

Book Collection

Learning Path Spring 5: End-To-End Programming

Build enterprise-grade applications using Spring MVC,
Hibernate, and RESTful APIs

Claudio Eduardo de Oliveira, Dinesh Rajput
and Rajesh R V

Packt

www.packt.com



Spring 5: End-To-End Programming

Build enterprise-grade applications using Spring MVC,
Hibernate, and RESTful APIs

Claudio Eduardo de Oliveira
Dinesh Rajput
Rajesh R V

Packt

BIRMINGHAM - MUMBAI

Spring 5: End-To-End Programming

Copyright © 2018 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 authors, 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.

First published: December 2018

Production reference: 1191218

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

ISBN 978-1-78995-966-6

www.packtpub.com



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.packtpub.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.packtpub.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 authors

Claudio Eduardo de Oliveira has been active in software development since 2007, in small, medium, and large companies in Brazil. He has built cloud-centric applications where elasticity, resilience, fault tolerance, and monitoring were necessary for business success. His experience also includes Java and frameworks such as Spring Ecosystem, Guice, and Node.js, among others. Claudio is also a Docker enthusiast and believes it is an important aspect for developers to learn.

Dinesh Rajput is a founder of Dineshonjava (dot) com, a blog for Spring and Java techies. He is a Spring enthusiast and a Pivotal Certified Spring Professional. He has written two bestselling books, Spring 5 Design Patterns and Mastering Spring Boot 2.0. Mastering Spring Boot 2.0 is the Amazon #1 best-selling book on Java. He has more than 10 years of experience with various aspects of Spring and cloud-native development, such as REST APIs and microservice architecture. He is currently working as an architect at a leading company. He has worked as a tech lead at Bennett, Coleman & Co. Ltd, and Paytm. He has a master's degree in computer engineering from JSS Academy of Technical Education, Noida, and lives in Noida with his family.

Rajesh R V is a seasoned IT architect with extensive experience in diversified technologies and more than 18 years of airline IT experience.

He received a degree in computer engineering from the University of Cochin, India, and he joined the JEE community during the early days of EJB. During his course as an architect, he worked on many large-scale, mission-critical projects, including the new generation Passenger Reservation System (iFly Res) and next generation Cargo Reservation System (Skychain, CROAMIS) in the Airline domain.

At present, as a chief architect at Emirates, Rajesh handles the solution architecture portfolio spread across various capabilities, such as JEE, SOA, NoSQL, IoT, cognitive computing, mobile, UI, and integration. At Emirates, the Open Travel Platform (OTP) architected by him earned the group the prestigious 2011 Red Hat Innovation Award in the Carved Out Costs category. In 2011, he introduced the innovative concept of the Honeycomb architecture based on the hexagonal architecture pattern for transforming the legacy mainframe system.

Rajesh has a deep passion for technology and architecture. He also holds several certifications, such as BEA Certified Weblogic Administrator, Sun Certified Java Enterprise Architect, Open Group Certified TOGAF practitioner, Licensed ZapThink Architect in SOA, and IASA global CITA-A Certified Architecture Specialist.

He has written Spring Microservices and reviewed Service-Oriented Java Business Integration by Packt Publishing.

About the reviewers

Paulo Zanco is a solution architect working for Daitan Labs. He is also a system architect with over 25 years of experience at national and international large/middle-sized companies. He has led many complex projects consisting of medium and large teams. He has extensive experience of designing and developing object-oriented and services systems. He is also certified by Sun and Oracle, in J2EE and SOA technologies. He holds a Master's degree in Management Information Systems from Pontifícia Universidade Católica de Campinas.

Rajeev Kumar Mohan has over 17 years of experience in IT, Software Development, and Corporate Training. He has worked for various IT majors like IBM, Pentasoft, Sapient, and Deft Infosystems.

He started career as a programmer and managed various projects. He is subject matter expert in Java, J2EE and related Frameworks, Android, and many UI Technologies. Besides SCJP and SCWCD, Rajeev has completed four masters.

He is Organic Chemistry and Computer Science master MCA and MBA. Rajeev is recruitment consultant and impaneled training consultant for HCL, Amdocs, Steria, TCS, Wipro, Oracle University, IBM, CSC, Genpact , Sapient Infosys and Capgemini.

Rajeev is the founder of Greater Noida based firm SNS Infotech. He also worked for the National Institute Of Fashion Technology [NIFT].

Aditya Abhay Halabe is a full-stack web application developer at Springer Nature's technology division. His primary technology stack includes Scala, Java, micro-web services, NOSQL databases, and multiple frameworks such as Spring, Play, and Angular. He has a very good understanding of Agile development, with continuous integration, which includes knowledge of DevOps, Docker, and automated deployment pipelines.

He is passionate about his work and likes to take on new challenges and responsibilities. Previously, Aditya has worked as a consultant with Oracle, and as a software developer with John Deere.

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
Chapter 1: Journey to the Spring World	8
Spring modularity	9
Spring Core Framework	10
Core container	10
Spring Messaging	11
Spring AMQP	11
Spring for Apache Kafka	12
Spring JMS	13
Spring Web MVC	13
Spring WebFlux	14
Spring Data	15
Spring Security	16
Spring Cloud	18
Spring Cloud Netflix	19
Spring Cloud Config	21
Spring Cloud Consul	22
Spring Cloud Security	22
Spring Cloud Bus	23
Spring Cloud Stream	23
Spring Integration	24
Spring Boot	25
Microservices and Spring Boot	26
Setting up our development environment	27
Installing OpenJDK	28
Installing Maven	28
Installing IDE	29
IntelliJ IDEA	30
Spring Tools Suite	30
Installing Docker	30
Introducing Docker concepts	32
Docker images	32
Containers	33
Docker networks	33
Docker volumes	34
Docker commands	34
Docker run	34
Docker container	35
Docker network	36
Docker volume	37
Summary	37
Chapter 2: Starting in the Spring World: The CMS Application	38

Creating the CMS application structure	39
The CMS project	40
Project metadata section	40
The dependencies section	41
Generating the project	41
Running the application	42
Looking under the hood	46
Running the application	46
IntelliJ IDEA	47
Command line	47
Command line via the Maven goal	47
Command line via the JAR file	48
Creating the REST resources	49
Models	50
Adding Lombok dependency	50
Creating the models	51
Tag	51
Category	51
User	52
News	52
Hello REST resources	53
Creating the CategoryResource class	53
UserResource	55
NewsResource	56
Adding service layer	57
Changes in the model	57
Adding a new review	58
Keeping the news safely	58
Before starting the service layer	58
CategoryService	59
UserService	60
NewsService	61
Configuring Swagger for our APIs	62
Adding dependencies to pom.xml	62
Configuring Swagger	62
First documented API	64
Integrate with AngularJS	68
AngularJS concepts	69
Controllers	69
Services	70
Creating the application entry point	70
Creating the Category Controller	74
Creating the Category Service	75
Summary	76
Chapter 3: Persistence with Spring Data and Reactive Fashion	77
Learning the basics of Docker	78
Preparing MongoDB	78
Preparing a PostgreSQL database	80

Spring Data project	81
Spring Data JPA	82
Configuring pom.xml for Spring Data JPA	82
Configuring the Postgres connections	83
Mapping the models	84
Adding the JPA repositories in the CMS application	85
Configuring transactions	87
Installing and configuring pgAdmin3	89
Checking the data on the database structure	90
Creating the final data access layer	91
Spring Data MongoDB	91
Removing the PostgreSQL and Spring Data JPA dependencies	92
Mapping the domain model	93
Configuring the database connection	94
Adding the repository layer	95
Checking the persistence	96
Creating the Docker image for CMS	99
Configuring the docker-maven-plugin	100
Adding the plugin on pom.xml	100
Pushing the image to Docker Hub	102
Configuring the Docker Spring profile	103
Running the Dockerized CMS	103
Putting in Reactive fashion	104
Reactive Spring	105
Project Reactor	105
Components	105
Hot and cold	108
Reactive types	108
Let's play with the Reactor	108
Spring WebFlux	111
Event-loop model	112
Spring Data for Reactive Extensions	112
Spring Data Reactive	112
Reactive repositories in practice	113
Creating the first Reactive repository	113
Fixing the service layer	114
Changing the CategoryService	114
Changing the REST layer	116
Adding the Spring WebFlux dependency	117
Changing the CategoryResource	118
Summary	120
Chapter 4: Kotlin Basics and Spring Data Redis	121
Learning Kotlin basics	122
Main characteristics of Kotlin	122
Syntax	123
Semantics	124
Declaring functions in Kotlin	124
Simple function with parameters and return type	124

Simple function without return	125
Single expressions functions	125
Overriding a function	125
Data classes	126
Objects	127
Companion objects	128
Kotlin idioms	129
String interpolation	129
Smart Casts	129
Range expressions	130
Simple case	130
The until case	130
The downTo case	131
Step case	131
Null safety	131
Safe calls	132
Elvis operator	132
Wrapping it up	133
Creating the project	133
Project use case	133
Creating the project with Spring Initializr	134
Adding Jackson for Kotlin	135
Looking for the Maven plugins for Kotlin	135
Creating a Docker network for our application	136
Pulling the Redis image from the Docker Hub	137
Running the Redis instance	137
Configuring the redis-cli tool	138
Understanding Redis	138
Data types	139
Strings	139
Main commands	139
Lists	140
Main commands	140
Sets	141
Main commands	142
Spring Data Reactive Redis	142
Configuring the ReactiveRedisConnectionFactory	143
Providing a ReactiveRedisTemplate	143
Creating Tracked Hashtag repository	145
Creating the service layer	146
Exposing the REST resources	147
Creating a Twitter application	147
Configuring pom.xml	150
Creating the image	151
Running the container	152
Testing APIs	153
Summary	154
Chapter 5: Reactive Web Clients	155

Creating the Twitter Gathering project	156
Project structure	156
Starting the RabbitMQ server with Docker	158
Pulling the RabbitMQ image from Docker Hub	158
Starting the RabbitMQ server	158
Spring Messaging AMQP	159
Adding Spring AMQP in our pom.xml	160
Integrating Spring Application and RabbitMQ	160
Understanding RabbitMQ exchanges, queues, and bindings	161
Exchanges	161
Direct exchanges	161
Fanout exchanges	161
Topic exchanges	161
Header exchanges	161
Queues	162
Bindings	162
Configuring exchanges, queues, and bindings on Spring AMQP	162
Declaring exchanges, queues, and bindings in yaml	162
Declaring Spring beans for RabbitMQ	163
Consuming messages with Spring Messaging	165
Producing messages with Spring Messaging	166
Enabling Twitter in our application	166
Producing Twitter credentials	167
Configuring Twitter credentials in application.yaml	167
Modelling objects to represent Twitter settings	167
Twittertoken	168
TwitterAppSettings	168
Declaring Twitter credentials for the Spring container	168
Spring reactive web clients	169
Producing WebClient in a Spring Way	169
Creating the models to gather Tweets	170
Authentication with Twitter APIs	172
Some words about server-sent events (SSE)	173
Creating the gather service	174
Listening to the Rabbit Queue and consuming the Twitter API	175
Changing the Tracked Hashtag Service	177
Adding the Spring Starter RabbitMQ dependency	177
Configuring the RabbitMQ connections	178
Creating exchanges, queues, and bindings for the Twitter Hashtag Service	179
Sending the messages to the broker	180
Testing the microservice's integrations	181
Running Tracked Hashtag Service	182
Running the Twitter Gathering	182
Testing stuff	184
Spring Actuator	185
Adding Spring Boot Actuator in our pom.xml	186

Actuator Endpoints	186
Application custom information	187
Testing endpoints	188
Summary	190
Chapter 6: Playing with Server-Sent Events	191
Creating the Tweet Dispatcher project	192
Using Spring Initializr once again	192
Additional dependencies	193
Server-Sent Events	193
A few words about the HTTP protocol	194
HTTP and persistent connections	195
WebSockets	196
Server-Sent Events	196
Reactor RabbitMQ	197
Understanding the Reactor RabbitMQ	197
Configuring RabbitMQ Reactor beans	198
Consuming the RabbitMQ queues reactively	199
Filtering streams	201
Dockerizing the whole solution	202
Tweet Gathering	202
Tweet Dispatcher	204
Running the containerized solution	205
Running the Tracked Hashtag Service container	206
Running the Tweet Gathering container	208
Running the Tweet Dispatcher container	209
The docker-compose tool	210
Installing docker-compose	211
Creating a docker-compose file	212
Running the solution	214
Testing the network	215
Summary	216
Chapter 7: Airline Ticket System	217
The Airline Ticket System	218
Airline functionalities	218
Solution diagram	220
Spring Cloud Config Server	220
Creating the Config Server project	222
Enabling Spring Cloud Config Server	222
Using GitHub as a repository	223
Configuring the Spring Boot application	223
Configuring the Git repository as a properties source	224
Running the Config Server	224
Testing our Config Server	226
Spring Cloud service discovery	227

Creating Spring Cloud Eureka	229
Creating the Eureka server main class	229
Configuring the Spring Cloud Eureka server	230
Running the Spring Cloud Eureka server	232
Spring Cloud Zipkin server and Sleuth	233
Infrastructure for the Zipkin server	234
Creating the Spring Cloud Zipkin server	236
Configuring bootstrap.yaml and application.yaml	238
Running the Zipkin server	239
Spring Cloud Gateway	241
Creating the Spring Cloud Gateway project	242
Creating the Spring Cloud Gateway main class	243
Configuring the Spring Cloud Gateway project	244
Running the Spring Cloud Gateway	246
Checking the Eureka server	246
Creating our first route with Spring Cloud Gateway	247
Putting the infrastructure on Docker	251
Summary	255
Chapter 8: Putting It All Together	256
The airline Bookings microservice	257
The airline Payments microservice	257
Learning about the Turbine server	257
Creating the Turbine server microservice	259
Hystrix Dashboard	260
Creating the Mail microservice	262
Creating the SendGrid account	263
Creating the Mail microservice project	264
Adding RabbitMQ dependencies	264
Configuring some RabbitMQ stuff	265
Modeling a Mail message	267
The MailSender class	267
Creating the RabbitMQ queue listener	269
Running the Mail microservice	270
Creating the Authentication microservice	271
Creating the Auth microservice	272
Configuring the security	273
Testing the Auth microservice	278
Client credentials flow	278
Implicit grant flow	280
Protecting the microservices with OAuth 2.0	282
Adding the security dependency	282
Configuring the application.yaml file	282
Creating the JwtTokenStore Bean	283
Monitoring the microservices	285
Collecting metrics with Zipkin	285

Collection commands statistics with Hystrix	289
Dockerizing the microservices	291
Running the system	293
Summary	294
Chapter 9: Overview of GOF Design Patterns - Core Design Patterns	295
Introducing the power of design patterns	297
Common GoF Design Pattern overview	298
Creational design patterns	299
Factory design pattern	299
Implementing the Factory design pattern in Spring Framework	300
Sample implementation of the Factory design pattern	301
Abstract factory design pattern	304
Common problems where you should apply the Abstract factory design pattern	304
Implementing the Abstract factory design pattern in the Spring Framework	305
Sample implementation of the Abstract Factory design pattern	305
Singleton design pattern	310
Common problems where you should apply Singleton pattern	311
Singleton design pattern implementation in the Spring Framework	311
Sample implementation of the Singleton design pattern	312
Prototype design pattern	313
Benefits of the Prototype design pattern	313
UML class structure	314
Sample implementation of the Prototype design pattern	315
Builder design pattern	317
Benefits of the Builder pattern:	317
UML class structure	317
Implementing the Builder pattern in the Spring Framework	318
Common problems where you should apply Builder pattern	318
Sample implementation of the Builder design pattern	319
Summary	321
Chapter 10: Consideration of Structural and Behavioral Patterns	322
Examining the core design patterns	323
Structural design patterns	323
The adapter design pattern	324
Benefits of the adapter pattern	324
Common requirements for the adapter pattern	325
Implementation of the adapter design pattern in the Spring Framework	325
Sample implementation of the adapter design pattern	326
The Bridge design pattern	328
Benefits of the Bridge pattern	329
Common problems solved by the Bridge design pattern	329
Implementing the Bridge design pattern in the Spring Framework	329
Sample implementation of the Bridge design pattern	330
Composite design pattern	334

Common problems solved by the composite pattern	335
UML structure of the Composite design pattern	335
Sample implementation of the Composite design pattern	337
Decorator design pattern	339
Common problems solved by the Decorator pattern	340
Implementing the Decorator pattern	343
Decorator design pattern in the Spring Framework	345
Facade Design Pattern	346
Knowing when to use the Facade Pattern	346
Implementing the Facade design pattern	349
The UML structure for the Facade design pattern	350
Facade Pattern in the Spring Framework	351
Proxy design pattern	352
Purpose of the Proxy pattern	352
UML structure for the Proxy design pattern	352
Implementing the Proxy design pattern	353
Proxy pattern in the Spring Framework	354
Behavioral design patterns	354
Chain of Responsibility design pattern	355
Chain of Responsibility pattern in the Spring Framework	356
Command design pattern	356
Command design pattern in the Spring Framework	357
Interpreter Design pattern	358
Interpreter design pattern in the Spring Framework	359
Iterator Design Pattern	359
Iterator design pattern in the Spring Framework	360
Observer pattern in the Spring Framework	363
Template Design Pattern	363
JEE design patterns	364
Summary	366
Chapter 11: Wiring Beans using the Dependency Injection Pattern	367
The dependency injection pattern	369
Solving problems using the dependencies injection pattern	369
Without dependency injection	370
With dependency injection pattern	373
Types of dependency injection patterns	375
Constructor-based dependency injection pattern	375
Setter-based dependency injection	377
Configuring the dependency injection pattern with Spring	380
Dependency injection pattern with Java-based configuration	381
Creating a Java configuration class - AppConfig.java	382
Declaring Spring beans into configuration class	382
Injecting Spring beans	383
Best approach to configure the dependency injection pattern with Java	384
Dependency injection pattern with XML-based configuration	385
Creating an XML configuration file	386
Declaring Spring beans in an XML file	386
Injecting Spring beans	387
Using constructor injection	387

Using setter injection	388
Dependency injection pattern with Annotation-based configuration	390
What are Stereotype annotations?	391
Creating auto searchable beans using Stereotype annotations	392
Searching beans using component scanning	394
Annotating beans for autowiring	397
Using @Autowired with setter method	398
Using @Autowired with the fields	398
The Autowiring DI pattern and disambiguation	399
Resolving disambiguation in Autowiring DI pattern	400
Implementing the Abstract Factory Pattern in Spring (FactoryBean interface)	401
Implementation of FactoryBean interface in Spring	402
Sample implementation of FactoryBean interface	402
Best practices for configuring the DI pattern	404
Summary	406
Chapter 12: Accessing a Database with Spring and JDBC Template Patterns	407
The best approach to designing your data-access	408
The resource management problem	411
Implementing the template design pattern	412
Problems with the traditional JDBC	413
Solving problems with Spring's JdbcTemplate	414
Configuring the data source and object pool pattern	416
Configuring a data source using a JDBC driver	417
Configuring the data source using pool connections	419
Implementing the Builder pattern to create an embedded data source	421
Abstracting database access using the DAO pattern	422
The DAO pattern with the Spring Framework	423
Working with JdbcTemplate	424
When to use JdbcTemplate	424
Creating a JdbcTemplate in an application	425
Implementing a JDBC-based repository	425
Jdbc callback interfaces	427
Creating a RowMapper class	427
Implementing RowCallbackHandler	428
Implementing ResultSetExtractor	429
Best practices for Jdbc and configuring JdbcTemplate	431
Summary	432
Chapter 13: Improving Application Performance Using Caching Patterns	433
What is cache?	434
Where do we use caching?	434
Understanding cache abstraction	435

Enabling caching via the Proxy pattern	436
Enabling the caching proxy using Annotation	437
Enabling the Caching Proxy using the XML namespace	438
Declarative Annotation-based caching	439
The @Cacheable annotation	439
The @CachePut annotation	440
Customizing the cache key	441
Conditional caching	442
The @CacheEvict annotation	443
The @Caching annotation	444
The @CacheConfig annotation	444
Declarative XML-based caching	445
Configuring the cache storage	448
Setting up the CacheManager	448
Third-party cache implementations	449
Ehcache-based cache	449
XML-based configuration	450
Creating custom caching annotations	451
Top caching best practices to be used in a web application	452
Summary	454
Chapter 14: Demystifying Microservices	455
Evolution of microservices	455
Business demand as a catalyst for microservices evolution	456
Technology as a catalyst for microservices evolution	459
Imperative architecture evolution	459
What are Microservices?	460
Microservices - The honeycomb analogy	464
Principles of microservices	465
Single responsibility per service	465
Microservices are autonomous	466
Characteristics of microservices	467
Services are first class citizens	467
Characteristics of service in a microservice	468
Microservices are lightweight	469
Microservices with polyglot architecture	470
Automation in microservices environment	471
Microservices with a supporting ecosystem	472
Microservices are distributed and dynamic	472
Antifragility, fail fast, and self healing	474
Microservices examples	475
An example of a holiday portal	475
An example of a travel agent portal	478
Microservices benefits	481
Supports polyglot architecture	481

Enables experimentation and innovation	482
Elastically and selectively scalable	483
Allows substitution	485
Enables to build organic systems	486
Helps managing technology debt	487
Allowing co-existence of different versions	488
Supporting building self-organizing systems	490
Supporting event-driven architecture	491
Enables DevOps	492
Summary	492
Chapter 15: Related Architecture Styles and Use Cases	493
Service-Oriented Architecture (SOA)	494
Service-oriented integration	495
Legacy modernization	496
Service-oriented application	496
Monolithic migration using SOA	497
Twelve-Factor Apps	498
Single code base	499
Bundle dependencies	499
Externalizing configurations	500
Backing services are addressable	500
Isolation between build, release, and run	501
Stateless, shared nothing processes	502
Expose services through port bindings	502
Concurrency for scale out	502
Disposable, with minimal overhead	503
Development, production parity	503
Externalizing logs	503
Package admin processes	504
Serverless computing	504
Lambda architecture	506
DevOps, Cloud, and Containers	508
DevOps as the practice and process for microservices	509
Cloud and Containers as the self-service infrastructure for microservices	509
Reactive microservices	510
A reactive microservice-based order management system	512
Microservice use cases	514
Microservices early adopters - Is there a common theme?	516
Monolithic migration as the common use case	518
Microservice frameworks	519
Summary	521
Chapter 16: Building Microservices with Spring Boot	522
Setting up a development environment	523

