productive approach. For example, let's assume you are the member of an online shopping portal development team and want to develop an application based on microservices. You can divide your application based on different domains such as products, payments, cart, and so on, and package all these components as a separate package. Once you deploy all these packages separately, these would act as a single component that can be developed, tested, and deployed independently—these are called microservices.

Now let's see how this helps you. Let's say that after the release of new features, enhancements, and bug fixes, you find flaws in the payment service that need an immediate fix. Since the architecture is based on microservices, you can roll back just the payment service, instead of rolling back the whole release. You could also apply the fixes to the payment microservice without affecting the other services. This not only allows you to handles failure properly, but helps to deliver features/fixes swiftly to the customer.

`Chapter 2`, *Environment Setup*, teaches you how to set up the development environment from an **integrated development environment** (**IDE**), and looks at other development tools from different libraries. This chapter covers everything from creating a basic project, to setting up Spring Boot configuration, to building and developing our first microservice. Here, we'll use Java 11 as our language and Jetty as our web server.

`Chapter 3`, *Domain-Driven Design*, sets the tone for rest of the chapters by referring to one sample project designed using domain-driven design. This sample project is used to explain different microservice concepts from this chapter onward. This chapter uses this sample project to drive through different functional and domain-based combinations of services or apps to explain domain-driven design.

`Chapter 4`, *Implementing a Microservice*, takes you from the design to the implementation of a sample project. Here, the design of our sample project explained in the last chapter is used to build the microservices. This chapter not only covers the coding, but also other different aspects of the microservices—build, unit testing, and packaging. At the end of this chapter, the sample microservice project will be ready for deployment and consumption.

`Chapter 5`, *Microservice Pattern – Part 1*, elaborates upon the different design patterns and why these are required. You'll learn about service discovery, registration, configuration, how these services can be implemented, and why these services are the backbone of microservice architecture. During the course of microservice implementation, you'll also explore Netflix OSS components, which have been used for reference implementation.

`Chapter 6`, *Microservice Pattern – Part 2*, continues from the first chapter on microservice patterns. You'll learn about the API Gateway pattern and its implementation. Failures are bound to happen, and a successful system design prevents the failure of the entire system due to one component failure. We'll learn about the circuit breaker, its implementation, and how it acts as a safeguard against service failure.

`Chapter 7`, *Securing Microservices*, explains how to secure microservices with respect to authentication and authorization. Authentication is explained using basic authentication and authentication tokens. Similarly, authorization is examined using Spring Security 5.0. This chapter also explains common security problems and their solutions.

`Chapter 8`, *Consuming Microservices Using the Angular App*, explains how to develop a web application using AngularJS to build the prototype of a web application that will consume microservices to show the data and flow of a sample project – a small utility project.

`Chapter 9`, *Inter-Process Communication Using REST*, explains how REST can be used for inter-process communication. The use of `RestTemplate` and the Feign client for implementing inter-process communication is also considered. Lastly, it examines the use of load balanced calls to services where more than one instance of a service is deployed in the environment.

`Chapter 10`, *Inter-Process Communication Using gRPC*, explains how to implement gRPC services and how these can be used for inter-process communication.

`Chapter 11`, *Inter-Process Communication Using Events*, discusses reactive microservices and their fundamentals. It outlines the difference between plain microservices and reactive microservices. At the end, you'll learn how to design and implement a reactive microservice.

`Chapter 12`, *Transaction Management*, teaches you about the problem of transaction management when a transaction involves multiple microservices, and a call when routed through various services. We'll discuss the two-phase commit and distributed saga patterns, and resolve the transaction management problem with a distributed saga implementation.

`Chapter 13`, *Service Orchestration*, introduces you to different designs for establishing inter-process communication among services for specific flows or processes. You'll learn about choreography and orchestration. You will also learn about using Netflix Conductor to implement the orchestration.

`Chapter 14`, *Troubleshooting Guide*, talks about scenarios when you may encounter issues and get stuck. This chapter explains the most common problems encountered during the development of microservices, along with their solutions. This will help you to follow the book smoothly and will make learning swift.

`Chapter 15`, *Best Practices and Common Principles*, teaches the best practices and common principles of microservice design. It provides details about microservices development using industry practices and examples. This chapter also contains a few examples where microservice implementation can go wrong, and how you can avoid such problems.

`Chapter 16`, *Converting a Monolithic App to a Microservices-Based App*, shows you how to migrate a monolithic application to a microservice-based application.

To get the most out of this book

You need to have a basic knowledge of Java and Spring Framework. You can explore the reference links given at the end of each chapter to get the more out of this book.

For this book, you can use any operating system (out of Linux, Windows, or macOS) with a minimum of 4 GB RAM. You will also require NetBeans with Java, Maven, Spring Boot, Spring Cloud, Eureka Server, Docker, and a continuous integration/continuous deployment application. For Docker containers, you may need a separate virtual machine or cloud host, preferably with 16 GB or more of RAM.

Download the example code files

You can download the example code files for this book from your account at `www.packt.com`. If you purchased this book elsewhere, you can visit `www.packt.com/support` and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at `www.packt.com`.
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Mastering-Microservices-with-Java-Third-Edition`. 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 `https://github.com/PacktPublishing/`. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://www.packtpub.com/sites/default/files/downloads/9781789530728_ColorImages.pdf`.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: " First, we'll add Spring Cloud dependencies, as shown in `pom.xml`."

A block of code is set as follows:

```
logging:
    level:
      ROOT: INFO
      org.springframework.web: INFO
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
endpoints:
    restart:
        enabled: true
    shutdown:
        enabled: true
```