Spring Boot for building RESTful microservices	523
Getting started with Spring Boot	524
Developing a Spring Boot microservice	525
Developing our first Spring Boot microservice	526
Testing Spring Boot microservice	534
HATEOAS-enabled Spring Boot microservice	535
Reactive Spring Boot microservices	541
Reactive microservices using Spring WebFlux	541
Understanding Reactive Streams	545
Publisher	545
Subscriber	545
Subscription	545
Processor	546
Reactive microservices using Spring Boot and RabbitMQ	546
Implementing security	549
Securing a microservice with basic security	550
Securing microservice with OAuth2	550
Enabling cross origin for microservices interactions	553
Spring Boot actuators for microservices instrumentation	555
Monitoring using JConsole	557
Monitoring using ssh	557
Adding a custom health module	558
Building custom metrics	560
Documenting microservices	561
Putting it all together - Developing a customer registration microservice example	562
Summary	575
Chapter 17: Microservices Capability Model	576
Microservices capability model	577
Core capabilities	578
Service listeners and libraries	578
Storage capability	579
Service implementation	579
Service endpoint	580
Infrastructure capabilities	580
Cloud	581
Container runtime	581
Container orchestration	582
Supporting capabilities	582
Service gateway	583
Software-defined load balancer	583
Central log management	583
Service discovery	584
Security service	585

Service configuration	585
Ops monitoring	585
Dependency management	586
Data lake	587
Reliable messaging	588
Process and governance capabilities	588
DevOps	589
Automation tools	589
Container registry	590
Microservice documentation	591
Reference architecture and libraries	592
Microservices maturity model	592
Level 0 - Traditional	593
Level 1 - Basic	594
Level 2 - Intermediate	594
Level 3 - Advanced	595
Entry points for adoption	596
Summary	597
Chapter 18: Scale Microservices with Spring Cloud Components	598
What is Spring Cloud?	599
Spring Cloud releases	600
Setting up the environment for the BrownField PSS	600
Spring Cloud Config	601
Building microservices with Config Server	604
Setting up the Config Server	605
Understanding the Config Server URL	607
Accessing the Config Server from clients	609
Handling configuration changes	613
Spring Cloud Bus for propagating configuration changes	613
Setting up high availability for the Config Server	614
Monitoring Config Server health	616
Config Server for configuration files	616
Completing changes to use Config Server	617
Eureka for registration and discovery	617
Understanding dynamic service registration and discovery	618
Understanding Eureka	619
Setting up the Eureka Server	621
High availability for Eureka	626
Zuul proxy as the API Gateway	629
Setting up Zuul	630
High availability of Zuul	635
High availability of Zuul when the client is also a Eureka Client	636
High availability when client is not a Eureka Client	637
Completing Zuul for all other services	638

Streams for reactive microservices	638
Protecting microservices with Spring Cloud Security	643
Summarising the BrownField PSS architecture	644
Summary	647
Chapter 19: Logging and Monitoring Microservices	648
Understanding log management challenges	649
Centralized logging solution	650
Selection of logging solutions	652
Cloud services	652
Off-the-shelf solutions	653
Best of the breed integration	653
Log shippers	653
Log stream processors	654
Log storage	654
Dashboards	654
Custom logging implementation	654
Distributed tracing with Spring Cloud Sleuth	658
Monitoring microservices	661
Monitoring challenges	662
Monitoring tools	665
Monitoring microservice dependency	666
Spring Cloud Hystrix for fault-tolerant microservices	667
Aggregate Hystrix streams with Turbine	671
Data analysis using Data Lake	675
Summary	676
Chapter 20: Scaling Dockerized Microservices with Mesos and Marathon	678
Scaling microservices	678
Understanding autoscaling	680
The missing pieces	681
Container orchestration	682
Why is container orchestration important	682
What does container orchestration do?	683
Relationship with microservices	686
Relationship with virtualization	687
Container orchestration solutions	687
Docker Swarm	688
Kubernetes	689
Apache Mesos	689
HashiCorp Nomad	690
CoreOS Fleet	691
Container orchestration with Mesos and Marathon	691
Mesos in details	692
Mesos architecture	692

Table of Contents

Marathon	695
Implementing Mesos and Marathon with DCOS	696
Implementing Mesos and Marathon for BrownField microservices	697
Installing Mesos, Marathon, and related components	697
Running Mesos and Marathon	698
Preparing BrownField PSS services	700
Deploying BrownField PSS services	701
Summary	704
Other Books You May Enjoy	706
Index	709

Preface

When it comes to developing applications for the web or enterprises, the Spring Framework has become one of the most popular choices among Java developers. With an array of tools and features, Spring offers developers experience that is rivaled by none.

Spring 5: End-to-End Programming is all about leveraging these features and developing your own business applications with hands-on examples. You will create applications of increasing complexity, such as a CMS app, a messaging app, and a real-world microservice. While developing these applications, you will learn about Project Reactor in Spring, Spring Webflux, Spring Data, and Angular. You will also develop these applications using standard design patterns in Spring, helping you to solve common design problems with ease.

By the end of this Learning Path, you will be well equipped to develop enterprise applications on the web using Spring 5.

This Learning Path includes content from the following Packt products:

- Spring 5.0 By Example by Claudio Eduardo de Oliveira
- Spring 5 Design Patterns by Dinesh Rajput
- Spring 5.0 Microservices by Rajesh R V

Who this book is for

If you're a developer starting out with Spring, then this learning path will help you understand new Spring 5.0 framework concepts followed by their implementation in Java and Kotlin. If you are an experienced Spring developer, then this Learning Path will enable you to gain insight into the new Spring 5.0 features.

What this book covers

Chapter 1, *Journey to the Spring World*, will guide you through the main concepts of Spring Framework. Here we learn to setup the environment by installing OpenJDK, Maven, IntelliJ IDEA, and Docker. By the end, we will create our first Spring application.

Chapter 2, *Starting in the Spring World - the CMS Application*, will begin by getting our hands dirty with Spring Initializr to create configurations for our CMS application. We will then learn how to create REST resources, add the service layer and finally integrate with AngularJS.

Chapter 3, *Persistence with Spring Data and Reactive Fashion*, will build upon our CMS application created in the previous chapter. Here we will learn how to persist data on a real database by learning about Spring Data MongoDB and PostgreSQL. We will finally learn about Project Reactor which will help you to create a non-blocking application in the JVM ecosystem.

Chapter 4, *Kotlin Basics and Spring Data Redis*, will give you a basic introduction to Kotlin while presenting the benefits of the language. We will then learn how to use Redis which will be used as a message broker using the publish-subscribe feature.

Chapter 5, *Reactive Web Clients*, will teach you how to use the Spring Reactive Web Client and make HTTP calls in a reactive fashion. We will also be introduced to RabbitMQ and Spring Actuator.

Chapter 6, *Playing with Server-Sent Events*, will help you develop an application which will filter tweets by text content. We will accomplish this by consuming the tweeter stream using Server-Sent Events which is a standard way to send data streams from a server to clients.

Chapter 7, *Airline Ticket System*, will teach you to use Spring Messaging, WebFlux, and Spring Data components to build a airline ticket system. You will also learn about circuit breakers and OAuth in this chapter. By the end, we will create a system with many microservices to ensure scalability.

Chapter 8, *Putting It All Together*, will bring the entire book into perspective while also teaching you about the Turbine server. We will also look into the Hystrix Dashboard to monitor our.

Chapter 9, *Overview of GOF Design Patterns - Core Design Patterns*, gives an overview of the Core Design Pattern of the GoF Design Patterns family, including some best practices for an application design. You'll also get an overview of the common problems solving with design patterns.

Chapter 10, *Consideration of Structural and Behavioral Patterns*, gives an overview of the Structural and Behavioural Design Pattern of the GoF Design Patterns family, including some best practices for an application design. You'll also get an overview of the common problem solving with design patterns.

Chapter 11, *Wiring Beans using the Dependency Injection Pattern*, explores dependency injection pattern and detail about the configuration of Spring in an application, showing you various ways of configurations in your application. This includes a configuration with XML, Annotation, Java, and Mix.

Chapter 12, *Accessing a Database with Spring and JDBC Template Patterns*, explores how to access the data with Spring and JDBC; here, you'll see how to use Spring's JDBC abstraction and JDBCTemplate to query relational databases in a way that is far simpler than native JDBC.

Chapter 13, *Improving Application Performance Using Caching Patterns*, shows how to improve application performance by avoiding the database altogether if the data needed is readily available. So, I will show you how Spring provides support for caching data.

Chapter 14, *Demystifying Microservices*, gives you an introduction to microservices. This chapter covers the background, evaluation, and fundamental concepts of microservices.

Chapter 15, *Related Architecture Styles and Use Cases*, covers the relationship with Service-Oriented Architecture, the concepts of cloud native and Twelve Factor applications, and explains some of the common use cases.

Chapter 16, *Building Microservices with Spring Boot*, introduces building REST and message-based microservices using the Spring Framework and how to wrap them with Spring Boot. In addition, we will also explore some core capabilities of Spring Boot.

Chapter 17, *Microservices Capability Model*, introduces a capability model required to successfully manage a microservices ecosystem. This chapter also describes a maturity assessment model, which will be useful when adopting microservices at enterprise level.

Chapter 18, *Scale Microservices with Spring Cloud Components*, shows you how to scale previous examples using Spring Cloud stack capabilities. It details the architecture and different components of Spring Cloud and how they integrate together.

Chapter 19, *Logging and Monitoring Microservices*, covers the importance of logging and monitoring aspects when developing microservices. Here, we look at the details of some of the best practices when using microservices, such as centralized logging and monitoring capabilities using open source tools and how to integrate them with Spring projects.

Chapter 20, *Scaling Dockerized Microservices with Mesos and Marathon*, explains auto-provisioning and deployment of microservices. Here, we will also learn how to use Docker containers in the preceding example for large-scale deployments.

To get the most out of this book

The readers are expected to have a basic knowledge of Java. Notion about Distributed Systems is an added advantage. To execute code files in this book, you would need to have the following software/dependencies:

- IntelliJ IDEA Community Edition
- Docker CE
- pgAdmin
- Docker Compose

You will also need your favorite IDE for the examples, but I have also used the Software Spring Tool Suite; download the latest version of Spring Tool Suite (STS) from <https://spring.io/tools/sts/all> according to your system OS. The Java 8 and STS work on a variety of platforms--Windows, macOS, and Linux.

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/Learning-Spring-5-End-to-End-Programming>. 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!

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: "Some common annotations are `@Service`, `@Component`, `@Bean`, and `@Configuration`."

A block of code is set as follows:

```
http
.formLogin()
.loginPage("/login")
.failureUrl("/login?error")
.and()
.authorizeRequests()
.antMatchers("/signup", "/about").permitAll()
.antMatchers("/admin/**").hasRole("ADMIN")
.anyRequest().authenticated();
```

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

```
sudo apt-get install openjdk-8-jdk -y  
java -version
```

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: "In the **Project Metadata** section, we can put the coordinates for Maven projects."



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 Journey to the Spring World

Spring is an open source modular framework for the JVM platform. A framework is a collection of libraries whose primary goal is to address common software development problems. The framework should solve these problems in a generic form.

Rod Johnson created the Spring Framework in 2002 together with his book publication, which was called *Expert One-on-One J2EE Design and Development*. The idea behind the creation of the framework was to tackle the complexities of Java Enterprise Edition.

At that time, this kind of solution-focused a lot on the details of the infrastructure, and a developer using the solution would spend a lot of time writing code to solve infrastructural problems. Since its creation, one of Rod Johnson's primary concerns has been to increase developer productivity.

The framework was first seen as a lightweight container for Java Runtime Environment, and it became popular in the community, especially because of the dependency injection feature. The framework made dependency injection incredibly easy. Developers hadn't seen such a feature before, and as a consequence, people the world over adopted the project. Year by year, its popularity within the software development world has been increasing.

In the earliest versions, the framework had to work with the XML file to configure the container. At the time, this was so much better than J2EE applications, where it was necessary to create many Ant files to create the boilerplate classes and interfaces.

The framework was always seen as an advanced technology for the Java platform, but in 2014, the Spring team launched the Spring Boot platform. This platform was incredibly successful in the Java Enterprise ecosystem, and it changed the way in which developers built Java Enterprise applications.

Today, Spring is the *de facto* framework for Java development, and companies around the world use it in their systems. The community is vibrant and contributes to development in different ways, such as opening issues, adding the code, and discussing the framework in the most important Java conferences around the world. Let's look at and play with the famous framework for Java developers.

We will cover the following topics in this chapter:

- Main modules of the Spring Framework
- Spring annotations for each module
- Setting up the development environment
- Docker and Docker commands

Spring modularity

Since its foundation, the framework has had a particular focus on modularity. It is an important framework characteristic because it makes the framework an excellent option for different architectural styles and different parts of applications.

It means the framework is not an opinionated, full-stack framework that dictates the rules to make everything work. We can use the framework as we need and integrate it with a wide range of specification and third-party libraries.

For example, for portal web applications, the Spring MVC supports features such as template engines and REST endpoints and integrates them with the popular JavaScript framework, AngularJS.

Also, if the application needs support for a distributed system, the framework can supply an amazing module called Spring Cloud, which has some essential features for distributed environments, such as service registration and discovery, a circuit breaker, intelligent routing, and client-side load balancing.

Spring makes the development applications for Java Runtime easy with different languages, such as Java, Kotlin, and Groovy (with which you can choose the flavor and make the development task fun).

It is divided into various modules. The main modules are as follows:

- Spring Core
- Spring Data

- Spring Security
- Spring Cloud
- Spring Web-MVC

In this book, we will cover the most common solutions involved in Java Enterprise applications, including the awesome Spring Cloud project. Also, we can find some interesting projects such as Spring Batch and Spring Integration, but these projects are for specific needs.

Spring Core Framework

This module is the base of the framework and contains the essential support for dependency injection, web features supported by Spring MVC (**model-view-controller**) and the pretty new WebFlux frameworks, and aspect-oriented programming. Also, this module supports the foundation for JDBC, JMS, JPA and a declarative way to manage transactions. We will explore it and understand the main projects of this module. So let's do it!

Core container

The core container is the basis of the whole Spring ecosystem and comprehends four components—core, beans, context, and expression language.

Core and beans are responsible for providing the fundamentals of the framework and dependency injection. These modules are responsible for managing the IoC container, and the principal functions are the instantiation, configuration, and destruction of the object residents in the Spring container.



Spring contexts are also called Spring IoC containers, which are responsible for instantiating, configuring, and assembling beans by reading configuration metadata from XML, Java annotations, and/or Java code in the configuration files.

There are two critical interfaces inside these modules—`BeanFactory` and `ApplicationContext`. The `BeanFactory` takes care of the bean lifecycle, instantiating, configuring, managing, and destroying, and the `ApplicationContext` helps developers to work with files resources in a generic way, enable to publish events to registered listeners. Also, the `ApplicationContext` supports internationalization and has the ability to work with messages in different Locales.

These modules help the context component to provide a way to access the objects inside the container. The context component has the `ApplicationContext` interface with the essential class for the container.



Some common annotations are `@Service`, `@Component`, `@Bean`, and `@Configuration`.

Spring Messaging

Spring Framework supports a wide range of messaging systems. The Java platform is recognized as providing excellent support for messaging applications, and Spring Framework follows this approach and offers a variety of projects to help developers to write powerful applications with more productivity and fewer lines of infrastructure code. The basic idea of these projects is to provide some template classes that have the convenience methods to interact with the messaging systems.

Also, the project supplies some listener annotations to provide support for listening to messages from the brokers. The framework maintains the standard for different projects. In general, the prefix of the annotations is the name of the messaging system, for example, `@KafkaListener`.

The framework supplies many abstractions to create messaging applications in a generic way. This is interesting stuff because the application requirements change during the application lifecycle and the message broker solution may change as well. Then, with small changes, the application built with the Spring message module can work in different brokers. This is the goal.

Spring AMQP

This subproject supports the AMQP protocol in Spring Framework. It provides a template to interact with the message broker. A template is like a super high-level API that supports the `send` and `receive` operations.

There are two projects in this set: `spring-amqp`, which can be used for ActiveMQ for instance, and `spring-rabbit`, which adds support for the RabbitMQ broker. This project enables broker administration through the APIs to declare queues, bindings, and exchanges.

These projects encourage the extensive use of dependency injection provided by the core container, because they make the configuration more declarative and easy to understand.

Nowadays, the RabbitMQ broker is the popular choice for the messaging applications, and Spring provides full support for client interactions up to the level of administration tasks.



Some common annotations are `@Exchange` and `@QueueBinding`.

Spring for Apache Kafka

Spring for Apache Kafka supports the broker-based Apache Kafka applications. It provides a high-level API to interact with Apache Kafka. Internally, the projects use the Kafka Java APIs.

This module supports the annotation programming model. The basic idea is that with a couple of annotations and some POJO models, we can bootstrap the application and start listening to and producing messages.

`KafkaTemplate` is a central class of this project. It enables us to send messages to Apache Kafka with a high-level API. Asynchronous programming is supported as well.

This module offers support for transactions via annotations. This feature is enabled via standard transactional annotations used in Spring-based applications, such as `@Transactional`.

We also learned about Spring AMQP. This project adds the Spring concept of creating applications based on this broker. The dependency injection features are supported as well.



Some common annotations are `@EnableKafka` and `@KafkaListener`.

Spring JMS

The idea of this project provides a JMS integration with ideas of Spring Framework projects and supplies a high-level API to interact with brokers. The worst part of a JMS specification is that it has a lot of boilerplate code to manage and close connections.

The `JmsTemplate` is a central class for this module, and it enables us to send messages to the broker. The JMS specification has a lot of intrinsic behaviors to handle the creation and releases resources, for instance, the `JmsTemplate` class do this tasks automatically for developers.

The module also supports transactional requirements. The `JmsTransactionManager` is the class that handles the transactional behavior of the Spring JMS module.

Spring removes the boilerplate code with a couple of annotations. The framework increases the readability of the code and makes the code more intuitive as well.

Some common annotations are `@JmsListener` and `@EnableJms`.



Spring Web MVC

This module is the first one built by the Spring Team to support the web applications in Spring Framework. This module uses the Servlet API as its foundation, and then these web applications must follow the Servlet Specification and be deployed into servlet containers. In version 5.0, the Spring Team created a Reactive web framework, which will be covered later in this book.

The Spring Web MVC module was developed using the front controller pattern. When the framework was created, this pattern was a common choice for many frameworks, such as Struts and JSF, among others. Under the hood, there is the main servlet in Spring called `DispatcherServlet`. This servlet will redirect through an algorithm to do the desired work.

It enables developers to create amazing web applications on the Java platform. This portion of the framework provides full support to develop this kind of application. There are some interesting features for this purpose, such as support for internationalization and support for handling cookies. Also, multipart requests are an exciting feature for when the application needs to handle upload files and support routing requests.

These characteristics are common for most web applications, and the framework has excellent support for these features. This support makes the framework a good choice for this kind of application. In [Chapter 2, Starting in the Spring World - The CMS Application](#), we will create an application using this module and the main features will be explored in depth.

The module has full support for annotation programming since to declare HTTP endpoints until to wrap the request attribute in an HTTP request. It makes the application extremely readable without the boilerplate code to get the request parameter, for example.

Web application-wise, it enables developers to work with robust template engines such as Thymeleaf and Freemarker. It is entirely integrated with routing features and bean validation.

Also, the framework allows developers to build REST APIs with this module. Given all of this support, the module has become a favorite in the Spring ecosystem. Developers have started to create APIs with this stack, and some important companies have started to use it, especially given that the framework provides an easy way to navigate through the annotations. Because of this, the Spring Team added the new annotation `@RestController` in version 4.0.

We will work a lot with this module. Chapter by chapter, we will learn interesting things about this part of the framework.



Some common annotations are `@RequestMapping`, `@Controller`, `@Model`, `@RestController`, and `@RequestBody`.

Spring WebFlux

A new module introduced in Spring 5.0, Spring WebFlux, can be used to implement web applications built with Reactive Streams. These systems have nonblocking characteristics and are deployed in servers built on top of Netty, such as Undertow and servlet containers that support + 3.1.



Netty is an open source framework that helps developers to create network applications—that is, servers and clients using the asynchronous, event-driven pattern. Netty provides some interesting advantages, such as lower latency, high throughput, and less resource consumption. You can find more information at <https://netty.io>.

This module supports annotations based on Spring MVC modules, such as `@GetMapping`, `@PostMapping`, and others. This is an important feature that enables us to migrate to this new version. Of course, some adjustments are necessary, such as adding Reactor classes (Mono or Flux).

This module meets the modern web requirements to handle a lot of concurrent channels where the thread-per-request model is not an option.

We will learn about this module in [Chapter 3, Adding Persistence with Spring Data and Putting it into Reactive Fashion](#) and implement a fully Reactive application based on Reactive Streams.



Some common annotations are `@RequestMapping`, `@RestController`, and `@RequestBody`.

Spring Data

Spring Data is an interesting module that provides the easiest way to manage application data with Spring-based programming. The project is an umbrella project, with subprojects to support different databases technologies, even relational and nonrelational databases. The Spring Team supports some databases technologies, such as Apache Cassandra, Apache Solr, Redis, and JPA Specification, and the community maintains the other exciting projects, such as ElasticSearch, Aerospike, DynamoDb, and Couchbase. The full list of projects can be found at <http://projects.spring.io/spring-data>.

The goal is to remove the boilerplate code from the persistence code. In general, the data access layer is quite similar, even in different projects, differing only in the project model, and Spring Data provides a powerful way to map the domain model and repository abstraction.

There are some central interfaces; they're a kind of marker to instruct the framework to choose the correct implementation. Under the hood, Spring will create a proxy and delegate the correct implementation. The amazing thing here is that developers don't have to write any persistence code and then take care of this code; they simply choose the required technology and Spring takes care of the rest.

The central interfaces are `CrudRepository` and `PagingAndSortingRepository`, and their names are self-explanatory. `CrudRepository` implements the CRUD behaviors, such as `create`, `retrieval`, `update`, and `delete`. `PagingAndSortingRepository` is an extension of `CrudRepository` and adds some features such as paging and sorting. Usually, we will find derivations of these interfaces such as `MongoRepository`, which interacts with MongoDB database technology.



Some common annotations are `@Query`, `@Id`, and `@EnableJpaRepositories`.

Spring Security

Security for Java applications was always a pain for developers, especially in Java Enterprise Edition. There was a lot of boilerplate code to look up objects in the application servers, and the security layer was often heavily customized for the application.

In that chaotic scenario, the Spring Team decided to create a Spring Security project to help developers handle the security layer on the Java application.

In the beginning, the project had extensive support for Java Enterprise Edition and integration with EJB 3 security annotations. Nowadays, the project supports many different ways to handle authorization and authentication for Java applications.

Spring Security provides a comprehensive model to add authorization and authentication for Java applications. The framework can be configured with a couple of annotations, which makes the task of adding a security layer extremely easy. The other important characteristics concern how the framework can be extended. There are some interfaces that enable developers to customize the default framework behaviors, and it makes the framework customized for different application requirements.