Any command-line input or output is written as follows:

```
Chapter6> mvn clean package
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "After the values are updated, click on the **Save and Test** button."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at `customercare@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packt.com/submit-errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

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 `copyright@packt.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 `authors.packtpub.com`.

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 Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packt.com`.

1
Section 1: Fundamentals

The following part of this book will teach you about the fundamentals of microservices and the basics that you need in order to implement microservice-based systems.

In this section, we will cover the following chapters:

- `Chapter 1`, *A Solution Approach*
- `Chapter 2`, *Environment Setup*
- `Chapter 3`, *Domain-Driven Design*
- `Chapter 4`, *Implementing a Microservice*

1
A Solution Approach

As a prerequisite for proceeding with this book, you should have a basic understanding of microservices and different software architecture styles. Having a basic understanding of these will help you understand what we discuss in this book.

After reading this book, you will be able to implement microservices for on-premises or cloud production deployments and you will understand the complete life cycle, from design and development to testing and deployment, of continuous integration and deployment. This book is specifically written for practical use and to stimulate your mind as a solution architect. Your learning will help you to develop and ship products in any situation, including **Software-as-a-Service** (**SaaS**) and **Platform-as-a-Service** (**PaaS**) environments. We'll primarily use Java and Java-based framework tools, such as Spring Boot and Jetty, and we will use Docker for containerization.

In this chapter, you will learn about microservices and how they have evolved. This chapter highlights the problems that on-premises and cloud-based products face and how microservices architectures deal with them. It also explains the common problems encountered during the development of SaaS, enterprise, or large applications and their solutions.

In this chapter, we will explore the following topics:

- Services and **service-oriented architecture** (**SOA**)
- Microservices, nanoservices, teraservices, and serverless
- Deployment and maintenance

Services and SOA

Martin Fowler explains the following:

> *The term microservice was discussed at a workshop of software architects near Venice in May 2011 to describe what the participants saw as a common architectural style that many of them had been recently exploring. In May 2012, the same group decided on µServices as the most appropriate name.*

Let's get some background on the way microservices have evolved over the years. Enterprise architecture evolved from historic mainframe computing, through client-server architecture (two-tier to *n*-tier), to SOA.

The transformation from SOA to microservices is not a standard defined by an industry organization, but a practical approach practiced by many organizations. SOA eventually evolved to become microservices.

Adrian Cockcroft, a former Netflix architect, describes a microservice-based architecture as follows:

> *Fine grain SOA. So microservice is SOA with emphasis on small ephemeral components.*

Similarly, the following quote from Mike Gancarz, a member who designed the X Windows system, which defines one of the paramount precepts of Unix philosophy, describes the microservice paradigm as well:

> *Small is beautiful.*

Microservice architectures share many common characteristics with SOAs, such as the focus on services and how one service decouples from another. SOA evolved around monolithic application integration by exposing APIs that were mostly **Simple Object Access Protocol** (**SOAP**)-based. Therefore, having middleware such as an **enterprise service bus** (**ESB**) is very important for SOA. Microservices are less complex than SOAs, and, even though they may use a message bus, it is only used for message transport and it does not contain any logic. It is simply based on smart endpoints.

Tony Pujals defined microservices beautifully:

> *In my mental model, I think of self-contained (as in containers) lightweight processes communicating over HTTP, created and deployed with relatively small effort and ceremony, providing narrowly-focused APIs to their consumers.*