It is an umbrella project, and it is subdivided into these modules:

- spring-security-core
- spring-security-remoting
- spring-security-web
- spring-security-config
- spring-security-ldap
- spring-security-acl
- spring-security-cas
- spring-security-openid
- spring-security-test

These are the main modules, and there are many other projects to support a wide range of types of authentication. The module covers the following authentication and authorization types:

- LDAP
- HTTP Basic
- OAuth
- OAuth2
- OpenID
- CAAS
- JAAS

The module also offers a **domain-specific language (DSL)** to provide an easy configuration. Let's see a simple example:

```
http
    .formLogin()
        .loginPage("/login")
        .failureUrl("/login?error")
        .and()
    .authorizeRequests()
        .antMatchers("/signup", "/about").permitAll()
        .antMatchers("/admin/**").hasRole("ADMIN")
        .anyRequest().authenticated();
```



The example was extracted from the [spring.io blog](https://spring.io/blog/2013/07/11/spring-security-java-config-preview-readability/). For more details, go to <https://spring.io/blog/2013/07/11/spring-security-java-config-preview-readability/>.

As we can see, the DSL makes the configuration task extremely easy and very understandable.

Spring Security's main features are as follows:

- Session management
- Protection against attacks (CSRF, session fixation, and others)
- Servlet API integration
- Authentication and authorization

We will learn more about Spring Security in Chapter 8, *Circuit Breakers and Security*. We will also put it into practice.



`@EnableWebSecurity` is a common annotation.

Spring Cloud

Spring Cloud is another umbrella project. The primary goal of this project is to help developers create distributed systems. Distributed systems have some common problems to solve and, of course, a set of patterns to help us, such as service discovery, circuit breakers, configuration management, intelligent route systems, and distributed sessions. Spring Cloud tools have all these implementations and well-documented projects.

The main projects are as follows:

- Spring Cloud Netflix
- Spring Cloud Config
- Spring Cloud Consul
- Spring Cloud Security
- Spring Cloud Bus
- Spring Cloud Stream

Spring Cloud Netflix

Spring Cloud Netflix is perhaps the most popular Spring module nowadays. This fantastic project allows us to integrate the Spring ecosystem with the Netflix OSS via Spring Boot AutoConfiguration features. The supported Netflix OSS libraries are Eureka for service discovery, Ribbon to enable client-side load balancing, circuit breaker via Hystrix to protect our application from external outages and make the system resilient, the Zuul component provides an intelligent routing and can act as an edge service. Finally, the Feign component can help developers to create HTTP clients for REST APIs with a couple of annotations.

Let's look at each of these:

- **Spring Cloud Netflix Eureka:** The focus of this project is to provide service discovery for applications while conforming to Netflix standards. Service discovery is an important feature and enables us to remove hardcoded configurations to supply a hostname and ports; it is more important in cloud environments because the machine is ephemeral, and thus it is hard to maintain names and IPs. The functionality is quite simple, the Eureka server provides a service registry, and Eureka clients will contact its registers themselves.



Some common annotations are `@EnableEurekaServer` and `@EnableEurekaClient`.

- **Spring Cloud Feign:** The Netflix team created the Feign project. It's a great project that makes the configuration of HTTP clients for REST applications significantly easier than before. These implementations are based on annotations. The project supplies a couple of annotations for HTTP paths, HTTP headers, and much more, and of course, Spring Cloud Feign integrates it with the Spring Cloud ecosystem through the annotations and autoconfiguration. Also, Spring Cloud Feign can be combined with the Eureka server.



Some common annotations are `@EnableFeignClients` and `@FeignClient`.

- **Spring Cloud Ribbon:** Ribbon is a client-side load balancer. The configuration should mainly provide a list of servers for the specific client. It must be named. In Ribbon terms, it is called the **named client**. The project also provides a range of load-balancing rules, such as Round Robin and Availability Filtering, among others. Of course, the framework allows developers to create custom rules. Ribbon has an API that works, integrated with the Eureka server, to enable service discovery, which is included in the framework. Also, essential features such as fault tolerance are supported because the API can recognize the running servers at runtime.



Some common annotations are `@RibbonClient` and `@LoadBalanced`.

- **Spring Cloud Hystrix:** An acclaimed Netflix project, this project provides a circuit breaker pattern implementation. The concept is similar to an electrical circuit breaker. The framework will watch the method marked with `@HystrixCommand` and watch for failing calls. If the failed calls number more than a figure permitted in configuration, the circuit breaker will open. While the circuit is open, the fallback method will be called until the circuit is closed and operates normally. It will provide resilience and fault-tolerant characteristics for our systems. The Spring ecosystem is fully integrated with Hystrix, but it works only on the `@Component` and `@Service` beans.



Some common annotations are `@EnableCircuitBreaker` and `@HystrixCommand`.

Spring Cloud Config

This exciting project provides an easy way to manage system configurations for distributed systems, and this is a critical issue in cloud environments because the file system is ephemeral. It also helps us to maintain different stages of the deployment pipeline. Spring profiles are fully integrated with this module.

We will need an application that will provide the configuration for other applications. We can understand its workings by thinking of the concepts of the **server** and the **client**, the server will provide some configurations through HTTP and the client will look up the configuration on the server. Also, it is possible to encrypt and decrypt property values.

There are some storage implementations to provide these property files, and the default implementation is Git. It enables us to store our property files in Git, or we can use the file system as well. The important thing here is that the source does not matter.



Git is a distributed version control. The tool is commonly used for development purposes, especially in the open-source community. The main advantage, when you compare it to some market players, such as SVN, is the *distributed architecture*.

There is an interesting integration between **Spring Cloud Bus** and this module. If they are integrated, it is possible to broadcast the configuration changes on the cluster. This is an important feature if the application configuration changes with frequency. There are two annotations that tell Spring to apply changes at runtime: `@RefreshScope` and `@ConfigurationProperties`.

In Chapter 7, *Airline Ticket System*, we will implement an exciting service to provide external configurations for our microservices using this module. Server concepts will be explained in more detail. The client details will be presented as well.



`@EnableConfigServer` is a common annotation.

Spring Cloud Consul

Spring Cloud Consul provides integrations with Hashicorp's Consul. This tool addresses problems in the same way as service discovery, a distributed configuration, and control bus. This module allows us to configure Spring applications and Consul with a few annotations in a Spring-based programming model. Autoconfiguration is supported as well. The amazing thing here is that this module can be integrated with some Netflix OSS libraries, such as Zuul and Ribbon, via Spring Cloud Zuul and Spring Cloud Ribbon respectively (for example).

`@EnableDiscoveryClient` is a common annotation.



Spring Cloud Security

This module is like an extension from Spring Security. However, distributed systems have different requirements for security. Normally, they have central identity management, or the authentication lies with the clients in the case of REST APIs. Normally, in distributed systems, we have microservices, and these services might have more than one instance in the runtime environment whose characteristics make the authentication module slightly different from monolithic applications. The module can be used together with Spring Boot applications and makes the OAuth2 implementation very easy with a couple of annotations and a few configurations. Also, some common patterns are supported, such as single sign-on, token relay, and token exchange.

For the microservice applications based on the Spring Cloud Netflix, it is particularly interesting because it enables downstream authentication to work with a Zuul proxy and offers support from Feign clients. An interceptor is used to fetch tokens.

Some commons annotations are `@EnableOAuth2Sso` and `@EnableResourceServer`.



Spring Cloud Bus

The main goal of this project is to provide an easy way to broadcast changes spread throughout the cluster. The applications can connect the distributed system nodes through the message broker.

It provides an easy way for developers to create a publish and subscribe mechanism using the `ApplicationContext` provided by Spring Container. It enables the possibility to create applications using the event-driven architecture style with the Spring Ecosystem.

To create custom events, we need to create a child class from `RemoteApplicationEvent` and mark the class to be scanned via `@RemoteApplicationEventScan`.

The projects support three message brokers as the transport layer:

- AMQP
- Apache Kafka
- Redis

`@RemoteApplicationEventScan` is a common annotation.



Spring Cloud Stream

The idea behind this module is to provide an easy way to build message-driven microservices. The module has an opinionated way of configuration. It means we need to follow some rules to create these configurations. In general, the application is configured by the `yml|properties` file.

The module supports annotations as well. This means that a couple of annotations are enough to create consumers, producers, and bindings; it decouples the application and makes it easy to understand. It supplies some abstractions around the message brokers and channels, and it makes the developer's life more comfortable and productive as well.

Spring Cloud Stream has Binder implementations for RabbitMQ and Kafka.



Some common annotations are `@EnableBinding`, `@Input`, and `@Output`.

Spring Integration

This module supports a lot of Enterprise Application patterns and brings the Spring programming model to this topic. The Spring programming model enables extensive dependence injection support and is annotations programming-centric. The annotations instruct us as to how the framework needs to be configured and defines framework behaviors.

The POJO model is suggested because it is simple and widely known in the Java development world.

This project has some intersections with the other modules. Some other projects use these module concepts to do their work. There is a project called Spring Cloud Stream, for instance.

The Enterprise Integration patterns are based on a wide range of communication channels, protocols, and patterns. This project supports some of these.

The modules support a variety of features and channels, such as the following:

- Aggregators
- Filters
- Transformers
- JMS
- RabbitMQ
- TCP/UDP
- Web services
- Twitter
- Email
- And much more

There are three main concepts of Enterprise application integration:

- Messages
- Message channel
- Message endpoint

Finally, the Spring Integration module offers a comprehensive way to create application integration and enables developers to do it using amazing support.



Some common annotations are `@EnableIntegration`,
`@IntegrationComponentScan`, and `@EnablePublisher`.

Spring Boot

Spring Boot was released in 2014. The idea behind this project was to present a way to deploy the web application outside of any container, such as Apache Tomcat, Jetty, and so on. The benefit of this kind of deployment is the independence from any external service. It allows us to run the web applications with one JAR file. Nowadays, this is an excellent approach because this forms the most natural way to adopt DevOps culture.

Spring Boot provides embedded servlet containers, such as Apache Tomcat, Jetty, and Undertow. It makes the development process more productive and comfortable when testing our web applications. Also, customizations during configuration are allowed via a configuration file, or by providing some beans.

There are some advantages when adopting the Spring Boot framework. The framework does not require any XML for configuration. This is a fantastic thing because we will find all the dependencies in the Java files. This helps the IDEs to assist developers, and it improves the traceability of the code. Another important advantage is that the project tries to keep the configuration as automatic as possible. Some annotations make the magic happen. The interesting thing here is that Spring will inject the implementation of any code that is generated at runtime.

The Spring Boot framework also provides interesting features to help developers and operations, such as health checks, metrics, security, and configuration. This is indispensable for modern applications where the modules are decomposed in a microservices architecture.

There are some other interesting features that can help the developers DevOps-wise. We can use the `application-{profile}.properties` or `application.yaml` files to configure different runtime profiles, such as development, testing, and production. It is a really useful Spring Boot feature.

Also, the project has full support for the tests, since the web layer up to the repository layer.

The framework provides a high-level API to work with unit and integration tests. Also, the framework supplies many annotations and helpers classes for developers.

The Spring Boot project is a production-ready framework with default optimized configurations for the web servers, metrics, and monitoring features to help the development team deliver high-quality software.

We can develop applications by coding in the Groovy and Java languages. Both are JVM languages. In version 5.0, the Spring Team announced the full support for Kotlin, the new language for JVM. It enables us to develop consistent and readable codes. We will look at this feature in depth in [Chapter 7, Airline Ticket System](#).

Microservices and Spring Boot

The microservices architectural style, in general, is distributed, must be loosely coupled, and be well-defined. These characteristics must be followed when you want a microservices architecture.

Much of Spring Boot is aimed at developer productivity by making common concepts, such as RESTful HTTP and embedded web application runtimes, easy to wire up and use. In many respects, it also aims to serve as a *micro-framework*, by enabling developers to pick and choose the parts of the framework they need, without being overwhelmed by bulky or otherwise unnecessary runtime dependencies. This also enables Boot applications to be packaged into small units of deployment, and the framework is able to use build systems to generate those deployables as runnable Java archives.

The main characteristics of microservices are:

- Small-grained components
- Domain responsibility (orders, shopping carts)

- Programming-language agnostic
- Database agnostic

Spring Boot enables us to run an application on embedded web servers such as Tomcat, Jetty, and Undertow. This makes it extremely easy to deploy our components because it is possible to expose our HTTP APIs in one JAR.

The Spring Team even thinks in terms of developer productivity, and they offer a couple of projects called **starters**. These projects are groups of dependencies with some compatibilities. These awesome projects additionally work with the convention over configuration. Basically, they are common configurations that developers need to make on every single project. We can change these settings in our `application.properties` or `application.yaml` files.

Another critical point for microservices architecture is monitoring. Let's say that we're working on an e-commerce solution. We have two components, shopping cart and payments. The shopping cart probably needs to have several instances and payments need to have fewer instances. How can we check these several instances? How can we check the health of these services? We need to fire an alarm when these instances go down. This is a common implementation for all services. The Spring Framework supplies a module called Spring Boot Actuator that provides some built-in health checks for our application, databases, and much more.

Setting up our development environment

Before we start, we need to set up our development environment. Our development environment consists of the following four tools:

- JDK
- Build tool
- IDE
- Docker

We will install JDK version 8.0. This version is fully supported in Spring Framework 5. We will present the steps to install Maven 3.3.9, the most famous build tool for Java development, and in the last part, we will show you some detailed instructions on how to install IntelliJ IDEA Community Edition. We will use Ubuntu 16.04, but you can use your favorite OS. The installation steps are easy.

Installing OpenJDK

OpenJDK is a stable, free, and open source Java development kit. This package will be required for everything related to code compilation and runtime environments.

Also, it is possible to use an Oracle JDK, but you should pay attention to the **License and Agreements**.

To install OpenJDK, we will open a terminal and run the following command:

```
sudo apt-get install openjdk-8-jdk -y
```



We can find more information on how to install Java 8 JDK in the installation section (<http://openjdk.java.net/install/>) of the OpenJDK page.

Check the installation using the following command:

```
java -version
```

You should see the OpenJDK version and its relevant details displayed as follows:

```
ubuntu@ubuntu-xenial:~$ java -version
openjdk version "1.8.0_131"
OpenJDK Runtime Environment (build 1.8.0_131-8u131-b11-2ubuntu1.16.04.3-b11)
OpenJDK 64-Bit Server VM (build 25.131-b11, mixed mode)
ubuntu@ubuntu-xenial:~$
```

Now that we have installed the Java development kit, we are ready for the next step. In the real world, we must have a build tool to help developers to compile, package, and test the Java applications.

Let's install Maven in the next section.

Installing Maven

Maven is a popular build tool for Java development. Some important open source projects were built using this tool. There are features that facilitate the build process, standardize the project structure, and provide some guidelines for best practices development.

We will install Maven, but the installation step should be executed after the OpenJDK installation.

Open a terminal and execute the following:

```
sudo apt-get install maven -y
```

Check the installation using this command:

```
mvn -version
```

You should see the following output, although the version may be different for you:

```
ubuntu@ubuntu-xenial:~$ mvn -version
Apache Maven 3.3.9
Maven home: /usr/share/maven
Java version: 1.8.0_131, vendor: Oracle Corporation
Java home: /usr/lib/jvm/java-8-openjdk-amd64/jre
Default locale: en_US, platform encoding: ANSI_X3.4-1968
OS name: "linux", version: "4.4.0-97-generic", arch: "amd64", family: "unix"
ubuntu@ubuntu-xenial:~$ █
```

Well done. Now we have Maven installed. Maven has a vibrant community that produces many plugins to help developers with important tasks. There are plugins to execute a unit test and plugins to prepare the project for the release event that can be integrated with SCM software.

We will use the `spring boot maven plugin` and `docker maven plugin`. The first converts our application to a JAR file and the second enables us to integrate with Docker Engine to create images, run containers, and much more. In the next few chapters, we will learn how to configure and interact with these plugins.

Installing IDE

The IDE is an important tool to help developers. In this book, we will use the IntelliJ IDEA as an *official* tool for developing our projects. There are no restrictions for other IDEs because the project will be developed using Maven as a build tool.

The IDE is a personal choice for developers, and in general, it involves passion; what some people love, other developers hate. Please feel free to use your favorite.

IntelliJ IDEA

IntelliJ IDEA is a JetBrains product. We will use the Community Edition, which is open source and a fantastic tool with which to code Java and Kotlin. The tool offers a fantastic autocomplete feature, and also fully supports Java 8 features.

Go to <https://www.jetbrains.com/idea/download/#section=linux> and download the Community Edition. We can extract the `.tar.gz` and execute it.

Spring Tools Suite

The Spring Tools Suite is based on Eclipse IDE, provided by the Eclipse Foundation, of course. The goal is to provide support for the Spring ecosystem and make the developer's life easier. Interesting features such as Beans Explorer are supported in this tool.

Download the tool at the following link:

http://download.springsource.com/release/STS/3.6.4.RELEASE/dist/e4.4/groovy-grails-tool-suite-3.6.4.RELEASE-e4.4.2-linux-gtk-x86_64.tar.gz

Installing Docker

Docker is an open source project that helps people to run and manage containers. For developers, Docker helps in different stages of the development lifecycle.

During the development phase, Docker enables developers to spin up different infrastructure services such as databases and service discoveries like Consul without installation in the current system operational. It helps the developers because developers do not need to install these kinds of systems in the operating system layer. Usually, this task can cause conflicts with the libraries during the installation process and consumes a lot of time.

Sometimes, developers need to install the exact version. In this case, it is necessary to reinstall the whole application on the expected version. It is not a good thing because the developer machine during this time becomes slow. The reason is quite simple, there are many applications that are used during software development.

Docker helps developers at this stage. It is quite simple to run a container with MongoDB. There is no installation and it enables developers to start the database with one line. Docker supports the image tag. This feature helps to work with different versions of the software; this is awesome for developers who need to change the software version every time.

Another advantage is that when the developers need to deliver the artifacts for test or production purposes, Docker enables these tasks via Docker images.

Docker helps people to adopt the DevOps culture and delivers amazing features to improve the performance of the whole process.

Let's install Docker.

The easiest way to install Docker is to download the script found at <https://get.docker.com>:

```
curl -fsSL get.docker.com -o get-docker.sh
```

After the download is completed, we will execute the script as follows:

```
sh get-docker.sh
```

Wait for the script execution and then check the Docker installation using the following command:

```
docker -v
```

The output needs to look like the following:

```
ubuntu@ubuntu-xenial:~$ docker -v
Docker version 17.10.0-ce, build f4ffd25
ubuntu@ubuntu-xenial:~$
```



Sometimes, the version of Docker can be increased, and the version should be at least **17.10.0-ce**.

Finally, we will add the current user to the Docker group, and this enables us to use the Docker command line without the `sudo` keyword. Type the following command:

```
sudo usermod -aG docker $USER
```

We need to log out to effect these changes. Confirm whether the command works as expected by typing the following. Make sure that the `sudo` keyword is not present:

```
docker ps
```

The output should be as follows:

```
ubuntu@ubuntu-xenial:~$ docker ps
CONTAINER ID        IMAGE               COMMAND       CREATED          STATUS          PORTS          NAMES
ubuntu@ubuntu-xenial:~$ █
```

Introducing Docker concepts

Now, we will introduce some Docker concepts. This book is not about Docker, but some basic instructions on how to use Docker are necessary to interact with our containers during the next few chapters. Docker is a de facto tool that is used to manage containers.

Docker images

The Docker image is like a template for a Docker container. It contains a set of folders and files that are necessary to start the Docker container. We will never have an image in execution mode. The image provides a template for Docker Engine to start up the container. We can create an analogy with object orientation to understand the process better. The image is like a class that provides an *infrastructure* to instantiate some objects, and instances are like a container.

Also, we have a Docker registry to store our images. These registries can be public or private. Some cloud vendors provide these private registries. The most famous is Docker Hub. It can be free, but if you choose this option, the image should be public. Of course, Docker Hub supports private images, but in this case, you have to pay for the service.

Containers

Docker containers are a *lightweight* virtualization. The term *lightweight* means that Docker uses the SO functionalities to cage the system process and manager memory, processors, and folders. This is different from virtualization with VMs because, in this mode, the technology needs to simulate the whole SO, drivers, and storage. This task consumes a lot of computational power and can sometimes be inefficient.

Docker networks

A Docker network is a layer that provides runtime isolation for containers. It is a kind of sandbox in which to run containers that are isolated from other containers. When the Docker is installed, by default it creates three networks that should not be removed. These three networks are as follows:

- bridge
- none
- host

Also, Docker provides the user with an easy way to create your network. For this purpose, Docker offers two drivers—**bridge** and **overlay**.

Bridge can be used for the local environment, and it means this kind of network is allowed on a single host. It will be useful for our applications because it promotes isolation between containers regarding security. This is a good practice. The name of the container attached to this kind of network can be used as a **DNS** for the container. Internally, Docker will associate the container name with the container IP.

The overlay network provides the ability to connect containers to different machines. This kind of network is used by Docker Swarm to manage the container in a clustered environment. In the newest version, the Docker Compose tool natively supports Docker Swarm.

Docker volumes

Docker volumes are the suggested way to persist data outside of a container. These volumes are fully managed by Docker Engine, and these volumes can be writable and readable depending on the configuration when they are used with a Docker command line. The data of these volumes is persisted on a directory path on a host machine.

There is a command-line tool to interact with volumes. The base of this tool is the `docker volume` command; the `--help` argument on the end shows the help instructions.

Docker commands

Now we will take a look at Docker commands. These commands are used mainly in the development life cycle, commands such as `spin up` container, `stop` containers, `remove`, and `inspect`.

Docker run

`docker run` is the most common Docker command. This command should be used to start the containers. The basic structure of a command is as follows:

```
docker run [OPTIONS] IMAGE[:TAG|@DIGEST] [COMMAND] [ARG...]
```

The options arguments enable some configurations for the container, for instance, the `--name` argument permits you to configure a name for a container. It is important for DNS when the container is running in a bridge network.

The network settings can be configured on the `run` command as well, and the parameter is `-- net`. This enables us to configure the network to which the container will be attached.

Another important option is `detached`. It indicates whether the container will run in the background. The `-d` parameter instructs Docker to run a container in the background.

Docker container

The `docker container` command permits you to manage the containers. There are many commands, as shown in the following list:

- `docker container attach`
- `docker container commit`
- `docker container cp`
- `docker container create`
- `docker container diff`
- `docker container exec`
- `docker container export`
- `docker container inspect`
- `docker container kill`
- `docker container logs`
- `docker container ls`
- `docker container pause`
- `docker container port`
- `docker container prune`
- `docker container rename`
- `docker container restart`
- `docker container rm`
- `docker container run`
- `docker container start`
- `docker container stats`
- `docker container stop`
- `docker container top`
- `docker container unpause`
- `docker container update`
- `docker container wait`

There are some important commands here. The `docker container exec` permits you to run commands on a running container. This is an important task to debug or look inside the container files. The `docker container prune` removes the stopped containers. It is helpful in the development cycle. There are some known commands, such as `docker container rm`, `docker container start`, `docker container stop`, and `docker container restart`. These commands are self-explanatory and have similar behaviors.

Docker network

The `docker network` commands enable you to manage the Docker network stuff via the command line. There are six basic commands, and the commands are self-explanatory:

- `docker network create`
- `docker network connect`
- `docker network ls`
- `docker network rm`
- `docker network disconnect`
- `docker network inspect`

`docker network create`, `docker network ls`, and `docker network rm` are the main commands. It is possible to compare them with the Linux commands, where the `rm` command is used to remove things and the `ls` command is usually used to list things such as folders. The `create` command should be used to create networks.

The `docker network connect` and `docker network disconnect` commands allow you to connect the running container to the desired network. They may be useful in some scenarios.

Finally, the `docker network inspect` command provides detailed information on the requested network.

Docker volume

The `docker volume` command permits you to manage the Docker volumes via the command-line interface. There are five commands:

- `docker volume create`
- `docker volume inspect`
- `docker volume ls`
- `docker volume prune`
- `docker volume rm`

The `docker volume create`, `docker volume rm` and `docker volume ls` commands are effectively used to manage the `docker volume` by Docker Engine. The behaviors are quite similar to those of the networks, but for volumes. The `create` command will create a new volume with some options allowed. The `ls` command lists all volumes and the `rm` command will remove the requested volume.

Summary

In this chapter, we looked at the main concepts of Spring Framework. We understood the main modules of the framework and how these modules can help developers to build applications in different kinds of architecture, such as messaging applications, REST APIs, and web portals.

We also spent some time preparing our development environment by installing essential tools, such as Java JDK, Maven, and IDE. This was a critical step to take before we continue to the next chapters.

We used Docker to help us to set up a development environment, such as containers for databases and delivery for our application in Docker images. We installed Docker and looked at the main commands for managing containers, networks, and volumes.

In the next chapter, we will create our first Spring application and put it into practice!

2

Starting in the Spring World: The CMS Application

Now, we'll create our first application; at this point, we have learned the Spring concepts, and we are ready to put them into practice. At the beginning of this chapter, we'll introduce the Spring dependencies to create a web application, also we know that Spring Initializr is a fantastic project that enables developers to create Spring skeleton projects, with as many dependencies as they want. In this chapter, we will learn how to put up our first Spring application on IDE and command line, expose our first endpoint, understand how this works under the hood, and get to know the main annotations of Spring REST support. We will figure out how to create a service layer for the **CMS (Content Management System)** application and understand how Dependency Injection works in a Spring container. We will meet the Spring stereotypes and implement our first Spring bean. At the end of this chapter, we will explain how to create a view layer and integrate that with AngularJS.

In this chapter, the following topics will be covered:

- Creating the project structure
- Running the first Spring application
- Introducing the REST support
- Understanding the Dependency Injection in Spring

Creating the CMS application structure

Now we will create our first application with the Spring Framework; we will create a basic structure for the CMS application with Spring Initializr. This page helps to bootstrap our application, it's a kind of guide which allows us to configure the dependencies on Maven or Gradle. We can also choose the language and version of Spring Boot.

The page looks like this:

The screenshot shows the Spring Initializr interface. At the top, it says "SPRING INITIALIZR bootstrap your application now". Below that, there's a search bar with the placeholder "Generate a [Maven Project] with [Java] and Spring Boot [1.5.8]".

The main area is divided into two sections:

- Project Metadata**: Contains fields for "Group" (com.example) and "Artifact" (demo).
- Dependencies**: A search bar with suggestions like "Web, Security, JPA, Actuator, Devtools..." and a "Selected Dependencies" section.

At the bottom center is a large green button labeled "Generate Project". Below it, a note says "Don't know what to look for? Want more options? [Switch](#) to the full version." At the very bottom, it says "start.spring.io is powered by [Spring Initializr](#) and [Pivotal Web Services](#)".

In the **Project Metadata** section, we can put the coordinates for Maven projects; there is a group field which refers to the `groupId` tag, and we have artifacts which refer to the `artifactId`. This is all for the Maven coordinates.

The dependencies section enables the configuration of the Spring dependencies, the field has the autocomplete feature and helps developers to put in the correct dependency.

The CMS project

Before we start to code and learn amazing things, let's understand a little bit about the CMS project, the main purpose of this project is to help companies manage the CMS content for different topics. There are three main entities in this project:

- The `News` class is the most important, it will store the content of the news.
- It has a *category* which makes the search easier, and we can also group news by category, and of course, we can group by the user who has created the news. The news should be approved by other users to make sure it follows the company rules.
- The news has some *tags* as well, as we can see the application is pretty standard, the business rules are easy as well; this is intentional because we keep the focus on the new things we will learn.

Now we know how Spring Initializr (<https://start.spring.io>) works and the business rules we need to follow, we are ready to create the project. Let's do it right now.

Project metadata section

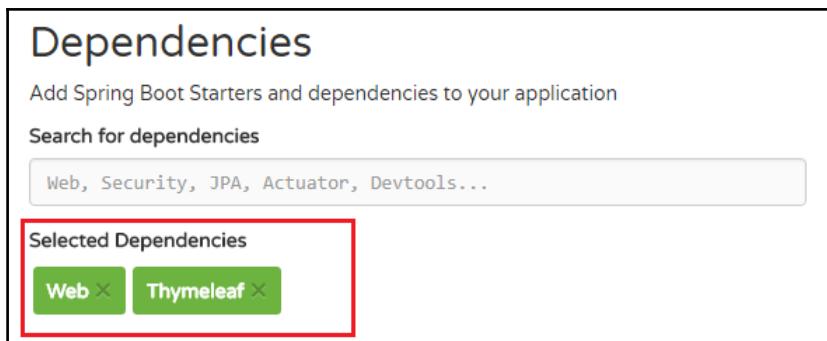
Insert `spring-five` in the **Group** field and `cms` in the **Artifact** field. If you want to customize it, no problem, this is a kind of informative project configuration:

The screenshot shows the 'Project Metadata' section of the Spring Initializr interface. It includes fields for 'Artifact coordinates', 'Group' (containing 'spring-five'), and 'Artifact' (containing 'cms').

Project Metadata	
Artifact coordinates	
Group	spring-five
Artifact	cms

The dependencies section

Type the **MVC** word in the **Search for Dependencies** field. The Web module will appear as an option, the Web module contains the full-stack web development with Embedded Tomcat and Spring MVC, select it. Also, we need to put Thymeleaf dependencies in this module. It is a template engine and will be useful for the view features at the end of this chapter. Type **Thymeleaf**, it includes the Thymeleaf templating engine, and includes integration with Spring. The module will appear, and then select it as well. Now we can see **Web** and **Thymeleaf** in the **Selected Dependencies** pane:



Generating the project

After we have finished the project definition and chosen the project dependencies, we are ready to download the project. It can be done using the **Generate Project** button, click on it. The project will be downloaded. At this stage, the project is ready to start our work:



The zip file will be generated with the name `cms.zip` (the **Artifact** field input information) and the location of the downloaded file depends on the browser configuration.



Before opening the project, we must uncompress the artifact generated by **Spring Initializr** to the desired location. The command should be: `unzip -d <target_destination> /<path_to_file>/cms.zip`. Follow the example: `unzip -d /home/john /home/john/Downloads/cms.zip`.

Now, we can open the project in our IDE. Let's open it and take a look at the basic structure of the project.

Running the application

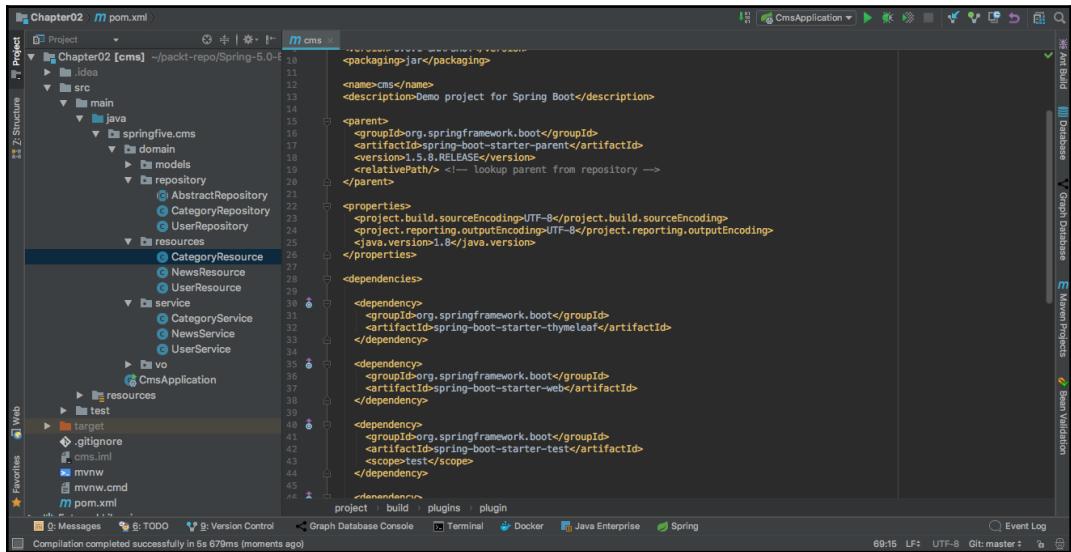
Before we run the application, let's have a walk through our project structure.

Open the project on IntelliJ IDEA using the **Import Project** or **Open** options (both are similar), the following page will be displayed:



Then we can open or import the `pom.xml` file.

The following project structure should be displayed:



Open the `pom.xml`, we have three dependencies, `spring-boot-starter-thymeleaf`, `spring-boot-starter-web`, `spring-boot-starter-test`, and an interesting plugin, `spring-boot-maven-plugin`.

These starter dependencies are a shortcut for developers because they provide full dependencies for the module. For instance, on the `spring-boot-starter-web`, there is `web-mvc`, `jackson-databind`, `hibernate-validator-web`, and some others; these dependencies must be on the classpath to run the web applications, and starters make this task considerably easier.

Let's analyze our `pom.xml`, the file should look like this:

```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>spring-five</groupId>
  <artifactId>cms</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>

```

```
<name>cms</name>
<description>Demo project for Spring Boot</description>

<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.5.8.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>

<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
    <java.version>1.8</java.version>
</properties>

<dependencies>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-thymeleaf</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>

    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>1.16.16</version>
        <scope>provided</scope>
    </dependency>

    <dependency>
        <groupId>io.springfox</groupId>
        <artifactId>springfox-swagger2</artifactId>
        <version>2.7.0</version>
    </dependency>
```

```
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>2.7.0</version>
</dependency>

</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>

</project>
```

Also, we have a `spring-boot-maven-plugin`, this awesome plugin provides Spring Boot support for Maven. It enables you to package the application in a Fat-JAR, and the plugin supports the run, start, and stop goals, as well interacting with our applications.



Fat-JAR: a JAR which contains all project class files and resources packed together with all its dependencies.

For now, that is enough on Maven configurations; let's take a look at the Java files.

The Spring Initializr created one class for us, in general, the name of this class is artifact name plus `Application`, in our case `CmsApplication`, this class should look like this:

```
package springfive.cms;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class CmsApplication {

    public static void main(String[] args) {
        SpringApplication.run(CmsApplication.class, args);
    }
}
```

Looking under the hood

We have some interesting things here, let's understand them. The `@SpringBootApplication` is the essential annotation for the Spring Boot application; it's a kind of alias for `@Configuration`, `@EnableAutoConfiguration`, and `@Component` annotations. Let's dig in:

- The first annotation, `@Configuration` indicates that the class can produce a beans definitions for the Spring container. This is an interesting annotation to work with external dependencies such as `DataSources`; this is the most common use case for this annotation.
- The second annotation, `@EnableAutoConfiguration` means that with the `Spring ApplicationContext` container, it will try to help us configure the default beans for the specific context. For instance, when we create the web MVC application with Spring Boot, we will probably need a web server container to run it. In a default configuration, the Spring container, together with `@EnableAutoConfiguration`, will configure a bean Tomcat-embedded container for us. This annotation is very helpful for developers.
- The `@Component` is a stereotype, the container understands which class is considered for auto-detection and needs to instantiate it.

The `SpringApplication` class is responsible for bootstrapping the Spring application from the main method, it will create an `ApplicationContext` instance, take care of configurations provided by the configuration files, and finally, it will load the singleton beans that are defined by annotations.



Stereotype Annotations denote a conceptual division in an architecture layer. They help the developers understand the purpose of the class and the layer which the beans represent, for example, `@Repository` means the data access layer.

Running the application

We will run the application in IntelliJ IDEA and command line. It is an important task to learn because we are working in different development environments; sometimes the configurations of the application are a little bit complicated, and we are not able to run it with IDEs, or sometimes the companies have different IDEs as standard, so we will learn about two different ways.

IntelliJ IDEA

In general, the IntelliJ IDEA recognizes the main class annotated with `@SpringBootApplication` and creates a run configuration for us, but it depends on the version of the tool, let's do it.

Command line

The command line is a more generic tool to run the project. Also, this task is easy, thanks to the Spring Boot Maven plugin. There are two ways to run, and we will cover both.

Command line via the Maven goal

The first one is a goal of the Spring Boot Maven plugin, and it is straightforward; open the terminal then go to the root project folder, pay attention as this is the same folder where we have the `pom.xml`, and execute the following command:

```
mvn clean install spring-boot:run
```

The Maven will now compile the project and run the main class, the class `CmsApplication`, and we should see this output:

Command line via the JAR file

To run it through the Java file, we need to compile and package it, and then we can run the project with the Java command line. To compile and package it, we can use the pretty standard Maven command like this:

```
mvn clean install
```

After the project is compiled and packaged as a Fat-JAR, we can execute the JAR file, go to the target folder and check the files from this folder, probably the result will look like this:

```
classes/           generated-sources/   generated-test-sources/ maven-archiver/      maven-status/      surefire-reports/      test-classes/
ubuntu@ubuntu-xenial:/vagrant/cms$ cd target/
ubuntu@ubuntu-xenial:/vagrant/cms/target$ ls -l
total 20808
drwxr-xr-x 1 ubuntu ubuntu    128 Oct 28 16:44 classes
-rw-r--r-- 1 ubuntu ubuntu 21301788 Oct 28 16:44 cms-0.0.1-SNAPSHOT.jar
-rw-r--r-- 1 ubuntu ubuntu    2745 Oct 28 16:44 cms-0.0.1-SNAPSHOT.jar.original
drwxr-xr-x 1 ubuntu ubuntu     96 Oct 28 16:44 generated-sources
drwxr-xr-x 1 ubuntu ubuntu     96 Oct 28 16:44 generated-test-sources
drwxr-xr-x 1 ubuntu ubuntu     96 Oct 28 16:44 maven-archiver
drwxr-xr-x 1 ubuntu ubuntu     96 Oct 28 16:44 maven-status
drwxr-xr-x 1 ubuntu ubuntu    128 Oct 28 16:44 surefire-reports
drwxr-xr-x 1 ubuntu ubuntu     96 Oct 28 16:44 test-classes
ubuntu@ubuntu-xenial:/vagrant/cms/target$
```

We have two main files in our target folder, the `cms-0.0.1-SNAPSHOT.jar` and the `cms-0.0.1-SNAPSHOT.jar.original`, the file with the `.original` extension is not executable. It is the original artifact resulting from the compilation, and the other is our executable file. It is what we are looking for, let's execute it, type the following command:

```
java -jar cms-0.0.1-SNAPSHOT.jar
```

The result should be as displayed. The application is up and running:

```

ubuntu@ubuntu-xenial:/vagrant/cms/target$ java -jar cms-0.0.1-SNAPSHOT.jar
[...]
:: Spring Boot ::      (v1.5.8.RELEASE)

2017-10-28 16:56:44.189 INFO 19419 --- [           main] springfive.cms.CmsApplication      : Starting CmsApplication v0.0.1-SNAPSHOT on ubuntu-xenial with PID 19419 (/vagrant/cms/target/cms-0.0.1-SNAPSHOT.jar started by ubuntu in /vagrant/cms/target)
2017-10-28 16:56:44.211 INFO 19419 --- [           main] springfive.cms.CmsApplication      : No active profile set, falling back to default profiles: default
2017-10-28 16:56:46.074 INFO 19419 --- [           main] o.s.w.e.AnnotationConfigEmbeddedWebApplicationContext$5a2e4553: startup date [Sat Oct 28 16:56:46 UTC 2017]; root of context hierarchy
2017-10-28 16:56:50.690 INFO 19419 --- [           main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat initialized with port(s): 8080 (http)
2017-10-28 16:56:50.740 INFO 19419 --- [           main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2017-10-28 16:56:50.752 INFO 19419 --- [           main] o.apache.catalina.core.StandardEngine : Starting Servlet Engine: Apache Tomcat/8.5.23
2017-10-28 16:56:51.133 INFO 19419 --- [ost-startStop-1] o.a.c.c.c.Tomcat$localhost [/]      : Initializing Spring embedded WebApplicationContext
2017-10-28 16:56:51.133 INFO 19419 --- [ost-startStop-1] o.s.w.context.ContextLoader      : Root WebApplicationContext: initialization completed in 5061 ms
2017-10-28 16:56:51.617 INFO 19419 --- [ost-startStop-1] o.s.b.w.servlet.ServletRegistrationBean : Mapping servlet 'dispatcherServlet' to [/]
2017-10-28 16:56:51.644 INFO 19419 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean : Mapping filter: 'characterEncodingFilter' to: [//*]
2017-10-28 16:56:51.644 INFO 19419 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean : Mapping filter: 'hiddenHttpMethodFilter' to: [*]
2017-10-28 16:56:51.644 INFO 19419 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean : Mapping filter: 'httpPutFormContentFilter' to: [*]
2017-10-28 16:56:51.644 INFO 19419 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean : Mapping filter: 'requestContextFilter' to: [*]
2017-10-28 16:56:51.644 INFO 19419 --- [           main] s.w.s.m.m.RequestMappingHandlerMapping : Looking for @ControllerAdvice: org.springframework.boot.context.embedded.AnnotationConfigEmbeddedWebApp
[...]
[http://127.0.0.1:8080/cms-0.0.1-SNAPSHOT/] Started Date [Sat Oct 28 16:56:46 UTC 2017] at [19419] in [Tomcat]
2017-10-28 16:56:53.269 INFO 19419 --- [           main] o.s.w.s.h.RequestMappingHandlerMapping : Mapped "[{}]" onto public org.springframework.web.ModelAndView org.springframework.web.autoconfigure.web.BasicErrorController.errorHtml(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
2017-10-28 16:56:53.271 INFO 19419 --- [           main] o.s.w.s.h.RequestMappingHandlerMapping : Mapped "[{}]" onto public org.springframework.http.ResponseEntity<java.util.Map> java.lang.String,java.lang.Object org.springframework.boot.autoconfigure.web.BasicErrorController.error(javax.servlet.http.HttpServletRequest)
2017-10-28 16:56:53.334 INFO 19419 --- [           main] o.s.w.s.h.SimpleUrlHandlerMapping : Mapped URL path [/webjars/**] onto handler of type [class org.springframework.web.servlet.resource.ResourceHttpRequestHandler]
2017-10-28 16:56:53.339 INFO 19419 --- [           main] o.s.w.s.h.SimpleUrlHandlerMapping : Mapped URL path [/**favicon.ico] onto handler of type [class org.springframework.web.servlet.resource.ResourceHttpRequestHandler]
2017-10-28 16:56:53.444 INFO 19419 --- [           main] o.s.w.s.h.SimpleUrlHandlerMapping : Mapped URL path [/**] onto handler of type [class org.springframework.web.servlet.resource.ResourceHttpRequestHandler]
2017-10-28 16:56:53.554 WARN 19419 --- [           main] o.s.w.s.h.SimpleUrlHandlerMapping : Cannot find template location: classpath:/templates/ (please add some templates or check your Thymeleaf configuration)
2017-10-28 16:56:55.272 INFO 19419 --- [           main] o.s.j.e.AnnotationBeanExporter      : Registering beans for JMX exposure on startup
2017-10-28 16:56:55.447 INFO 19419 --- [           main] o.s.b.c.e.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
2017-10-28 16:56:55.466 INFO 19419 --- [           main] springfive.cms.CmsApplication      : Started CmsApplication in 13.255 seconds (JVM running for 14.936)

```

That is it for this part, in the next section, we will create the first **REST (Representational State Transfer)** resources and understand how the REST endpoints work.

Creating the REST resources

Now, we have an application up and running in this section, and we will add some REST endpoints and model some initial classes for the CMS application, the REST endpoints will be useful for the AngularJS integration.

One of the required characteristics for the APIs is the documentation, and a popular tool to help us with these tasks is Swagger. The Spring Framework supports Swagger, and we can do it with a couple of annotations. The project's Spring Fox is the correct tool to do this, and we will take a look at the tool in this chapter.

Let's do this.

Models

Before we start to create our class, we will add the Lombok dependency in our project. It is a fantastic library which provides some interesting things such as GET/SET at compilation time, the `val` keyword to make variables final, `@Data` to make a class with some default methods like getters/setters, `equals`, and `hashCode`.

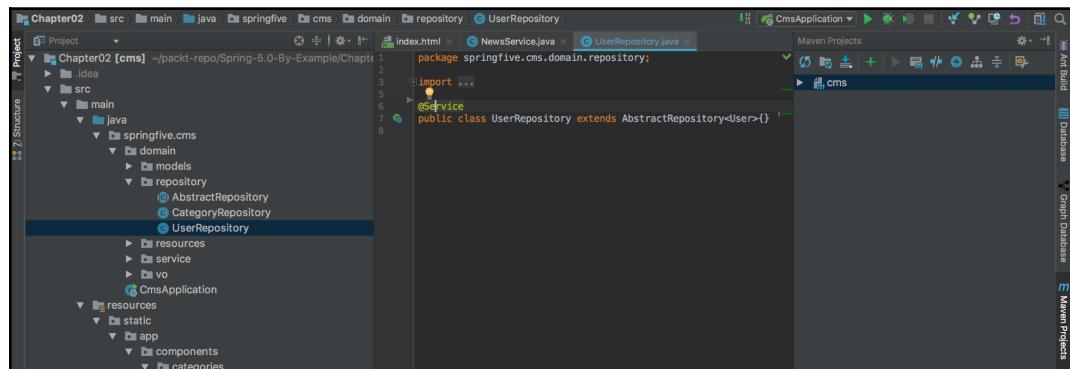
Adding Lombok dependency

Put the following dependency in a `pom.xml` file:

```
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.16.16</version>
    <scope>provided</scope>
</dependency>
```

The provided scope instructs Maven not to include this dependency in the JAR file because we need it at compile time. We do not need it at runtime. Wait for Maven to download the dependency, that is all for now.