Though Tony only talks about HTTP, event-driven microservices may use a different protocol for communication. You can make use of Kafka to implement event-driven microservices. Kafka uses the wire protocol, a binary protocol over TCP.

Monolithic architecture overview

Microservices are not new—they have been around for many years. For example, Stubby, a general purpose infrastructure based on **Remote Procedure Call** (**RPC**), was used in Google data centers in the early 2000s to connect a number of services with and across data centers. Its recent rise is due to its popularity and visibility. Before microservices became popular, monolithic architectures were mainly being used for developing on-premises and cloud-based applications.

A monolithic architecture allows the development of different components such as presentation, application logic, business logic, and **Data Access Objects** (**DAOs**), and then you either bundle them together in an **Enterprise Archive** (**EAR**) or a **Web Archive** (**WAR**), or store them in a single directory hierarchy (such as Rails or Node.js).

Many famous applications, such as Netflix, have been developed using a microservices architecture. Moreover, eBay, Amazon, and Groupon have evolved from monolithic architectures to microservices architectures.

Now that you have had an insight into the background and history of microservices, let's discuss the limitations of a traditional approach—namely, monolithic application development—and see how microservices would address them.

Limitations of monolithic architectures versus its solution with microservices architectures

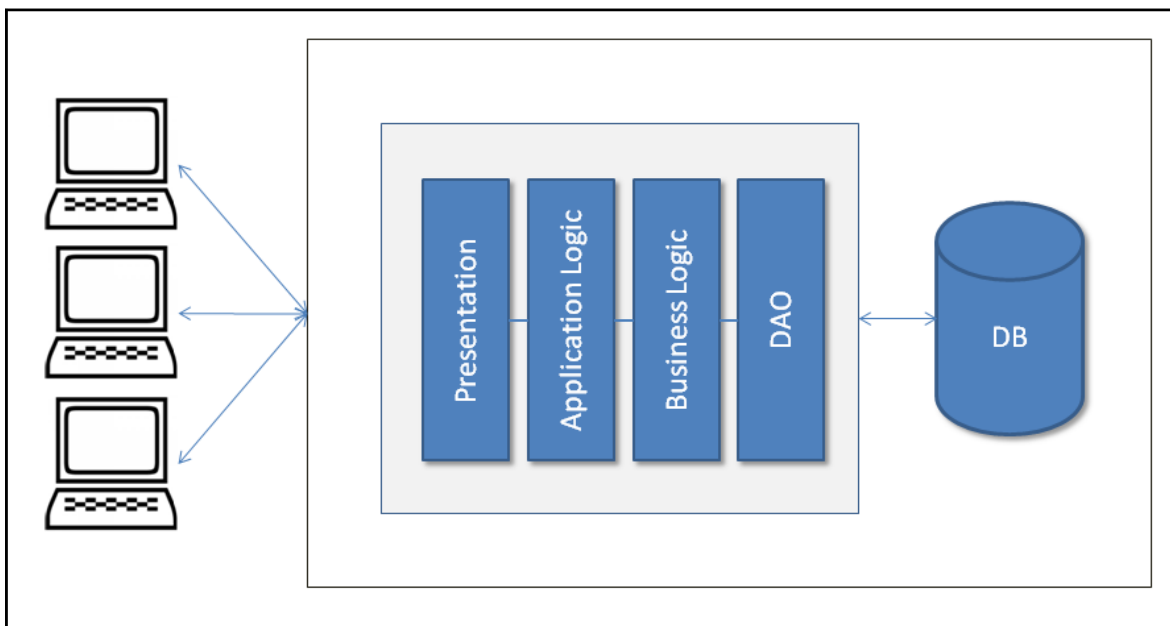
As we know, change is eternal. Humans always look for better solutions. This is how microservices became what it is today and it will evolve further in the future. Today, organizations are using agile methodologies to develop applications—it is a fast-paced development environment that has grown to a much larger scale after the invention of the cloud and distributed technologies. Many argue that monolithic architectures could also serve a similar purpose and be aligned with agile methodologies, but microservices still provide a better solution to many aspects of production-ready applications.

To understand the design differences between monolithic and microservices architectures, let's take an example of a restaurant table-booking application. This application may have many services to do with customers, bookings, analytics, and so on, as well as regular components, such as presentation and databases.

We'll explore three different designs here: the traditional monolithic design, the monolithic design with services, and the microservices design.

Traditional monolithic design

The following diagram explains the traditional monolithic application design. This design was widely used before SOA became popular:

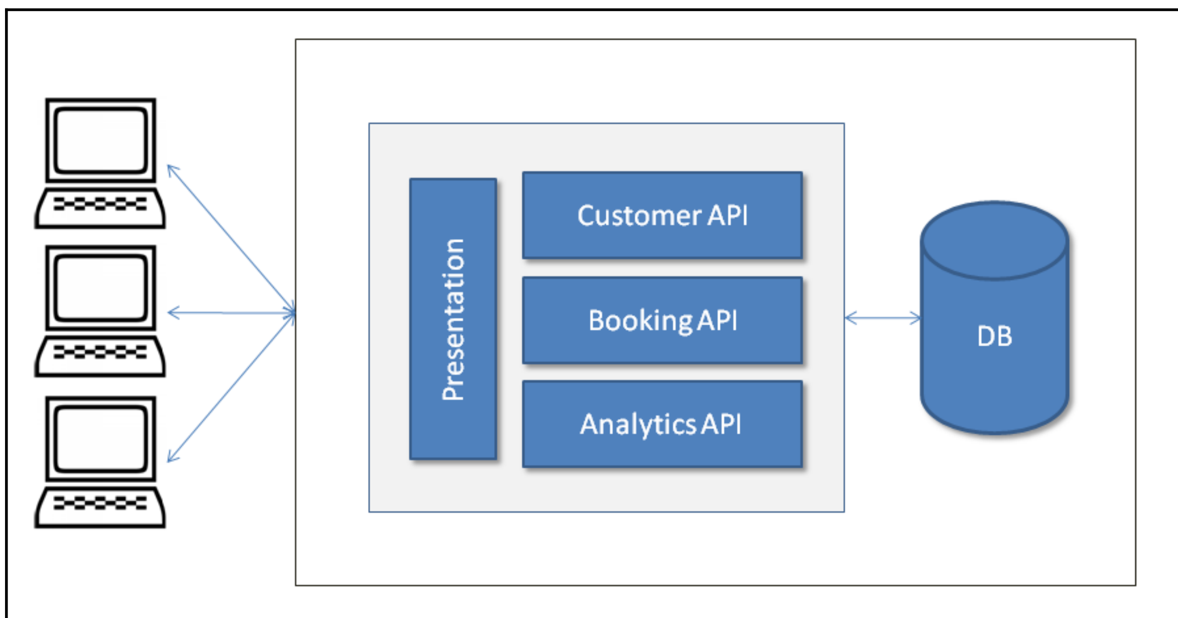


Traditional monolithic application design

In a traditional monolithic design, everything is bundled in the same archive (all the presentation code is bundled in with the **Presentation** archive, the application logic goes into the **Application Logic** archive, and so on), regardless of how it all interacts with the database files or other sources.

Monolithic design with services

After SOA, applications started being developed based on services, where each component provides services to other components or external entities. The following diagram depicts a monolithic application with different services; here, services are being used with a **Presentation** component. All services, the **Presentation** component, or any other components are bundled together:

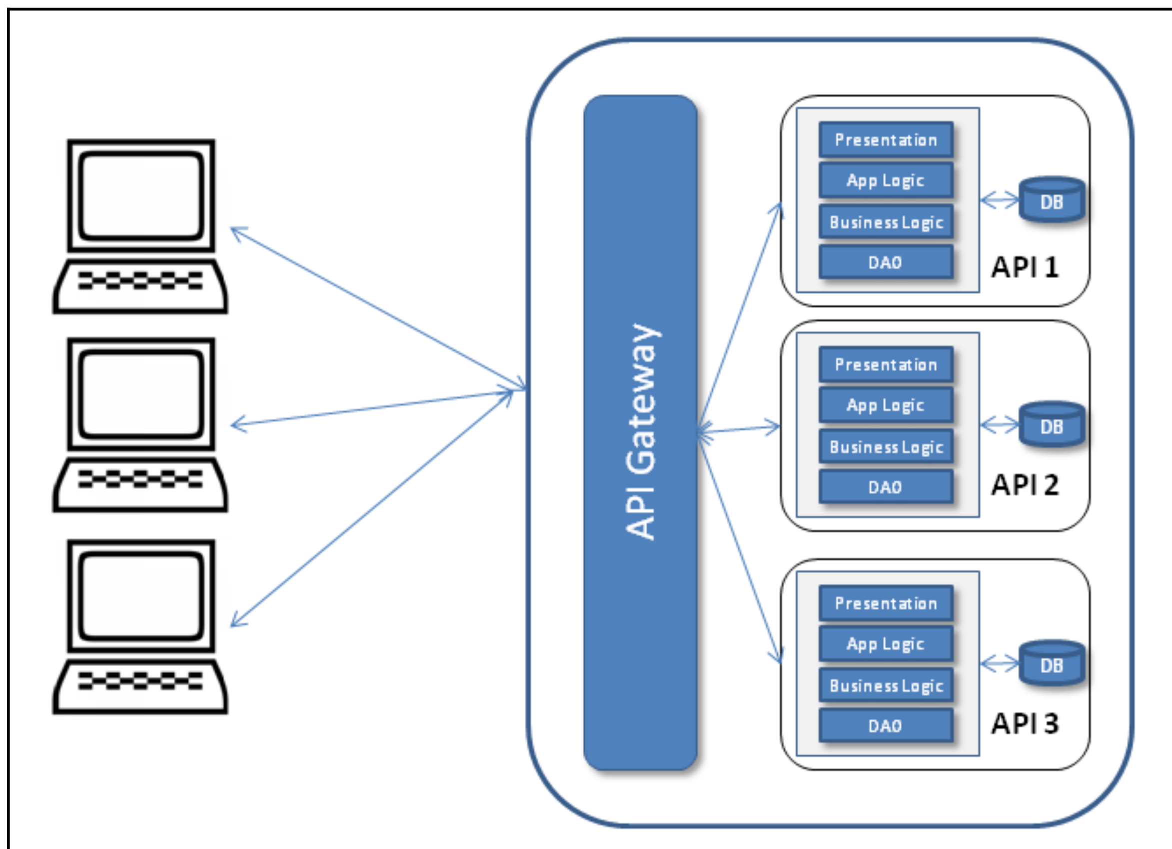


Microservices, nanoservices, teraservices, and serverless

The following diagram depicts the microservices design. Here each component is autonomous. Each component could be developed, built, tested, and deployed independently. Here, even the application **User Interface** (**UI**) component could also be a client and consume the microservices. For the purpose of our example, the layer designed is used within the µService.

The **API Gateway** provides an interface where different clients can access the individual services and solve various problems, such as what to do when you want to send different responses to different clients for the same service. For example, a booking service could send different responses to a mobile client (minimal information) and a desktop client (detailed information), providing different details to each, before providing something different again to a third-party client.

A response may require the fetching of information from two or more services:



After observing all the sample design diagrams we've just gone through, which are very high-level designs, you might find that in a monolithic design, the components are bundled together and tightly coupled. All the services are part of the same bundle. Similarly, in the second design diagram, you can see a variant of the first diagram where all services could have their own layers and form different APIs, but, as shown in the diagram, these are also all bundled together.

Conversely, in the microservices design, the design components are not bundled together and have loose couplings. Each service has its own layers and database, and is bundled in a separate archive to all others. All these deployed services provide their specific APIs, such as Customers or Bookings. These APIs are ready to consume. Even the UI is also deployed separately and designed using µServices. For this reason, the microservices provides various advantages over its monolithic counterpart. I would, nevertheless, remind you that there are some exceptional cases where monolithic application development is highly successful, such as Etsy, and peer-to-peer e-commerce web applications.

Now let us discuss the limitations you'd face while working with Monolithic applications.

One-dimension scalability

Monolithic applications that are large when scaled, scale everything, as all the components are bundled together. For example, in the case of a restaurant table reservation application, even if you would like to scale only the table-booking service, you would scale the whole application; you cannot scale the table-booking service separately. This design does not utilize resources optimally.

In addition, this scaling is one-dimensional. Running more copies of the application provides the scale with increasing transaction volume. An operation team could adjust the number of application copies that were using a load balancer based on the load in a server farm or a cloud. Each of these copies would access the same data source, therefore increasing the memory consumption, and the resulting I/O operations make caching less effective.

Microservices architectures give the flexibility to scale only those services where scale is required and allow optimal utilization of resources. As mentioned previously, when needed, you can scale just the table-booking service without affecting any of the other components. It also allows two-dimensional scaling; here we can not only increase the transaction volume, but also the data volume using caching (platform scale). A development team can then focus on the delivery and shipping of new features, instead of worrying about the scaling issues (product scale).

Microservices could help you scale platforms, people, and product dimensions, as we have seen previously. People scaling here refers to an increase or decrease in team size depending on the microservices' specific development needs.

Microservice development using RESTful web service development provides scalability in the sense that the server-end of REST is stateless; this means that there is not much communication between servers, which makes the design horizontally scalable.

Release rollback in case of failure

Since monolithic applications are either bundled in the same archive or contained in a single directory, they prevent the deployment of code modularity. For example, many of you may have experienced the pain of delaying rolling out the whole release due to the failure of one feature.

To resolve these situations, microservices give us the flexibility to roll back only those features that have failed. It's a very flexible and productive approach. For example, let's assume you are the member of an online shopping portal development team and want to develop an application based on microservices. You can divide your application based on different domains such as products, payments, cart, and so on, and package all these components as separate packages. Once you have deployed all these packages separately, these would act as single components that can be developed, tested, and deployed independently, and called **μService**.

Now, let's see how that helps you. Let's say that after a production release launching new features, enhancements, and bug fixes, you find flaws in the payment service that need an immediate fix. Since the architecture you have used is based on microservices, you can roll back the payment service instead of rolling back the whole release, if your application architecture allows, or apply the fixes to the microservices payment service without affecting the other services. This not only allows you to handle failure properly, but it also helps to deliver the features/fixes swiftly to a customer.

Problems in adopting new technologies