Also, we can use the **Reimport All Maven Projects** provided by IntelliJ IDEA, located in the Maven Projects tab, as shown here:



Creating the models

Now, we will create our models, which are Java classes annotated with `@Data`.

Tag

This class represents a tag in our system. There isn't necessarily any repository for it because it will be persisted together with our `News` entity:

```
package springfive.cms.domain.models;

import lombok.Data;

@Data
public class Tag {

    String value;

}
```

Category

A category model for our CMS application can be used to group the news. Also, the other important thing is that this makes our news categorized to make the search task easy. Take a look at the following code:

```
package springfive.cms.domain.models;

import lombok.Data;

@Data
public class Category {

    String id;

    String name;

}
```

User

It represents a user in our domain model. We have two different profiles, the author who acts as a news writer, and another one is a reviewer who must review the news registered at the portal. Take a look at the following example:

```
package springfive.cms.domain.models;

import lombok.Data;

@Data
public class User {

    String id;

    String identity;

    String name;

    Role role;

}
```

News

This class represents news in our domain, for now, it does not have any behaviors. Only properties and getters/setters are exposed; in the future, we will add some behaviors:

```
package springfive.cms.domain.models;

import java.util.Set;
import lombok.Data;

@Data
public class News {

    String id;

    String title;

    String content;

    User author;

    Set<User> mandatoryReviewers;
    Set<Review> reviewers;
```

```
Set<Category> categories;  
  
Set<Tag> tags;  
  
}
```

The Review class can be found at GitHub: (<https://github.com/PacktPublishing/Spring-5.0-By-Example/tree/master/Chapter02/src/main/java/springfive/cms/domain/models>).

As we can see, they are simple Java classes which represent our CMS application domain. It is the heart of our application, and all the domain logic will reside in these classes. It is an important characteristic.

Hello REST resources

We have created the models, and we can start to think about our REST resources. We will create three main resources:

- CategoryResource which will be responsible for the Category class.
- The second one is UserResource. It will manage the interactions between the User class and the REST APIs.
- The last one, and more important as well, will be the NewsResource which will be responsible for managing news entities, such as reviews.

Creating the CategoryResource class

We will create our first REST resource, let's get started with the CategoryResource class which is responsible for managing our Category class. The implementation of this entity will be simple, and we will create CRUD endpoints such as create, retrieve, update, and delete. We have two important things we must keep in mind when we create the APIs. The first one is the correct HTTP verb such as POST, GET, PUT and DELETE. It is essential for the REST APIs to have the correct HTTP verb as it provides us with intrinsic knowledge about the API. It is a pattern for anything that interacts with our APIs. Another thing is the status codes, and it is the same as the first one we must follow, this is the pattern the developers will easily recognize. The *Richardson Maturity Model* can help us create amazing REST APIs, and this model introduces some levels to measure the REST APIs, it's a kind of thermometer.

Firstly, we will create the skeleton for our APIs. Think about what features you need in your application. In the next section, we will explain how to add a service layer in our REST APIs. For now, let's build a `CategoryResource` class, our implementation could look like this:

```
package springfive.cms.domain.resources;

import java.util.Arrays;
import java.util.List;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;
import springfive.cms.domain.models.Category;
import springfive.cms.domain.vo.CategoryRequest;

@RestController
@RequestMapping("/api/category")
public class CategoryResource {

    @GetMapping(value = "/{id}")
    public ResponseEntity<Category> findOne(@PathVariable("id") String id) {
        return ResponseEntity.ok(new Category());
    }

    @GetMapping
    public ResponseEntity<List<Category>> findAll() {
        return ResponseEntity.ok(Arrays.asList(new Category(), new Category()));
    }

    @PostMapping
    public ResponseEntity<Category> newCategory(CategoryRequest category) {
        return new ResponseEntity<>(new Category(), HttpStatus.CREATED);
    }

    @DeleteMapping("/{id}")
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public void removeCategory(@PathVariable("id") String id){}
```

```
}

    @PutMapping("/{id}")
    public ResponseEntity<Category> updateCategory(@PathVariable("id")
String id,CategoryRequest category){
        return new ResponseEntity<>(new Category(), HttpStatus.OK);
    }

}
```

The CategoryRequest can be found at GitHub (<https://github.com/PacktPublishing/Spring-5.0-By-Example/tree/master/Chapter02/src/main/java/springfive/cms/domain>).

We have some important concepts here. The first one is `@RestController`. It instructs the Spring Framework that the `CategoryResource` class will expose REST endpoints over the Web-MVC module. This annotation will configure some things in a framework, such as `HttpMessageConverters` to handle HTTP requests and responses such as XML or JSON. Of course, we need the correct libraries on the classpath, to handle JSON and XML. Also, add some headers to the request such as `Accept` and `Content-Type`. This annotation was introduced in version 4.0. It is a kind of syntactic sugar annotation because it's annotated with `@Controller` and `@ResponseBody`.

The second is the `@RequestMapping` annotation, and this important annotation is responsible for the HTTP request and response in our class. The usage is quite simple in this code when we use it on the class level, it will propagate for all methods, and the methods use it as a relative. The `@RequestMapping` annotation has different use cases. It allows us to configure the HTTP verb, params, and headers.

Finally, we have `@GetMapping`, `@PostMapping`, `@DeleteMapping`, and `@PutMapping`, these annotations are a kind of shortcut to configure the `@RequestMapping` with the correct HTTP verbs; an advantage is that these annotations make the code more readable.

Except for the `removeCategory`, all the methods return the `ResponseEntity` class which enables us to handle the correct HTTP status codes in the next section.

UserResource

The `UserResource` class is the same as `CategoryResource`, except that it uses the `User` class. We can find the whole code on the GitHub (<https://github.com/PacktPublishing/Spring-5.0-By-Example/tree/master/Chapter02>).

NewsResource

The NewsResource class is essential, this endpoint enables users to review news previously registered, and it also provides an endpoint to return the updated news. This is an important feature because we are interested only in the relevant news. Irrelevant news cannot be shown on the portal. The resource class should look like this:

```
package springfive.cms.domain.resources;

import java.util.Arrays;
import java.util.List;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;
import springfive.cms.domain.models.News;
import springfive.cms.domain.models.Review;
import springfive.cms.domain.vo.NewsRequest;

@RestController
@RequestMapping("/api/news")
public class NewsResource {

    @GetMapping(value = "/{id}")
    public ResponseEntity<News> findOne(@PathVariable("id") String id) {
        return ResponseEntity.ok(new News());
    }

    @GetMapping
    public ResponseEntity<List<News>> findAll() {
        return ResponseEntity.ok(Arrays.asList(new News(), new News()));
    }

    @PostMapping
    public ResponseEntity<News> newNews(NewsRequest news) {
        return new ResponseEntity<>(new News(), HttpStatus.CREATED);
    }

    @DeleteMapping("/{id}")
    @ResponseStatus(HttpStatus.NO_CONTENT)
```

```
public void removeNews(@PathVariable("id") String id){  
}  
  
@PutMapping("/{id}")  
public ResponseEntity<News> updateNews(@PathVariable("id") String  
id, NewsRequest news){  
    return new ResponseEntity<>(new News(), HttpStatus.OK);  
}  
  
@GetMapping(value = "/{id}/review/{userId}")  
public ResponseEntity<Review> review(@PathVariable("id") String  
id, @PathVariable("userId") String userId){  
    return ResponseEntity.ok(new Review());  
}  
  
@GetMapping(value = "/revised")  
public ResponseEntity<List<News>> revisedNews(){  
    return ResponseEntity.ok(Arrays.asList(new News(), new News()));  
}  
}
```

The `NewsRequest` class can be found at [GitHub](#).

Pay attention to the HTTP verbs and the HTTP status code, as we need to follow the correct semantics.

Adding service layer

Now, we have the skeleton for the REST layer ready, and in this section, we will start to create a service layer for our application. We will show how the Dependency Injection works under the hood, learn the stereotype annotations on Spring Framework and also start to think about our persistence storage, which will be presented in the next section.

Changes in the model

We need to make some changes to our model, specifically in the `News` class. In our business rules, we need to keep our information safe, then we need to review all the news. We will add some methods to add a new review done by a user, and also we will add a method to check if the news was reviewed by all mandatory reviewers.

Adding a new review

For this feature, we need to create a method in our News class, the method will return a Review and should look like this:

```
public Review review(String userId, String status) {
    final Review review = new Review(userId, status);
    this.reviewers.add(review);
    return review;
}
```

We do not need to check if the user, who performs the review action, is a mandatory reviewer at all.

Keeping the news safely

Also, we need to check if the news is fully revised by all mandatory reviewers. It is quite simple, we are using Java 8, and it provides the amazing Stream interface, which makes the collections interactions easier than before. Let's do this:

```
public Boolean revised() {
    return this.mandatoryReviewers.stream().allMatch(reviewer ->
        this.reviewers.stream()
            .anyMatch(review -> reviewer.id.equals(review.userId) &&
"approved".equals(review.status)));
}
```

Thanks, Java 8, we appreciate it.

Before starting the service layer

Our application needs to have a persistence storage where our records can be loaded, even if the application goes down. We will create the fake implementation for our repositories. In chapter 3, *Persistence with Spring Data and Reactive Fashion*, we will introduce the Spring Data projects which help developers create amazing repositories with a fantastic DSL. For now, we will create some Spring beans to store our elements in memory, let's do that.

CategoryService

Let's start with our simplest service, the `CategoryService` class, the behaviors expected of this class are CRUD operations. Then, we need a representation of our persistence storage or repository implementation, for now, we are using the ephemeral storage and `ArrayList` with our categories. In the next chapter, we will add the real persistence for our CMS application.

Let's create our first Spring service. The implementation is in the following snippet:

```
package springfive.cms.domain.service;

import java.util.List;
import org.springframework.stereotype.Service;
import springfive.cms.domain.models.Category;
import springfive.cms.domain.repository.CategoryRepository;

@Service
public class CategoryService {

    private final CategoryRepository categoryRepository;

    public CategoryService(CategoryRepository categoryRepository) {
        this.categoryRepository = categoryRepository;
    }

    public Category update(Category category){
        return this.categoryRepository.save(category);
    }

    public Category create(Category category){
        return this.categoryRepository.save(category);
    }

    public void delete(String id){
        final Category category = this.categoryRepository.findOne(id);
        this.categoryRepository.delete(category);
    }

    public List<Category> findAll(){
        return this.categoryRepository.findAll();
    }

    public Category findOne(String id){
        return this.categoryRepository.findOne(id);
    }
```

```
}
```

There is some new stuff here. This class will be detected and instantiated by the Spring container because it has a `@Service` annotation. As we can see, there is nothing special in that class. It does not necessarily extend any class or implement an interface. We received the `CategoryRepository` on a constructor, this class will be provided by the Spring container because we instruct the container to produce this, but in Spring 5 it is not necessary to use `@Autowired` anymore in the constructor. It works because we had the only one constructor in that class and Spring will detect it. Also, we have a couple of methods which represent the CRUD behaviors, and it is simple to understand.

UserService

The `UserService` class is quite similar to the `CategoryService`, but the rules are about the `User` entity, for this entity we do not have anything special. We have the `@Service` annotation, and we received the `UserRepository` constructor as well. It is quite simple and easy to understand. We will show the `UserService` implementation, and it must be like this:

```
package springfive.cms.domain.service;

import java.util.List;
import java.util.UUID;
import org.springframework.stereotype.Service;
import springfive.cms.domain.models.User;
import springfive.cms.domain.repository.UserRepository;
import springfive.cms.domain.vo.UserRequest;

@Service
public class UserService {

    private final UserRepository userRepository;

    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    public User update(String id, UserRequest userRequest) {
        final User user = this.userRepository.findOne(id);
        user.setIdentity(userRequest.getIdentity());
        user.setName(userRequest.getName());
        user.setRole(userRequest.getRole());
        return this.userRepository.save(user);
    }
}
```

```
}

public User create(UserRequest userRequest) {
    User user = new User();
    user.setId(UUID.randomUUID().toString());
    user.setIdentity(userRequest.getIdentity());
    user.setName(userRequest.getName());
    user.setRole(userRequest.getRole());
    return this.userRepository.save(user);
}

public void delete(String id) {
    final User user = this.userRepository.findOne(id);
    this.userRepository.delete(user);
}

public List<User> findAll() {
    return this.userRepository.findAll();
}

public User findOne(String id) {
    return this.userRepository.findOne(id);
}

}
```

Pay attention to the class declaration with `@Service` annotation. This is a very common implementation in the Spring ecosystem. Also, we can find `@Component`, `@Repository` annotations. `@Service` and `@Component` are common for the service layer, and there is no difference in behaviors. The `@Repository` changes the behaviors a little bit because the frameworks will translate some exceptions on the data access layer.

NewsService

This is an interesting service which will be responsible for managing the state of our news. It will interact like a *glue* to call the domain models, in this case, the News entity. The service is pretty similar to the others. We received the `NewsRepository` class, a dependency and kept the repository to maintain the states, let's do that.

The `@Service` annotation is present again. This is pretty much standard for Spring applications. Also, we can change to the `@Component` annotation, but it does not make any difference to our application.

Configuring Swagger for our APIs

Swagger is the de facto tool for document web APIs, and the tool allows developers to model APIs, create an interactive way to play with the APIs, and also provides an easy way to generate the client implementation in a wide range of languages.

The API documentation is an excellent way to engage developers to use our APIs.

Adding dependencies to pom.xml

Before we start the configuration, we need to add the required dependencies. These dependencies included Spring Fox in our project and offered many annotations to configure Swagger properly. Let's add these dependencies.

The new dependencies are in the `pom.xml` file:

```
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>2.7.0</version>
</dependency>

<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>2.7.0</version>
</dependency>
```

The first dependency is the core of Swagger with annotations and related kinds of stuff. Spring Fox Swagger UI dependency provides a rich interface in HTML which permits developers to interact with the APIs.

Configuring Swagger

The dependencies are added, now we can configure the infrastructure for Swagger. The configuration is pretty simple. We will create a class with `@Configuration` to produce the Swagger configuration for the Spring container. Let's do it.

Take a look at the following Swagger configuration:

```
package springfive.cms.infra.swagger;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
```

```
import org.springframework.web.bind.annotation.RestController;
import springfox.documentation.builders.ParameterBuilder;
import springfox.documentation.builders.PathSelectors;
import springfox.documentation.builders.RequestHandlerSelectors;
import springfox.documentation.spi.DocumentationType;
import springfox.documentation.spring.web.plugins.Docket;
import springfox.documentation.swagger2.annotations.EnableSwagger2;

@Configuration
@EnableSwagger2
public class SwaggerConfiguration {

    @Bean
    public Docket documentation() {
        return new Docket(DocumentationType.SWAGGER_2)
            .select()
            .apis(RequestHandlerSelectors.withClassAnnotation(RestController.class
        ))
            .paths(PathSelectors.any())
            .build();
    }

}
```

The `@Configuration` instructs the Spring to generate a bean definition for Swagger. The annotation, `@EnableSwagger2` adds support for Swagger. `@EnableSwagger2` should be accompanied by `@Configuration`, it is mandatory.

The `Docket` class is a builder to create an API definition, and it provides sensible defaults and convenience methods for configuration of the Spring Swagger MVC Framework.

The invocation of method

`.apis(RequestHandlerSelectors.withClassAnnotation(RestController.class))` instructs the framework to handle classes annotated with `@RestController`.

There are many methods to customize the API documentation, for example, there is a method to add authentication headers.

That is the Swagger configuration, in the next section, we will create a first documented API.

First documented API

We will start with the `CategoryResource` class, because it is simple to understand, and we need to keep the focus on the technology stuff. We will add a couple of annotations, and the magic will happen, let's do magic.

The `CategoryResource` class should look like this:

```
package springfive.cms.domain.resources;

import io.swagger.annotations.Api;
import io.swagger.annotations.ApiOperation;
import io.swagger.annotations.ApiResponse;
import io.swagger.annotations.ApiResponses;
import java.util.List;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;
import springfive.cms.domain.models.Category;
import springfive.cms.domain.service.CategoryService;
import springfive.cms.domain.vo.CategoryRequest;

@RestController
@RequestMapping("/api/category")
@Api(tags = "category", description = "Category API")
public class CategoryResource {

    private final CategoryService categoryService;

    public CategoryResource(CategoryService categoryService) {
        this.categoryService = categoryService;
    }

    @GetMapping(value = "/{id}")
    @ApiOperation(value = "Find category", notes = "Find the Category by ID")
    @ApiResponses(value = {
        @ApiResponse(code = 200,message = "Category found"),
        @ApiResponse(code = 404,message = "Category not found"),
    })
}
```

```
    public ResponseEntity<Category> findOne(@PathVariable("id") String
id) {
    return ResponseEntity.ok(new Category());
}

@GetMapping
@ApiOperation(value = "List categories", notes = "List all
categories")
@ApiResponses(value = {
    @ApiResponse(code = 200,message = "Categories found"),
    @ApiResponse(code = 404,message = "Category not found")
})
public ResponseEntity<List<Category>> findAll(){
    return ResponseEntity.ok(this.categoryService.findAll());
}

@PostMapping
@ApiOperation(value = "Create category", notes = "It permits to
create a new category")
@ApiResponses(value = {
    @ApiResponse(code = 201,message = "Category created
successfully"),
    @ApiResponse(code = 400,message = "Invalid request")
})
public ResponseEntity<Category> newCategory(@RequestBody
CategoryRequest category){
    return new ResponseEntity<>(this.categoryService.create(category),
HttpStatus.CREATED);
}

@DeleteMapping("/{id}")
@ResponseStatus(HttpStatus.NO_CONTENT)
@ApiOperation(value = "Remove category", notes = "It permits to
remove a category")
@ApiResponses(value = {
    @ApiResponse(code = 200,message = "Category removed
successfully"),
    @ApiResponse(code = 404,message = "Category not found")
})
public void removeCategory(@PathVariable("id") String id){

}

@PutMapping("/{id}")
@ResponseStatus(HttpStatus.NO_CONTENT)
@ApiOperation(value = "Update category", notes = "It permits to
update a category")
@ApiResponses(value = {
    @ApiResponse(code = 200,message = "Category update
```

```
        successfully"),
        @ApiResponse(code = 404,message = "Category not found"),
        @ApiResponse(code = 400,message = "Invalid request")
    })
    public ResponseEntity<Category> updateCategory(@PathVariable("id")
String id,CategoryRequest category){
    return new ResponseEntity<>(new Category(), HttpStatus.OK);
}

}
```

There are a lot of new annotations to understand. The `@Api` is the root annotation which configures this class as a Swagger resource. There are many configurations, but we will use the tags and description, as they are enough.

The `@ApiOperation` describes an operation in our API, in general against the requested path. The `value` attribute is regarded as the summary field on Swagger, it is a brief of the operation, and `notes` is a description of an operation (more detailed content).

The last one is the `@ApiResponse` which enables developers to describe the responses of an operation. Usually, they want to configure the status codes and message to describe the result of an operation.



Before you run the application, we should compile the source code. It can be done using the Maven command line using the `mvn clean install`, or via IDE using the **Run Application**.

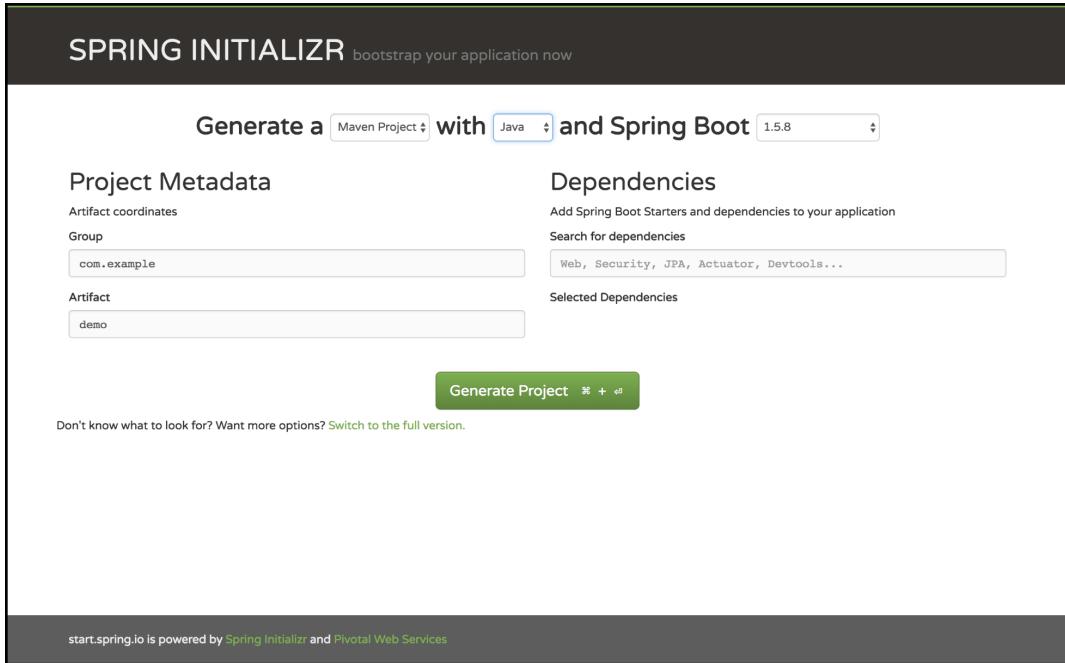
Now, we have configured the Swagger integration, we can check the API documentation on the web browser. To do it, we need to navigate to `http://localhost:8080/swagger-ui.html` and this page should be displayed:

The screenshot shows the Swagger UI homepage. At the top, there's a green header bar with the 'swagger' logo, a 'default (/v2/api-docs)' link, and an 'Explore' button. Below the header, the title 'Api Documentation' is displayed, along with links to 'Api Documentation' and 'Apache 2.0'. The main content area lists three categories: 'category : Category API', 'news-resource : News Resource', and 'user-resource : User Resource'. Each category has a 'Show/Hide' button, a 'List Operations' button, and an 'Expand Operations' button. At the bottom of the page, a note indicates '[BASE URL: /, API VERSION: 1.0]'.

We can see APIs endpoints configured in our CMS application. Now, we will take a look at **category** which we have configured previously, click on the **Show/Hide** link. The output should be:

The screenshot shows the Swagger UI interface for the CMS application's API documentation. At the top, there is a green header bar with the 'swagger' logo, a dropdown menu showing 'default (v2/api-docs)', and a 'Explore' button. Below the header, the title 'Api Documentation' is displayed, followed by 'Api Documentation' and 'Apache 2.0'. The main content area is titled 'category : Category API'. It lists five operations: GET /api/category (List categories), POST /api/category (Create category), DELETE /api/category/{id} (Remove category), GET /api/category/{id} (Find category), and PUT /api/category/{id} (Update category). Each operation has a 'Show/Hide' link, a 'List Operations' link, and an 'Expand Operations' link. Below the category section, there are sections for 'news-resource : News Resource' and 'user-resource : User Resource', each with similar operation lists. At the bottom left, it says '[BASE URL: /, API VERSION: 1.0]'.

As we can see, there are five operations in our **Category API**, the operation has a path and a summary to help understand the purpose. We can click on the requested operation and see detailed information about the operation. Let's do it, click on **List categories** to see detailed documentation. The page looks like this:



Outstanding job. Now we have an amazing API with excellent documentation. Well done.

Let's continue creating our CMS application.

Integrate with AngularJS

The AngularJS Framework has been becoming a trend for a few years, the community is super active, the project was created by Google.

The main idea of the framework is to help developers handle the complexities of the frontend layer, especially in the HTML part. The HTML markup language is static. It is a great tool to create static documents, but today it is not a requirement for modern web applications. These applications need to be dynamic. The UX teams around the world, work hard to create amazing applications, with different effects, these guys try to keep the applications more comfortable for the users.

AngularJS adds the possibility of extending the HTML with some additional attributes and tags. In this section, we will add some interesting behaviors on the frontend application. Let's do it.

AngularJS concepts

In our CMS application, we will work with some Angular components. We will use `Controllers` which will interact with our HTML and handle the behavior of some pages, such as those that show error messages. The `Services` is responsible for handling the infrastructure code such as interacting with our CMS API. This book is not intended to be an AngularJS guide. However, we will take a look at some interesting concepts to develop our application.

The AngularJS common tags are:

- `ng-app`
- `ng-controller`
- `ng-click`
- `ng-hide`
- `ng-show`

These tags are included in the AngularJS Framework. There are many more tags created and maintained by the community. There is, for example, a library to work with HTML forms, we will use it to add dynamic behaviors in our CMS Portal.

Controllers

Controllers are part of the framework to handle the business logic of the application. They should be used to control the flow of data in an application. The controller is attached to the DOM via the `ng-controller` directive.

To add some actions to our view, we need to create functions on controllers, the way to do this is by creating functions and adding them to the `$scope` object.

The controllers cannot be used to carry out DOM manipulations, format data and filter data, it is considered best practice in the AngularJS world.

Usually, the controllers inject the service objects to delegate handling the business logic. We will understand services in the next section.

Services

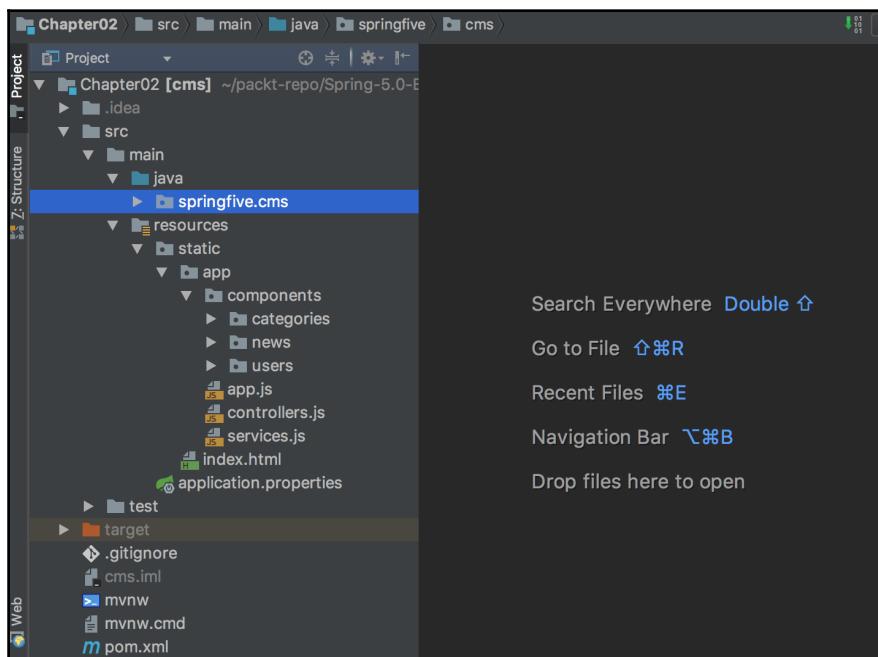
Services are the objects to handle business logic in our application. In some cases, they can be used to handle state. The services objects are a singleton which means we have only one instance in our entire application.

In our application, the services are responsible for interacting with our CMS APIs built on Spring Boot. Let's do that.

Creating the application entry point

The Spring Boot Framework allows us to serve static files. These files should be in the classpath in one of these folders, `/static`, `/public`, `/resources`, or `/META-INF/resources`.

We will use the `/static` folder, in this folder, we will put our AngularJS application. There are some standards to modularize the AngularJS application folder structure which depends on the application size and requirements. We will use the most simple style to keep the attention on Spring integration. Look at the project structure:



There are some assets to start and run an AngularJS application. We will use the Content Delivery Network (CDN) to load the AngularJS Framework, the Angular UI-Router which helps to handle routing on our web application, and the Bootstrap Framework which helps to develop our pages.



Content Delivery Network is distributed proxy servers around the world. It makes the content more high availability and improves performance because it will be hosted nearer the end user. The detailed explanation can be found at CloudFare Page (<https://www.cloudflare.com/learning/cdn/what-is-a-cdn/>).

Then we can start to configure our AngularJS application. Let's start with our entry point, `index.html`:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Spring Boot Security</title>
    <link rel="stylesheet"
        href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.mi
n.css">
</head>
<body ng-app="cms">

    <!-- Header -->
    <nav class="navbar navbar-default navbar-fixed-top">
        <div class="container">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle collapsed" data-
                toggle="collapse" data-target="#navbar"
                    aria-expanded="false" aria-controls="navbar">
                    <span class="sr-only">Toggle navigation</span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                <a class="navbar-brand" href="#">CMS</a>
            </div>
            <div id="navbar" class="collapse navbar-collapse">
                <ul class="nav navbar-nav">
                    <li class="active"><a href="#">Home</a></li>
                    <li><a href="#users">Users</a></li>
                    <li><a href="#categories">Categories</a></li>
                    <li><a href="#news">News</a></li>
```

```
</ul>
</div>
</div>
</nav>

<!-- Body -->
<div class="container">
    <div ui-view></div>
</div>

<script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.4.8/angular.min.js"></script>
<script
src="https://cdnjs.cloudflare.com/ajax/libs/angular-ui-router/1.0.3/angular-ui-router.js"></script>

<script type="text/javascript" src="app/app.js"></script>

<script type="text/javascript" src="app/controllers.js"></script>
<script type="text/javascript" src="app/services.js"></script>

<script type="text/javascript"
src="app/components/categories/category-controller.js"></script>
<script type="text/javascript"
src="app/components/categories/category-service.js"></script>

<script type="text/javascript" src="app/components/news/news-
controller.js"></script>
<script type="text/javascript" src="app/components/news/news-
service.js"></script>

<script type="text/javascript" src="app/components/users/user-
controller.js"></script>
<script type="text/javascript" src="app/components/users/user-
service.js"></script>

</body>
</html>
```

There are some important things here. Let's understand them.

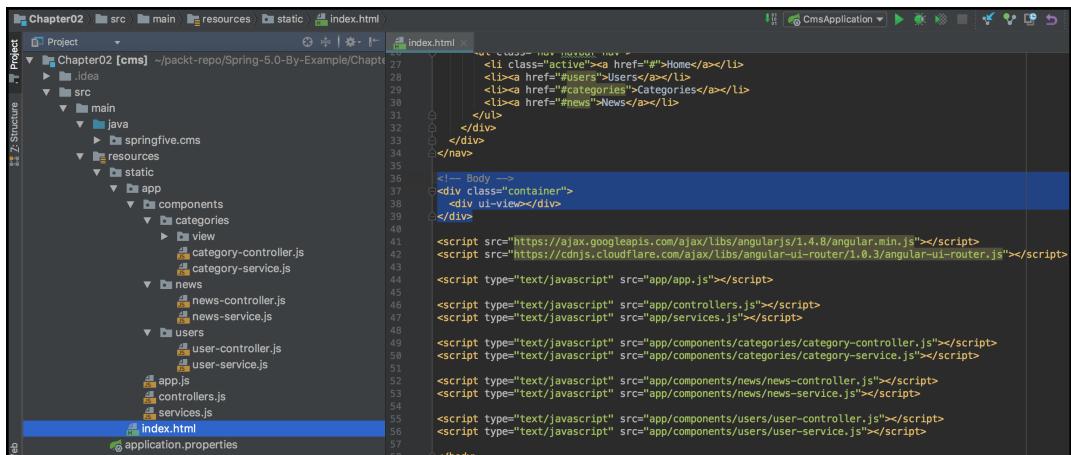
The `ng-app` tag is a directive which is used to bootstrap the AngularJS application. This tag is the root element of the application and is usually placed on the `<body>` or `<html>` tags.

The `ui-view` tag instructs the Angular UI-Router about which portion of the HTML document will be handled by the application states, in other words, the designated part has the dynamic behaviors and change depends on the routing system. Look at the following code snippet:

```
<!-- Body -->
<div class="container">
  <div ui-view></div>
</div>
```

This part of the code can be found at `index.html` file.

Following the `ui-view`, we have our JavaScript files, the first one is the AngularJS Framework, in this version the file is minified. Look at our JavaScript files, the files were created in the `/static/app/components` folder. Take a look at the image here:



The second one is the UI-Router which helps us to manage our routes. Finally, we have our JavaScript files which configure the AngularJS application, our controllers, and the services to interact with our CMS APIs.

Also, we have some Bootstrap classes to align fields and make design easier.

Creating the Category Controller

Now, we need to create our controllers. We will start with the simplest to make the example more easy to understand. The `CategoryController` has the responsibility of controlling the data of the `Category` entity. There are two controllers, one enables us to create a category, and another lists all categories stored in the database.

The `category-controller.js` should be like this:

```
(function (angular) {
    'use strict';

    // Controllers
    angular.module('cms.modules.category.controllers', []).

    controller('CategoryCreateController',
        ['$scope', 'CategoryService', '$state',
            function ($scope, CategoryService, $state) {

                $scope.resetForm = function () {
                    $scope.category = null;
                };

                $scope.create = function (category) {
                    CategoryService.create(category).then(
                        function (data) {
                            console.log("Success on create Category!!!")
                            $state.go('categories')
                        }, function (err) {
                            console.log("Error on create Category!!!")
                        });
                };
            }]);
        });

    controller('CategoryListController',
        ['$scope', 'CategoryService',
            function ($scope, CategoryService) {
                CategoryService.find().then(function (data) {
                    $scope.categories = data.data;
                }, function (err) {
                    console.log(err);
                });
            }]);
}) (angular);
```

We have created an AngularJS module. It helps us to keep the functions organized. It acts as a kind of namespace for us. The `.controller` function is a constructor to create our controller's instances. We received some parameters, the AngularJS framework will inject these objects for us.

Creating the Category Service

The `CategoryService` object is a singleton object because it is an AngularJS service. The service will interact with our CMS APIs powered by the Spring Boot application.

We will use the `$http` service. It makes the HTTP communications easier.

Let's write the `CategoryService`:

```
(function (angular) {
  'use strict';

  /* Services */
  angular.module('cms.modules.category.services', []).
    service('CategoryService', ['$http',
      function ($http) {

        var serviceAddress = 'http://localhost:8080';
        var urlCollections = serviceAddress + '/api/category';
        var urlBase = serviceAddress + '/api/category/';

        this.find = function () {
          return $http.get(urlCollections);
        };

        this.findOne = function (id) {
          return $http.get(urlBase + id);
        };

        this.create = function (data) {
          return $http.post(urlBase, data);
        };

        this.update = function (data) {
          return $http.put(urlBase + '/id/' + data._id, data);
        };

        this.remove = function (data) {
          return $http.delete(urlBase + '/id/' + data._id, data);
        };
    }]);
})()
```

```
]);  
}) (angular);
```

Well done, now we have implemented the CategoryService.

The `.service` function is a constructor to create a service instance, the angular acts under the hood. There is an injection on a constructor, for the service we need an `$http` service to make HTTP calls against our APIs. There are a couple of HTTP methods here. Pay attention to the correct method to keep the HTTP semantics.

Summary

In this chapter, we created our first Spring application. We saw Spring Initializr, the amazing tool that helps developers create the application skeleton.

We looked at how Spring works under the hood and how the framework got configured with a couple of annotations. Now, we have a basic knowledge of the Spring Bootstrap functions, and we can understand the Dependency Injection and component scan features present in the framework.

This knowledge is the basis for the next chapters, and now we are ready to start to work with more advanced features, such as persistence. Here we go. See you in the next chapter.

3

Persistence with Spring Data and Reactive Fashion

In the previous chapter, we created our **Content Management System (CMS)** application. We also introduced **REST (Representational State Transfer)** support in Spring, which enabled us to develop a simple web application. Also, we learned how dependency injection works in the Spring Framework, which is probably the most famous feature of the framework.

In this chapter, we will add more features to our application. Systems in the real world need to persist their data on a real database; this is an essential characteristic for a production-ready application. Also, based on our model, we need to choose the correct data structure to achieve performance and avoid the impedance mismatch.

In the first part of this chapter, we will use the traditional SQL database as a store for our application. We will deep dive on the Spring Data **JPA (Java Persistence API)** to achieve the persistence for our CMS application. We will understand how to enable transactions with this amazing Spring module.

After that, we will change to a more modern type of database called **NoSQL technologies**. In this field, we will use the famous database document model called **MongoDB** and then we will create the final solution for our CMS application.

MongoDB offers a fantastic solution for our application because it has support for a document storage model and enables us to store our objects in the form of JSON, which makes our data more readable. Also, MongoDB is schema-less, which is a fantastic feature because one collection can store different documents. It means records can have different fields, content, and sizes. The other important characteristic from MongoDB is the query model. It offers a document-based query that is easy to understand, and, based on JSON notations, our queries will be more readable than any other database can be.

Finally, we will add the most important feature present in Spring 5.0: support for Reactive Streams. Our application will be transformed into a modern web application which has some important requirements.

Here's an overview of what you will learn in this chapter:

- Implementing the Spring Data JPA
- Creating repositories with Spring Data Reactive MongoDB
- Learning the Reactive Spring
- Understand the Project Reactor

Learning the basics of Docker

We learned about Docker concepts in [Chapter 1, Journey to the Spring World](#). Now, it is time to test our knowledge and put it into practice. In the first part of this chapter, we will start MongoDB and Postgres instances to serve as a database for our application. We will configure connection settings in the application.

In the last part of this chapter, we will introduce the Maven plugin which provides an easy way to create Docker images via `pom.xml` with a couple of configurations on file. Finally, we will run our application in a Docker container.

Preparing MongoDB

Let's create our MongoDB container. We will use the official image provided by the Docker Hub.

First, we need to pull the image:

```
docker pull mongo:3.4.10
```

Then, we will see the Docker Engine downloading the image contents.

To create an isolation from our containers, we will create a separated network for our application and database. The network should use the bridge driver to allow the container communications.

Let's create a docker network:

```
docker network create cms-application
```

The command output should be an ID of a created network. Your ID will probably be different compared to mine:

```
2 updates are security updates.

Last login: Wed Nov 1 00:13:53 2017 from 10.0.2.2
ubuntu@ubuntu-xenial:~$ docker network create cms-application
5a8485d8da42a4680347635e57041b35d2d09642ac9f5e1194c7334a5e4bfe92
ubuntu@ubuntu-xenial:~$
```

To check if the network was created successfully, the `docker network ls` command can help us.

We will start our MongoDB. The network should be `cms-application`, but we will map the database port to a host port. For debugging purposes, we will connect a client to a running database, but please don't do this in a non-development environment.



Exposing a port over host is not a best practice. Hence, we use a Docker container, which is one of the main advantages is process isolation. In this case, we will have no control over the network. Otherwise, we may cause some port conflicts.

To start, type the following command:

```
docker run -d --name mongodb --net cms-application -p 27017:27017
mongo:3.4.10
```



Also, we can stop the Docker MongoDB container using `docker stop mongodb` and start our container again by using the following command: `docker start mongodb`.

The output will be a hash which represents the ID of the container.

The parameter instructions are:

- `-d`: This instructs Docker to run the container in a background mode
- `--name`: The container name; it will be a kind of hostname in our network

- `--net`: The network where the container will be attached
- `-p`: The host port and container port, which will be mapped to a container on a host interface

Now, we have a pretty standard MongoDB instance running on our machines, and we can start to add a persistence in our CMS application. We will do that soon.

Preparing a PostgreSQL database

Like MongoDB, we will prepare a PostgreSQL instance for our CMS application. We will change our persistence layer to demonstrate how Spring Data abstracts it for developers. Then, we need to prepare a Docker Postgres instance for that.

We will use the version 9.6.6 of Postgres and use the `alpine` tag because it is smaller than other Postgres images. Let's pull our image. The command should be like this:

```
docker pull postgres:9.6.6-alpine
```

Then, wait until the download ends.

In the previous section, we created our Docker network called `cms-application`. Now, we will start our Postgres instance on that network as we did for MongoDB. The command to start the Postgres should be the following:

```
docker run -d --name postgres --net cms-application -p 5432:5432 -e  
POSTGRES_PASSWORD=cms@springfive  
postgres:9.6.6-alpine
```

The list of parameters is the same as we passed for MongoDB. We want to run it in background mode and attach it to our custom network. As we can see, there is one more new parameter in the `docker run` command. Let's understand it:

- `-e`: This enables us to pass environment variables for a container. In this case, we want to change the password value.

Good job. We have done our infrastructure requirements. Let's understand the persistence details right now.

Spring Data project

The Spring Data project is an umbrella project that offers a familiar way to create our data access layer on a wide range of database technologies. It means there are high-level abstractions to interact with different kinds of data structures, such as the document model, column family, key-value, and graphs. Also, the JPA specification is fully supported by the Spring Data JPA project.

These modules offer powerful object-mapping abstractions for our domain model.

There is support for different types of data structures and databases. There is a set of sub-modules to keep the framework modularity. Also, there are two categories of these sub-modules: the first one is a subset of projects supported by the Spring Framework Team and the second one is a subset of sub-modules provided by the community.

Projects supported by the Spring Team include:

- Spring Data Commons
- Spring Data JPA
- Spring Data MongoDB
- Spring Data Redis
- Spring Data for Apache Cassandra

Projects supported by the community include:

- Spring Data Aerospike
- Spring Data ElasticSearch
- Spring Data DynamoDB
- Spring Data Neo4J

The base of the repositories interfaces chain is the `Repository` interface. It is a marker interface, and the general purpose is to store the type information. The type will be used for other interfaces that extend it.

There is also a `CrudRepository` interface. It is the most important, and the name is self-explanatory; it provides a couple of methods to perform CRUD operations, and it provides some utility methods, such as `count()`, `exists()`, and `deleteAll()`. Those are the most important base interfaces for the repository implementations.

Spring Data JPA

The Spring Data JPA provides an easy way to implement a data access layer using the JPA specification from Java EE. Usually, these implementations had a lot of boilerplate and repetitive code and it was hard to maintain the changes in the database code. The Spring Data JPA is trying to resolve these issues and provides a comprehensible way to do that without boilerplate and repetitive code.

The JPA specification provides an abstraction layer to interact with different database vendors that have been implemented. Spring adds one more layer to the abstraction in a high-level mode. It means the Spring Data JPA will create a repositories implementation and encapsulate the whole JPA implementation details. We can build our persistence layer with a little knowledge of the JPA spec.



The *JPA Specification* was created by the **JCP (Java Community Process)** to help developers to persist, access, and manage data between Java classes and relational databases. There are some vendors that implement this specification. The most famous implementation is **Hibernate** (<http://hibernate.org/orm/>), and by default, Spring Data JPA uses Hibernate as the JPA implementation.

Say goodbye to the **DAO (Data Access Object)** pattern and implementations. The Spring Data JPA aims to solve this problem with a well-tested framework and with some production-ready features.

Now, we have an idea of what the Spring Data JPA is. Let's put it into practice.

Configuring pom.xml for Spring Data JPA

Now, we need to put the correct dependencies to work with Spring Data JPA. There are a couple of dependencies to configure in our `pom.xml` file.

The first one is the Spring Data JPA Starter, which provides a lot of auto-configuration classes which permits us to bootstrap the application quickly. The last one is the PostgreSQL JDBC driver, and it is necessary because it contains the JDBC implementation classes to connect with the PostgreSQL database.

The new dependencies are:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

```
<dependency>
  <groupId>org.postgresql</groupId>
  <artifactId>postgresql</artifactId>
  <version>42.1.4</version>
</dependency>
```

Simple and pretty easy.

Configuring the Postgres connections

To connect our application with our recently created database, we need to configure a couple of lines in the `application.yaml` file. Once again, thanks to Spring Data Starter, our connection will be configured automatically.

We can produce the connection objects using the `@Bean` annotations as well, but there are many objects to configure. We will go forward with the configuration file. It is more simple and straightforward to understand as well.

To configure the database connections, we need to provide the Spring Framework a couple of attributes, such as the database URL, database username, password, and also a driver class name to instruct the JPA framework about the full path of the JDBC class.

The `application.yaml` file should be like this:

```
spring:
  datasource:
    url: jdbc:postgresql://localhost:5432/postgres
    username: postgres
    password: cms@springfive
    driver-class-name: org.postgresql.Driver
  jpa:
    show-sql: true
    generate-ddl: true
```

In the `datasource` section, we have configured the database credentials connections and database host as well.

The JPA section in `application.yaml` can be used to configure the JPA framework. In this part, we configured to log SQL instructions in the console. This is helpful to debug and perform troubleshooting. Also, we have configured the JPA framework to create our tables in a database when the application gets the startup process.

Awesome, the JPA infrastructure is configured. Well done! Now, we can map our models in the JPA style. Let's do that in the following section.

Mapping the models

We have configured the database connections successfully. Now, we are ready to map our models using the JPA annotations. Let's start with our `Category` model. It is a pretty simple class, which is good because we are interested in Spring Data JPA stuff.

Our first version of the `Category` model should be like this:

```
package springfive.cms.domain.models;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;
import lombok.Data;
import org.hibernate.annotations.GenericGenerator;

@Data
@Entity
@Table(name = "category")
public class Category {

    @Id
    @GeneratedValue(generator = "system-uuid")
    @GenericGenerator(name = "system-uuid", strategy = "uuid2")
    String id;

    String name;