Monolithic applications are mostly developed and enhanced based on the technologies primarily used during the initial development of a project or a product. This makes it very difficult to introduce new technology at a later stage of development or once the product is in a mature state (for example, after a few years). In addition, different modules in the same project that depend on different versions of the same library make this more challenging.

Technology is improving year on year. For example, your system might be designed in Java and then, a few years later, you may want to develop a new service in Ruby on Rails or Node.js because of a business need or to utilize the advantages of new technologies. It would be very difficult to utilize the new technology in an existing monolithic application.

It is not just about code-level integration, but also about testing and deployment. It is possible to adopt a new technology by rewriting the entire application, but it is a time-consuming and risky thing to do.

On the other hand, because of its component-based development and design, microservices architectures give us the flexibility to use any technology, new or old, for development. They do not restrict you to using specific technologies, and give you a new paradigm for your development and engineering activities. You can use Ruby on Rails, Node.js, or any other technology at any time.

So, how is this achieved? Well, it's very simple. Microservices-based application code does not bundle into a single archive and is not stored in a single directory. Each μService has its own archive and is deployed separately. A new service could be developed in an isolated environment and could be tested and deployed without any technical issues. As you know, microservices also own their own separate processes, serving their purpose without any conflicts to do with things such as shared resources with tight coupling, and processes remain independent.

Monolithic systems does not provide flexibility to introduce new technology. However, introduction of new technology comes as low risk features in microservices based system because by default these small and self contained components.

You can also make your microservice available as open source software so it can be used by others, and, if required, it may interoperate with a closed source, a proprietary one, which is not possible with monolithic applications.

Alignment with agile practices

There is no question that monolithic applications can be developed using agile practices, and these are being developed all the time. **Continuous integration (CI)** and **continuous deployment (CD)** could be used, but the question is—do they use agile practices effectively? Let's examine the following points:

- When there is a high probability of having stories dependent on each other, and there could be various scenarios, a story would not be taken up until the dependent story is complete.
- The build takes more time as the code size increases.
- The frequent deployment of a large monolithic application is a difficult task to achieve.
- You would have to redeploy the whole application even if you updated a single component.

- Redeployment may cause problems to already running components; for example, a job scheduler may change whether components impact it or not.
- The risk of redeployment may increase if a single changed component does not work properly or if it needs more fixes.
- UI developers always need more redeployment, which is quite risky and time-consuming for large monolithic applications.

The preceding issues can be tackled very easily by microservices. For example, UI developers may have their own UI component that can be developed, built, tested, and deployed separately. Similarly, other microservices might also be deployable independently and, because of their autonomous characteristics, the risk of system failure is reduced. Another advantage for development purposes is that UI developers can make use of JSON objects and mock Ajax calls to develop the UI, which can be taken up in an isolated manner. After development is finished, developers can consume the actual APIs and test the functionality. To summarize, you could say that microservices development is swift and it aligns well with the incremental needs of businesses.

Ease of development – could be done better

Generally, large monolithic application code is the toughest to understand for developers, and it takes time before a new developer can become productive. Even loading the large monolithic application into an **integrated development environment** (**IDE**) is troublesome, as it makes the IDE slower and the developer less productive.

A change in a large monolithic application is difficult to implement and takes more time due to the large code base, and there can also be a high risk of bugs if impact analysis is not done properly and thoroughly. Therefore, it becomes a prerequisite for developers to do a thorough impact analysis before implementing any changes.

In monolithic applications, dependencies build up over time as all components are bundled together. Therefore, the risk associated with code changes rises exponentially as the amount of modified lines of code grows.

When a code base is huge and more than 100 developers are working on it, it becomes very difficult to build products and implement new features because of the previously mentioned reason. You need to make sure that everything is in place, and that everything is coordinated. A well-designed and documented API helps a lot in such cases.

Netflix, the on-demand internet streaming provider, had problems getting their application developed, with around 100 people working on it. Then, they used a cloud service and broke up the application into separate pieces. These ended up being microservices. Microservices grew from the desire for speed and agility and to deploy teams independently.

Microcomponents are made loosely coupled thanks to their exposed APIs, which can be continuously integration tested. With microservices' continuous release cycle, changes are small and developers can rapidly exploit them with a regression test, then go over them and fix the defects found, reducing the risk of a flawed deployment. This results in higher velocity with a lower associated risk.

Owing to the separation of functionality and the single responsibility principle, microservices make teams very productive. You can find a number of examples online where large projects have been developed with very low team sizes, such as 8 to 10 developers.

Developers can have better focus with smaller code bases and better feature implementation, leading to a higher empathetic relationship with the users of the product. This conduces better motivation and clarity in feature implementation. An empathetic relationship with users allows for a shorter feedback loop and better and speedier prioritization of the feature pipeline. A shorter feedback loop also makes defect detection faster.

Each microservices team works independently and new features or ideas can be implemented without being coordinated with larger audiences. The implementation of endpoint failure handling is also easily achieved in the microservices design.

At a recent conference, a team demonstrated how they had developed a microservices-based transport-tracking application for iOS and Android, within 10 weeks, with Uber-type tracking features. A big consulting firm gave a seven-month estimation for this application to its client. This shows how the microservices design is aligned with agile methodologies and CI/CD.

So far, we have discussed only the microservices design—there are also nanoservices, teraservices, and serverless designs to explore.

Nanoservices

Microservices that are especially small or fine-grained are called nanoservices. A nanoservices pattern is really an **anti-pattern**.

In the case of nanoservices, overheads such as communication and maintenance activities outweigh its utility. Nanoservices should be avoided. An example of a nanoservices (anti-) pattern would be creating a separate service for each database table and exposing its CRUD operation using events or a REST API.

Teraservices

Teraservices are the opposite of microservices. The teraservices design entails a sort of a monolithic service. Teraservices require two terabytes of memory, or more. These services could be used when services are required only to be in memory and have high usage.

These services are quite costly in cloud environments due to the memory needed, but the extra cost can be offset by changing from quad-core servers to dual-core servers.

Such a design is not popular.

Serverless

Serverless is another popular cloud architecture offered by cloud platforms such as AWS. There are servers, but they are managed and controlled by cloud platforms.

This architecture enables developers to simply focus on code and implementing functionality. Developers need not worry about scale or resources (for instance, OS distributions as with Linux, or message brokers such as RabbitMQ) as they would with coded services.

A serverless architecture offers development teams the following features: zero administration, auto-scaling, pay-per-use schemes, and increased velocity. Because of these features, development teams just need to care about implementing functionality rather than the server and infrastructure.