}
```



We need to change some model classes to adapt to the JPA specification. We can find the model classes on GitHub at: <https://github.com/PacktPublishing/Spring-5.0-By-Example/tree/master/Chapter03/cms-postgres/src/main/java/springfive/cms/domain/models>.

There is some new stuff here. The `@Entity` annotation instructs the JPA framework that the annotated class is an entity, in our case, the `Category` class, and then the framework will correlate it with a database table. The `@Table` annotation is used to name the table in the database. These annotations are inserted on the class level, which means on top of the class declaration.

The `@Id` annotation instructs the JPA as to which annotated field is the primary key of the database table. It is not a good practice to generate IDs sequentially for entities, especially if you are creating the APIs. It helps hackers to understand the logic about the IDs and makes the attacks easier. So, we will generate UUIDs (Universally Unique IDentifiers) instead of simple sequentially IDs. The `@GenericGenerator` annotation instructs Hibernate, which is a JPA specification implementation vendor, to generate random UUIDs.

Adding the JPA repositories in the CMS application

Once the whole infrastructure and JPA mappings are done, we can add our repositories to our projects. In the Spring Data project, there are some abstractions, such as `Repository`, `CrudRepository`, and `JpaRepository`. We will use the `JpaRepository` because it supports the paging and sorting features.

Our repository will be pretty simple. There are a couple of standard methods, such as `save()`, `update()`, and `delete()`, and we will take a look at some DSL query methods which allow developers to create custom queries based on attribute names. We created an `AbstractRepository` to help us to store the objects in memory. It is not necessary anymore. We can remove it.

Let's create our first JPA repository:

```
package springfive.cms.domain.repository;

import java.util.List;
import org.springframework.data.jpa.repository.JpaRepository;
import springfive.cms.domain.models.Category;

public interface CategoryRepository extends JpaRepository<Category,
String> {

    List<Category> findByName(String name);

    List<Category> findByNameIgnoreCaseStartingWith(String name);

}
```

As we can see, the `JpaRepository` interface is typed with the desired entity and the type of ID of the entity as well. There is no secret to this part. This amazing thing happens to support the custom queries based on attribute names. In the `Category` model, there is an attribute called `name`. We can create custom methods in our `CategoryRepository` using the `Category` model attributes using the `By` instruction. As we can see, above `findByName(String name)`, Spring Data Framework will create the correct query to look up categories by name. It is fantastic.

There are many keywords supported by the custom query methods:

Logical Keyword	Logical Expressions
AND	And
OR	Or
AFTER	After, IsAfter
BEFORE	Before, IsBefore
CONTAINING	Containing, IsContaining, Contains
BETWEEN	Between, IsBetween
ENDING_WITH	EndingWith, IsEndingWith, EndsWith
EXISTS	Exists
FALSE	False, IsFalse
GREATER_THAN	GreaterThan, IsGreaterThan
GREATER_THAN_EQUALS	GreaterThanOrEqualTo, IsGreaterThanOrEqualTo
IN	In, IsIn
IS	Is, Equals, (or no keyword)
IS_EMPTY	IsEmpty, Empty
IS_NOT_EMPTY	IsNotEmpty, NotEmpty
IS_NOT_NULL	NotNull, IsNotNull
IS_NULL	Null, IsNull
LESS_THAN	LessThan, IsLessThan
LESS_THAN_EQUAL	LessThanOrEqualTo, IsLessThanOrEqualTo
LIKE	Like, IsLike
NEAR	Near, IsNear
NOT	Not, IsNot
NOT_IN	NotIn, IsNotIn
NOT_LIKE	NotLike, IsNotLike
REGEX	Regex, MatchesRegex, Matches
STARTING_WITH	StartingWith, IsStartingWith, StartsWith

TRUE	True, IsTrue
WITHIN	Within, IsWithin

There are many ways to create a query based on attributes names. We can combine the keywords using keywords as well, such as `findByNameAndId`, for instance. The Spring Data JPA provides a consistent way to create queries.

Configuring transactions

When we use the JPA specification, most of the applications need to have support for transactions as well. Spring has excellent support for transactions even in other modules. This support is integrated with Spring Data JPA, and we can take advantage of it. Configuring transactions in Spring is a piece of cake; we need to insert the `@Transactional` annotation whenever needed. There are some different use cases to use it. We will use the `@Transactional` in our services layer and then we will put the annotation in our service classes. Let's see our `CategoryService` class:

```
package springfive.cms.domain.service;

import java.util.List;
import java.util.Optional;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import springfive.cms.domain.exceptions.CategoryNotFoundException;
import springfive.cms.domain.models.Category;
import springfive.cms.domain.repository.CategoryRepository;
import springfive.cms.domain.vo.CategoryRequest;

@Service
@Transactional(readOnly = true)
public class CategoryService {

    private final CategoryRepository categoryRepository;

    public CategoryService(CategoryRepository categoryRepository) {
        this.categoryRepository = categoryRepository;
    }

    @Transactional
    public Category update(Category category) {
        return this.categoryRepository.save(category);
    }

    @Transactional
    public Category create(CategoryRequest request) {
```

```
Category category = new Category();
category.setName(request.getName());
return this.categoryRepository.save(category);
}

@Transactional
public void delete(String id) {
    final Optional<Category> category =
this.categoryRepository.findById(id);
    category.ifPresent(this.categoryRepository::delete);
}

public List<Category> findAll() {
    return this.categoryRepository.findAll();
}

public List<Category> findByName(String name) {
    return this.categoryRepository.findByName(name);
}

public List<Category> findByNameStartingWith(String name) {
    return
this.categoryRepository.findByNameIgnoreCaseStartingWith(name);
}

public Category findOne(String id) {
    final Optional<Category> category =
this.categoryRepository.findById(id);
    if (category.isPresent()) {
        return category.get();
    } else {
        throw new CategoryNotFoundException(id);
    }
}
}
```

There are many `@Transactional` annotations in the `CategoryService` class. The first annotation at class level instructs the framework to configure the `readOnly` for all methods present in those classes, except the methods configured with `@Transactional`. In this case, the class-level annotation will be overridden with `readOnly=false`. This is the default configuration when the value is omitted.

Installing and configuring pgAdmin3

To connect on our PostgreSQL instance, we will use pgAdmin 3, which is the free tool provided by the Postgres team.

To install pgAdmin 3, we can use the following command:

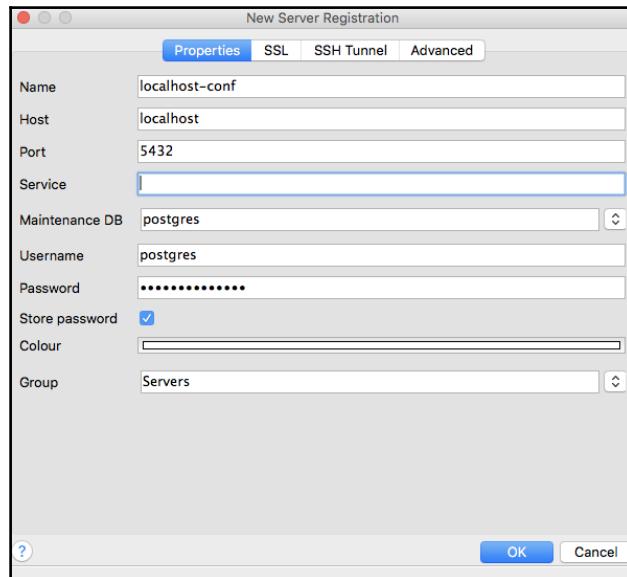
```
sudo apt-get install pgadmin3 -y
```

This will install pgAdmin 3 on our machine.

After installation, open pgAdmin 3 and then click on **Add a connection to a server**. The button looks like this:



Then, fill in the information, as shown in the following screenshot:



The password should be: cms@springfive.

Awesome, our pgAdmin 3 tool is configured.

Checking the data on the database structure

The whole application structure is ready. Now, we can check the database to get our persisted data. There are many open source Postgres clients. We will use pgAdmin 3, as previously configured.

The first time you open the application, you will be asked about the credentials and host. We must put the same information as we configured on the `application.yaml` file. Then, we are able to make instructions in the database.

Before checking the database, we can use Swagger to create some categories in our CMS system. We can use the instructions provided in [Chapter 2, Starting in the Spring World – The CMS Application](#), to create some data.

After that, we can execute the following SQL instruction in the database:

```
select * from category;
```

And the result should be the categories created on Swagger calls. In my case, I have created two categories, `sports`, and `movies`. The result will be like the ones shown in the following screenshot:

The screenshot shows the pgAdmin 3 interface. The top bar has tabs for 'SQL Editor' and 'Graphical Query Builder', and a connection dropdown set to 'postgres on postgres@localhost:5432'. Below the tabs is a toolbar with various icons. The main area has a 'Previous queries' pane containing the query 'select * from category;'. To the right is a 'Scratch pad' pane. At the bottom is an 'Output pane' with tabs for 'Data Output', 'Explain', 'Messages', and 'History'. The 'Data Output' tab is selected, displaying a table with two rows. The table has columns 'id' and 'name'. The data is as follows:

	id	name
1	efb86ff5-e33e-4b4d-bbf9-7cc719283493	sports
2	85b2a7c5-9e07-4770-8e74-f31c8328a453	movies

Awesome work, guys. The application is fully operational.

Now, we will create our final solution for the repositories. We have learned the basics of the Spring Data project and in the next section, we will change the persistence layer to a modern database.

Creating the final data access layer

We have played with the Spring Data JPA project, and we have seen how easy it can be. We learned how to configure the database connections to persist the real data on the Postgres database. Now, we will create the final solution for the data access layer for our application. The final solution will use MongoDB as a database and will use the Spring Data MongoDB project, which provides support for MongoDB repositories.

We will see some similarities with the Spring Data JPA projects. It is amazing because we can prove the power of Spring Data abstractions in practice. With a couple of changes, we can move to another database model.

Let's understand the new project and put it into practice in the following sections.

Spring Data MongoDB

The Spring Data MongoDB provides integration with our domain objects and the MongoDB document. With a couple of annotations, our entity class is ready to be persisted in the database. The mapping is based on a **POJO (Plain Old Java Object)** pattern, which is known by all Java developers.

There are two levels of abstraction supplied by the module. The first one is a high-level abstraction. It increases the developer productivity. This level provides a couple of annotations to instruct the framework to convert the domain objects in MongoDB documents and vice versa. The developer does not need to write any code about the persistence; it will be managed by the Spring Data MongoDB framework. There are more exciting things at this level, such as the rich mapping configurations provided by the Spring Conversion Service. The Spring Data projects provide a rich DSL to enable developers to create queries based on the attribute names.

The second level of abstraction is the low-level abstraction. At this level, behaviors are not automatically managed by the framework. The developers need to understand a little bit more about the Spring and MongoDB document model. The framework provides a couple of interfaces to enable developers to take control of the read and write instructions. This can be useful for scenarios where the high-level abstraction does not fit well. In this case, the control should be more granular in the entities mapping.

Again, Spring provides the power of choice for developers. The high-level abstraction improves the developer performance and the low-level permits developers to take more control.

Now, we will add mapping annotation to our model. Let's do it.

Removing the PostgreSQL and Spring Data JPA dependencies

We will convert our project to use the brand new Spring Data Reactive MongoDB repositories. After that, we will not use the Spring Data JPA and PostgreSQL drivers anymore. Let's remove these dependencies from our `pom.xml`:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>

<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <version>42.1.4</version>
</dependency>
```

And then, we can add the following dependency:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-mongodb-reactive</artifactId>
</dependency>
```



The final version of `pom.xml` can be found on GitHub at <https://github.com/PacktPublishing/Spring-5.0-By-Example/blob/master/Chapter03/cms-mongo-non-reactive/pom.xml>.

Mapping the domain model

We will add mapping annotations on our domain model. The Spring Data MongoDB will use these annotations to persist our objects in the MongoDB collections. We will start with the `Category` entity, which should be like this:

```
package springfive.cms.domain.models;

import lombok.Data;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

@Data
@Document(collection = "category")
public class Category {

    @Id
    String id;

    String name;

}
```

We added two new annotations in the `Category` class. The `@Document` from Spring Data MongoDB enables us to configure the collection name. Collections in MongoDB are similar to tables in SQL databases.

The `@Id` annotation is from the Spring Data Commons project. It is interesting because, as we can see, it is not specific for MongoDB mappings. The field annotation with this will be converted in the `_id` field on MongoDB collection.

With these few annotations, the `Category` class is configured to be persisted on MongoDB. In the following section, we will create our repository classes.

We need to do the same task for our other entities. The `User` and `News` need to be configured in the same way as we did for the `Category` class. The full source code can be found on GitHub at: <https://github.com/PacktPublishing/Spring-5.0-By-Example/tree/master/Chapter03/cms-mongo-non-reactive/src/main/java/springfive/cms/domain/models>.

Configuring the database connection

Before we create our repositories, we will configure the MongoDB connection. The repository layer abstracts the driver implementation, but is necessary to configure the driver correctly.

On the resources directory, we will change the `application.yaml` file, previously configured for the Spring Data JPA. The Spring Framework supports the configuration through the YAML file. This kind of file is more readable for humans and has a kind of hierarchy. These features are the reason to choose this extension.

The `application.yaml` file should be like the following example:

```
spring:  
  data:  
    mongodb:  
      database: cms  
      host: localhost  
      port: 27017
```



The `application.yaml` file for MongoDB can be found on GitHub (<https://github.com/PacktPublishing/Spring-5.0-By-Example/blob/master/Chapter03/cms-mongo-non-reactive/src/main/resources/application.yaml>).

The file is quite simple for now. There is a `database` tag for configuring the database name. The `host` and `port` tags are about the address that the MongoDB instance is running.

We also can configure the connections programmatically with a couple of objects, but it requires us to code a lot of boilerplate code. Spring Boot offers it out of the box for us. Let's enjoy it.

Excellent, the connection was configured successfully. The infrastructure requirements are solved. Let's go on to implement our repositories.



Spring Boot Framework supports profiles in `application.properties` or `application.yaml`. This means that if the application was configured in a properties file style, we could use `application-<profile>.properties`. Then, these properties will be applied to the required profile. In YAML style, we can use only one file with multiples profiles.

Adding the repository layer

Once the entities have been mapped, and the connections are done, it's time to create our repositories. The Spring Data Framework provides some interfaces that can be used in different use cases. We will use the specialization for the MongoDB database, which is `MongoRepository`. It extends the `PagingAndSortingRepository` and `QueryByExampleExecutor`. The first is about pagination and sorting features, and the other is about queries by example.



In some cases, the database query result set can be very large. This can cause some application performance issues because we will fetch a lot of database records. We can limit the number of records fetched from the database and configure limits for that. This technique is called **Pagination**. We can find the full documentation at *Spring Data Commons Documentation* (<https://docs.spring.io/spring-data/commons/docs/current/reference/html/>).

This interface offers a lot of built-in methods for convenience. There are a couple of methods to insert one or more instances, methods for listing all instances of requested entities, methods to remove one or more instances, and many more features, such as ordering and paging.

It enables developers to create repositories without code or even without a deep knowledge of MongoDB. However, some knowledge of MongoDB is necessary to troubleshoot various errors.

We will start by creating the `CategoryRepository`. Change the type of `CategoryRepository` to an interface instead of a class. The code in this interface is not necessary. The Spring container will inject the correct implementation when the application starts.

Let's create our first concrete repository, which means the repository will persist the data on the MongoDB we previously configured. The `CategoryRepository` needs to be like this:

```
package springfive.cms.domain.repository;

import org.springframework.data.mongodb.repository.MongoRepository;
import springfive.cms.domain.models.Category;

public interface CategoryRepository extends
MongoRepository<Category, String> {}
```

The type is an interface. Repositories do not have any stereotypes anymore. The Spring container can identify the implementation because it extends the MongoRepository interface.

The MongoRepository interface should be parameterized. The first argument is the type of model that it represents. In our case, it represents a repository for the Category class. The second parameter is about the type of ID of the model. We will use the string type for that.

Now, we need to do the same for the other entities, User, and News. The code is quite similar to the preceding code. You can find the full source code on GitHub at: <https://github.com/PacktPublishing/Spring-5.0-By-Example/tree/master/Chapter03/cms-mongo-non-reactive/src/main/java/springfive/cms/domain/repository>.

In the next section, we will check the database to assert that the rows are persisted correctly.

Checking the persistence

Now, we can test the persistence and all layers of the application. We will provide the API documentation for that. Let's open the Swagger documentation and create some records in our CMS application.

Creating sample categories on Swagger:

Parameters				
Parameter	Value	Description	Parameter Type	Data Type
category	{ "name": "sports" }	category	body	Model Example Value { "name": "string" }

Parameter content type: application/json

Response Messages

Fill in the category JSON, as shown in the preceding screenshot, and then click on **Try it out!**. It will invoke the Category API and persist the category on the database. Now, we can check it.

To connect to the MongoDB instance and check the collection, we will use the `mongo-express` tool. It is a web-based tool written in NodeJS to interact with our database instance.

The tool can be installed, but we will run the tool on a Docker container. The Docker tool will help us in this part. Let's start the container:

```
docker run -d --link mongodb:mongo --net cms-application -p 8081:8081  
mongo-express
```

It instructs Docker to spin up a container with the `mongo-express` tool and connect to the desired instance. The `--link` argument instructs Docker to create a kind of *hostname* for our MongoDB instance. Remember the name of our instance is `mongodb`; we did it on the run command previously.

Good job. Go to `http://localhost:8081` and we will see this page:

The screenshot shows the Mongo Express web application. At the top, there is a navigation bar with a green icon and the text "Mongo Express Database". Below the header, the title "Mongo Express" is centered. The main content area is divided into two sections: "Databases" and "Server Status".

Databases section:

	Database Name	Create Database
<button>View</button>	admin	<button>Del</button>
<button>View</button>	cms	<button>Del</button>
<button>View</button>	local	<button>Del</button>

Server Status section:

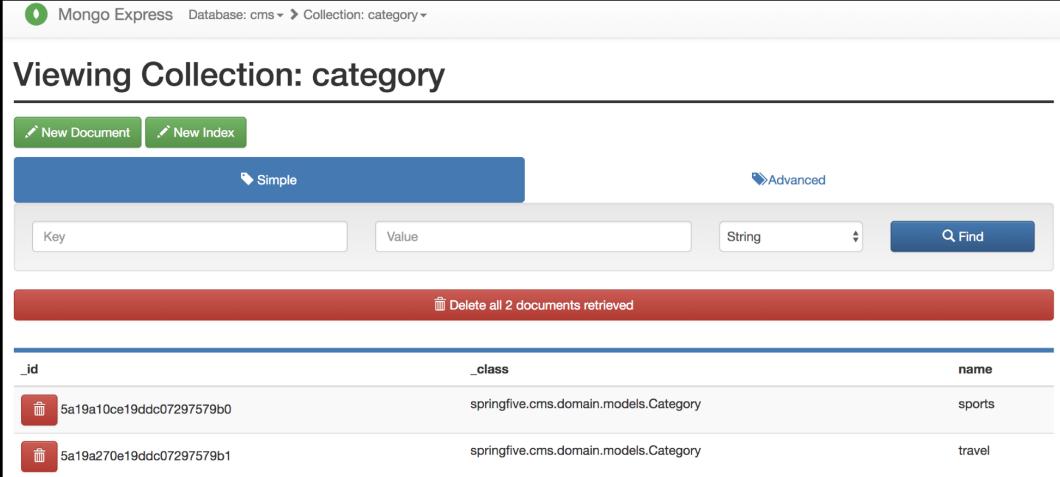
Hostname	dd71620d4f76	MongoDB Version	3.4.10
Uptime	5820 seconds	Server Time	Sat, 25 Nov 2017 17:16:39 GMT
Current Connections	6	Available Connections	838854
Active Clients	12	Queued Operations	0
Clients Reading	0	Clients Writing	0
Read Lock Queue	0	Write Lock Queue	0

There are a couple of databases. We are interested in the CMS database. Click on the **View** button next to **cms**. Then, the tool will present the collections of the selected database; in our case, the CMS database. The view should be like this:

The screenshot shows the Mongo Express interface for the CMS database. At the top, there's a header with a green icon, the text "Mongo Express", and "Database: cms". Below the header, the title "Viewing Database: cms" is displayed. Underneath the title, there's a toolbar with four buttons: "View" (green), "Export" (orange), "[JSON]" (orange), and "category" (blue). To the right of the toolbar is a search bar labeled "Collection Name" and a blue button labeled "+ Create collection". Below the toolbar, there's a section titled "Database Stats" containing a table of database statistics. The table has two columns: "Metric" and "Value". The metrics listed are: Collections (incl. system.namespaces) with value 1; Data Size with value 178 Bytes; Storage Size with value 36.9 KB; Avg Obj Size # with value 89.0 Bytes; Objects # with value 2; Extents # with value 0; Indexes # with value 1; and Index Size with value 36.9 KB.

Metric	Value
Collections (incl. system.namespaces)	1
Data Size	178 Bytes
Storage Size	36.9 KB
Avg Obj Size #	89.0 Bytes
Objects #	2
Extents #	0
Indexes #	1
Index Size	36.9 KB

The category is presented as a collection. We can **View**, **Export**, and export as JSON, but for now, we are interested in checking if our CMS application persisted the data properly. So, click on the **View** button. We will use the MongoDB collection data like this:



The screenshot shows the Mongo Express interface for viewing the 'category' collection. At the top, it says 'Mongo Express Database: cms Collection: category'. Below that is a title 'Viewing Collection: category'. There are two buttons: 'New Document' and 'New Index'. Underneath is a search bar with 'Simple' selected. The search bar has fields for 'Key', 'Value', and 'String' with a dropdown arrow, and a 'Find' button. A red banner below the search bar says 'Delete all 2 documents retrieved'. The main area shows a table with two rows of data:

_id	_class	name
5a19a10ce19ddc07297579b0	springfive.cms.domain.models.Category	sports
5a19a270e19ddc07297579b1	springfive.cms.domain.models.Category	travel

As we can see, the data was stored in MongoDB as expected. There are two categories in the database—**sports** and **travel**. There is a `_class` field that helps Spring Data to convert domain classes.

Awesome job, the CMS application is up and running, and also persisting the data in MongoDB. Now, our application is almost production ready, and the data is persisted outside in the amazing document datastore.

In the following section, we will create our Docker image, and then we will run the CMS application with Docker commands. It will be interesting.

Creating the Docker image for CMS

We are doing an awesome job. We created an application with the Spring Boot Framework. The application has been using the Spring REST, Spring Data, and Spring DI.

Now we will go a step forward and create our Docker image. It will be useful to help us to deliver our application for production. There are some advantages, and we can run the application on-premise or on any cloud providers because Docker abstracts the operating system layer. We do not need Java to be installed on the application host, and it also allows us to use different Java versions on the hosts. There are so many advantages involved in adopting Docker for delivery.

We are using Maven as a build tool. Maven has an excellent plugin to help us to create Docker images. In the following section, we will learn how Maven can help us.

Configuring the docker-maven-plugin

There is an excellent Maven plugin provided by fabric8 (<https://github.com/fabric8io/docker-maven-plugin>). It is licensed under the Apache-2.0 license, which means we can use it without any worries.

We will configure our project to use it, and after image creation, we will push this image on Docker Hub. It is a public Docker registry.

The steps are:

1. Configure the plugin
2. Push the Docker image
3. Configure the Docker Spring profile

Then, it is show time. Let's go.

Adding the plugin on pom.xml

Let's configure the Maven plugin. It is necessary to add a plugin to the plugin section on our `pom.xml` and add some configurations. The plugin should be configured as follows:

```
<plugin>
    <groupId>io.fabric8</groupId>
    <artifactId>docker-maven-plugin</artifactId>
    <version>0.21.0</version>
    <configuration>
        <images>
            <image>
                <name>springfivebyexample/${project.build.finalName}</name>
                <build>
                    <from>openjdk:latest</from>
                    <entryPoint>java -Dspring.profiles.active=container -
jar /application/${project.build.finalName}.jar</entryPoint>
                    <assembly>
                        <basedir>/application</basedir>
                        <descriptorRef>artifact</descriptorRef>
                        <inline>
                            <id>assembly</id>
```

```
        <files>
            <file>
<source>target/${project.build.finalName}.jar</source>
            </file>
        </files>
    </inline>
</assembly>
<tags>
    <tag>latest</tag>
</tags>
<ports>
    <port>8080</port>
</ports>
</build>
<run>
    <namingStrategy>alias</namingStrategy>
</run>
    <alias>${project.build.finalName}</alias>
</image>
</images>
</configuration>
</plugin>
```

There are a couple of new configurations here. Let's start with the `<name>` tag—it configures the repository and Docker image name to push to Docker Hub. For this book, we will use `springfivebyexample` as a Docker ID. We can see there is a *slash* as a separator for the repository and image name. The image name for us will be the final project name. Then, we need to configure it.



The Docker ID is free to use, which can be used to access some Docker services, such as Docker Store, Docker Cloud, and Docker Hub. We can find more information at Docker Page (<https://docs.docker.com/docker-id/>).

This configuration should be the same as shown in the following code snippet:

```
<build>
    <finalName>cms</finalName>
    ....
</build>
```

Another important tag is `<entrypoint>`. This is an exec system call instruction when we use the `docker run` command. In our case, we expected the application to run when the container bootstraps. We will execute `java -jar` passing the container as an active profile for Spring.

We need to pass the full path of the Java artifact. This path will be configured on the `<assembly>` tag with the `<basedir>` parameter. It can be any folder name. Also, there is a configuration to the Java artifact path. Usually, this is the target folder which is the result of the compilation. It can be configured in the `<source>` tag.

Finally, we have the `<port>` configuration. The port of the application will be exposed using this tag.

Now, we will create a Docker image by using the following instruction:

```
mvn clean install docker:build
```

It should be executed in the root folder of the project. The goal of the `docker:build` command is to build a Docker image for our project. After the build ends, we can check if the Docker image has been created successfully.

Then, type the following command:

```
docker images
```

The `springfivebyexample/cms` image should be present, as shown in the following screenshot:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
4855a23b3dc1	springfivebyexample/cms:latest	"bin/sh -c 'java ..."	22 minutes ago	Up 22 minutes	0.0.0.0:8080->8080/tcp	cms
e7ff58bc1a4b	mongo-express	"tini -- node app"	19 hours ago	Up 19 hours	0.0.0.0:8081->8081/tcp	friendly_goodall
dd71620d4f76	mongo	"docker-entrypoint..."	20 hours ago	Up 20 hours	0.0.0.0:27017->27017/tcp	mongo

Good. The image is ready. Let's push to the Docker Hub.

Pushing the image to Docker Hub

The Docker Hub is a public repository to store Docker images. It is free, and we will use it for this book. Now, we will push our image to the Docker Hub registry.

The command for that is pretty simple. Type:

```
docker push springfivebyexample/cms:latest
```



I have used the `springfivebyexample` user that I have created. You can test the `docker push` command creating by your own user on Docker Hub and changing the user on the `docker push` command. You can create your Docker ID at Docker Hub (<https://cloud.docker.com/>).

Then, the image will be sent to the registry. That is it.



We can find the image at Docker Hub (<https://store.docker.com/community/images/springfivebyexample/cms>). If you have used your own user, the link will probably change.

Configuring the Docker Spring profile

Before we run our application in a Docker container, we need to create a YAML file to configure a container profile. The new YAML file should be named as `application-container.yaml` because we will use the container profile to run it. Remember, we configured the `entrypoint` on `pom.xml` in the previous section.

Let's create our new file. The file should be the same content as described in the following snippet:

```
spring:  
  data:  
    mongodb:  
      database: cms  
      host: mongodb  
      port: 27017
```

The host must be changed for MongoDB. We have been running the MongoDB container with this name in the *Preparing a MongoDB* section. It is an important configuration, and we need to pay attention at this point. We cannot use `localhost` anymore because the application is running in the Docker container now. The `localhost` in that context means it is in the same container, and we do not have MongoDB in the CMS application container. We need to have one application per container and avoid multiple responsibilities for one container.

Done. In the following section, we will run our first application in the Docker container. It will be amazing. Let's do it.

Running the Dockerized CMS

In the previous section, we have created our file to configure the container profile properly. Now, it is time to run our container. The command is quite simple, but we need to pay attention to the arguments.

The instruction we run should be the same as the following code:

```
docker run -d --name cms --link mongodb:mongodb --net cms-application  
-p 8080:8080 springfivebyexample/cms:latest
```

We have been setting the link for the MongoDB container. Remember, we made this configuration in the YAML file, in the `host` property. During the bootstrapping phase, the application will look for MongoDB instance named `mongodb`. We solved this by using the link command. It will work perfectly.

We can check if our application is healthy by using the `docker ps` command. The output should be like this:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
4855a23b3ac1	springfivebyexample/cms:latest	"bin/sh -c 'java ..."	22 minutes ago	Up 22 minutes	0.0.0.0:8080->8080/tcp	cms
e7ff58bc104b	mongo-express	"tini -- node app"	19 hours ago	Up 19 hours	0.0.0.0:8081->8081/tcp	friendly_goodall
dd71620d4f76	mongo	"docker-entrypoint..."	20 hours ago	Up 20 hours	0.0.0.0:27017->27017/tcp	mongo

In the first line, we have our application container. It is up and running.

Awesome work. Our application is fully containerized and ready to deploy anywhere we want.

Putting in Reactive fashion

We have been creating an amazing application with Spring Boot. The application was built on the traditional web stack present on Spring Framework. It means the application uses the web servers based on Servlet APIs.

The servlet specification was built with the blocking semantics or one-request-per-thread model. Sometimes, we need to change the application architecture because of non-functional requirements. For example, if the application was bought by a huge company, and that company wanted to create a plan to launch the application for the entire world, the volume of requests would probably increase a lot. So, we need to change the architecture to adapt the application structure for cloud environments.

Usually, in a cloud environment, the machines are smaller than traditional data centers. Instead of a big machine, it is popular to use many small machines and try to scale applications horizontally. In this scenario, the servlet spec can be switched to an architecture created upon Reactive Streams. This kind of architecture fits better than servlet for the cloud environments.

Spring Framework has been creating the Spring WebFlux to helps developers to create Reactive Web Applications. Let's change our application architecture to reactive and learn the pretty new Spring WebFlux component.

Reactive Spring

The Reactive Stream Spec is the specification that provides a standard for asynchronous programming for stream processing. It is becoming popular in the programming world nowadays, and Spring introduces it on the framework.

This style of programming is more efficient regarding resources usage and fits amazingly with the new generation of machines with multiple cores.

Spring reactive uses the Project Reactor as the implementation for the Reactive Streams. The Project Reactor is powered by Pivotal and has the very good implementation of the Reactive Streams Spec.

Now, we will deep dive in the reactive module for Spring Boot and create an amazing reactive API and try the new style of the Spring Framework.

Project Reactor

The Project Reactor was created by the Spring and Pivotal teams. This project is an implementation of Reactive Streams for JVM. It is a fully non-blocking foundation and helps developers to create a non-blocking application in the JVM ecosystem.

There is a restriction to using Reactor in our application. The project runs on Java 8 and above. It is important because we will use many lambda expressions in our examples and projects.

The Spring Framework internally uses the Project Reactor as an implementation of Reactive Streams.

Components

Let's look at the different components of the Project Reactor:

- **Publishers:** The publishers are responsible for pushing data elements to the stream. It notifies the subscribers that a new piece of data is coming to the stream.

The publisher interface is defined in the following code snippet:

```
/*
*****
 * Licensed under Public Domain (CC0)
 *
 *
 *
 * To the extent possible under law, the person who associated
CC0 with *
 * this code has waived all copyright and related or
neighboring *
 * rights to this code.
*
*
*
 * You should have received a copy of the CC0 legalcode along
with this *
 * work. If not, see
<http://creativecommons.org/publicdomain/zero/1.0/>.******
*****/
```

```
package org.reactivestreams;

/**
 * A {@link Publisher} is a provider of a potentially
unbounded number of sequenced elements, publishing them
according to
 * the demand received from its {@link Subscriber}(s).
 * <p>
 * A {@link Publisher} can serve multiple {@link Subscriber}s
subscribed {@link #subscribe(Subscriber)} dynamically
 * at various points in time.
 *
 * @param <T> the type of element signaled.
 */
public interface Publisher<T> {

    public void subscribe(Subscriber<? super T> s);

}
```

- **Subscribers:** The subscribers are responsible for making the data flow in the stream. When the publisher starts to send the piece of data on the data flow, the piece of data will be collected by the `onNext (T instance)` method, which is the parametrized interface.
- The subscriber interface is defined in the following code snippet:

```
/*
 * Licensed under Public Domain (CC0)
 *
 *
 *
 * To the extent possible under law, the person who associated
CC0 with *
 * this code has waived all copyright and related or
neighboring *
 * rights to this code.
 *
 *
 *
 * You should have received a copy of the CC0 legalcode along
with this *
 * work. If not, see
<http://creativecommons.org/publicdomain/zero/1.0/>.*
```

```
package org.reactivestreams;

/**
 * Will receive call to {@link #onSubscribe(Subscription)}
once after passing an instance of {@link Subscriber} to {@link
Publisher#subscribe(Subscriber)}.
 * <p>
 * No further notifications will be received until {@link
Subscription#request(long)} is called.
 * <p>
 * After signaling demand:
 * <ul>
 * <li>One or more invocations of {@link #onNext(Object)} up
to the maximum number defined by {@link
Subscription#request(long)}</li>
 * <li>Single invocation of {@link #onError(Throwable)} or
{@link Subscriber#onComplete()} which signals a terminal state
after which no further events will be sent.
 * </ul>
 * <p>
```

```
* Demand can be signaled via {@link Subscription#request(long)} whenever the {@link Subscriber} instance is capable of handling more.  
*  
* @param <T> the type of element signaled.  
*/  
public interface Subscriber<T> {  
    public void onSubscribe(Subscription s);  
  
    public void onNext(T t);  
  
    public void onComplete();  
}
```

Hot and cold

There are two categories of reactive sequences—hot and cold. These functions affect the usage of the implementation directly. Hence, we need to understand them:

- **Cold:** The cold publishers start to generate data only if it receives a new subscription. If there are no subscriptions, the data never comes to the flow.
- **Hot:** The hot publishers do not need any subscribers to generate the data flow. When the new subscriber is registered, the subscriber will only get the new data elements emitted.

Reactive types

There are two reactive types which represent the reactive sequences. The `Mono` objects represent a single value or empty `0|1`. The `Flux` objects represent a sequence of `0|N` items.

We will find many references in our code. The Spring Data reactive repository uses these abstractions in their methods. The `findOne()` method returns the `Mono<T>` object and the `findAll()` returns a `Flux<T>`. The same behavior we will be found in our REST resources.

Let's play with the Reactor

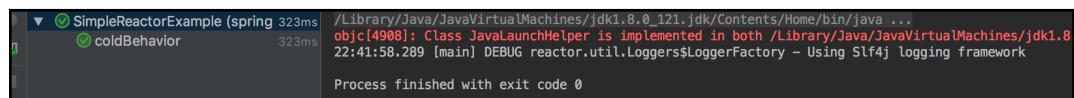
To understand it better, let's play with the Reactor. We will implement and understand the difference between hot and cold publishers in practice.

Cold publishers do not produce any data until a new subscription arrives. In the following code, we will create a cold publisher and the `System.out::println` will never be executed because it does not have any subscribers. Let's test the behavior:

```
@Test
public void coldBehavior() {
    Category sports = new Category();
    sports.setName("sports");
    Category music = new Category();
    sports.setName("music");
    Flux.just(sports, music)
        .doOnNext(System.out::println);
}
```

As we can see, the method `subscribe()` is not present in this snippet. When we execute the code, we will not see any data on the standard print output.

We can execute the method on the IDE. We will be able to see the output of this test. The output should be like this:



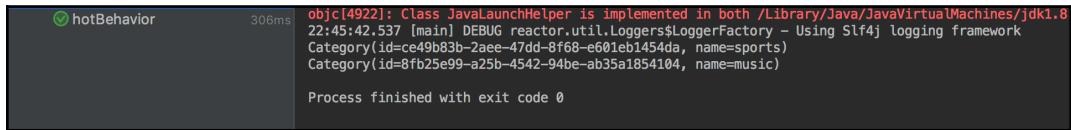
The process has finished, the test passed, and we will not be able to see the print. That is the cold publisher's behavior.

Now, we will subscribe the publisher and the data will be sent on the data flow. Let's try this.

We will insert the `subscribe` instruction after `doOnNext()`. Let's change our code:

```
@Test
public void coldBehaviorWithSubscribe() {
    Category sports = new Category();
    sports.setId(UUID.randomUUID().toString());
    sports.setName("sports");
    Category music = new Category();
    music.setId(UUID.randomUUID().toString());
    music.setName("music");
    Flux.just(sports, music)
        .doOnNext(System.out::println)
        .subscribe();
}
```

The output should be like this:



```
hotBehavior      306ms
objc[4922]: Class JavaLaunchHelper is implemented in both /Library/Java/JavaVirtualMachines/jdk1.8
22:45:42.537 [main] DEBUG reactor.util.Loggers$LoggerFactory - Using Slf4j logging framework
Category(id=ce49b83b-2aee-47dd-8f68-e601eb1454da, name=sports)
Category(id=8fb25e99-a25b-4542-94be-ab35a1854104, name=music)

Process finished with exit code 0
```

In the preceding screenshot, we can see that the publisher pushes the data on the stream after the stream got subscribed. That is the cold publisher behavior after the subscription.

Hot publishers do not depend on any subscribers. The hot publisher will publish data, even if there is no subscriber to receive the data. Let's see an example:

```
@Test
public void testHotPublisher(){
    UnicastProcessor<String> hotSource = UnicastProcessor.create();
    Flux<Category> hotPublisher = hotSource.publish()
        .autoConnect().map((String t) ->
    Category.builder().name(t).build());
    hotPublisher.subscribe(category -> System.out.println("Subscriber 1:
"+ category.getName()));
    hotSource.onNext("sports");
    hotSource.onNext("cars");
    hotPublisher.subscribe(category -> System.out.println("Subscriber 2:
"+category.getName()));
    hotSource.onNext("games");
    hotSource.onNext("electronics");
    hotSource.onComplete();
}
```

Let's understand what happens here. The `UnicastProcessor` is a processor that allows only one `Subscriber`. The processor replays notifications when the subscriber requests. It will emit some data on a stream. The first subscription will capture all the categories, as we will see, because it was registered before the event emissions. The second subscription will capture only the last events because it was registered before the last two emissions.

The output of the preceding code should be:

```
Run SimpleReactorExample.testHotPublisher
SimpleReactorExample (springfive.c: 516ms) 1 test passed - 516ms
/Library/Java/JavaVirtualMachines/jdk1.8.0_121.jdk/Contents/Home/bin/java ...
objc[363]: Class JavaLaunchHelper is implemented in both /Library/Java/JavaVirtualMachines/jdk1.8.0_121.jdk/Contents/Home/bin/java (0x10b7d84c0) and /
19:17:02.293 [main] DEBUG reactor.util.Loggers$LoggerFactory - Using Slf4j logging framework
Subscriber 1: sports
Subscriber 1: cars
Subscriber 1: games
Subscriber 2: games
Subscriber 1: electronics
Subscriber 2: electronics

Process finished with exit code 0
```

Awesome. This is the hot publisher's behavior.

Spring WebFlux

The traditional Java enterprise web applications are based on the servlet specification. The servlet specification before 3.1 is synchronous, which means it was created with blocking semantics. This model was good at the time because computers were big with a powerful CPU and hundreds of gigabytes of memory. Usually, the applications at the time were configured with a big thread pool with hundreds of threads because the computer was designed for this. The primary deployment model at that time was the replica. There are some machines with the same configuration and application deployments.

The developers have been creating applications like this for many years.

Nowadays, most of the applications are deployed in cloud vendors. There are no big machines anymore because the price is much higher. Instead of big machines, there are a number of small machines. It is much cheaper and these machines have a reasonable CPU power and memory.

In this new scenario, the application with the huge thread pools is not effective anymore, because the machine is small and it does not have the power to handle all these threads.

The Spring Team added the support for the Reactive Streams in the framework. This model of programming changes the application deployment and the way to build applications.

Instead of a thread-per-request model, the applications are created with the event-loop model. This model requires a small number of threads and is more efficient regarding resource usage.

Event-loop model

Popularized by the NodeJS language, this model is based on event-driven programming. There are two central concepts: the events which will be enqueued on a queue, and the handlers which keep track of and process these events.

There are some advantages of adopting this model. The first one is the ordering. The events are enqueued and dispatched in the same order in which the events are coming. In some uses cases, this is an important requirement.

The other one is the synchronization. The event-loop must be executed on only one thread. This makes the states easy to handle and avoids the shared state problems.

There is an important piece of advice here. The handlers must not be synchronous. Otherwise, the application will be blocked until the handlers end their workload.

Spring Data for Reactive Extensions

The Spring Data projects have some extensions to work with a reactive foundation. The project provides a couple of implementations based on asynchronous programming. It means the whole stack is asynchronous since database drivers are as well.

The Spring reactive repository supports Cassandra, MongoDB, and Redis as database stores. The repository implementations offer the same behaviors as the non-reactive implementation. There is a **DSL (Domain-Specific Language)** to create domain-specific query methods.

The module uses the Project Reactor as a reactive foundation implementation, but is possible to change the implementation to RxJava as well. Both libraries are production-ready and are adopted by the community. One point to be aware of is that if we change to RxJava, we need to ensure our method returns to `Observable` and `Single`.

Spring Data Reactive

The Spring Data Project has support for the reactive data access. Until now, Spring has support for MongoDB, Apache Cassandra, and Redis, all of which have reactive drivers.

In our CMS application, we will use the MongoDB reactive drivers to give the reactive characteristics for our repositories. We will use the new reactive interface provided by the Spring Data reactive. Also, we need to change the code a little bit. In this chapter, we will do that step by step. Let's start.

Reactive repositories in practice

Before we start, we can check out the full source code at GitHub, or we can perform the following steps.

Now, we are ready to build our new reactive repositories. The first thing that we need to do is add the Maven dependencies to our project. This can be done using pom.xml.

Let's configure our new dependency:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-mongodb-reactive</artifactId>
</dependency>
```

Our project is ready to use reactive MongoDB repositories.

Creating the first Reactive repository

We have a couple of repositories in our CMS project. Now, we need to convert these repositories to reactive ones. The first thing we will do is remove the extension from CrudRepository, which is not necessary anymore. Now, we want the reactive version of that.

We will update the ReactiveMongoRepository interface. The parameters of the interface are the same as the ones we inserted before. The interface should be like this:

```
package springfive.cms.domain.repository;

import
org.springframework.data.mongodb.repository.ReactiveMongoRepository;
import springfive.cms.domain.models.Category;

public interface CategoryRepository extends
ReactiveMongoRepository<Category, String> {
}
```

This is quite similar to the one we created before. We need to extend the new `ReactiveMongoRepository` interface, which contains methods for the CRUD operations and much more. The interface returns `Mono<Category>` or `Flux<Category>`. The methods do not return the entities anymore. It is a common way of programming when the Reactive Stream is adopted.

We need to change the other repositories as well. You can find the full source code on GitHub at: <https://github.com/PacktPublishing/Spring-5.0-By-Example/tree/master/Chapter03/cms-mongodb/src/main/java/springfive/cms/domain/repository>.

Now, we need to change the service layer. Let's do that.

Fixing the service layer

We need to change the service layer to adopt the new reactive programming style. We changed the repository layer, so now we need to fix the compilation problem result because of this change. The application needs to be reactive. Any point of the application can be blocked because we are using the event-loop model. If we do not do this, the application will be getting blocked.

Changing the CategoryService

Now, we will fix the `CategoryService` class. We will change the return type of a couple of methods. Before, we could return the model class, but now we need to change to return `Mono` or `Flux`, similar to what we did in the repository layer.

The new `CategoryService` should be like the implementation shown in the following code snippet:

```
package springfive.cms.domain.service;

import org.springframework.stereotype.Service;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;
import springfive.cms.domain.models.Category;
import springfive.cms.domain.repository.CategoryRepository;
import springfive.cms.domain.vo.CategoryRequest;

@Service
public class CategoryService {
```

```
private final CategoryRepository categoryRepository;

public CategoryService(CategoryRepository categoryRepository) {
    this.categoryRepository = categoryRepository;
}

public Mono<Category> update(String id, CategoryRequest category) {
    return
this.categoryRepository.findById(id).flatMap(categoryDatabase -> {
    categoryDatabase.setName(category.getName());
    return this.categoryRepository.save(categoryDatabase);
});
}

public Mono<Category> create(CategoryRequest request) {
    Category category = new Category();
    category.setName(request.getName());
    return this.categoryRepository.save(category);
}

public void delete(String id) {
    this.categoryRepository.deleteById(id);
}

public Flux<Category> findAll() {
    return this.categoryRepository.findAll();
}

public Mono<Category> findOne(String id) {
    return this.categoryRepository.findById(id);
}

}
```

As we can see, the return types changed in the methods.

The important thing here is that we need to follow the reactive principles. When the method returns only one instance, we need to use `Mono<Category>`. When the method returns one or more instances, we should use `Flux<Category>`. This is essential to follow because developers and Spring containers can then interpret the code correctly.

The `update()` method has an interesting call: `flatMap()`. The project reactor allows us to use a kind of DSL to compose calls. It is very interesting and very useful as well. It helps developers to create code that is easier to understand than before. The `flatMap()` method is usually used to convert the data emitted by `Mono` or `Flux`. In this context, we need to set the new name of the category on the category retrieved from the database.

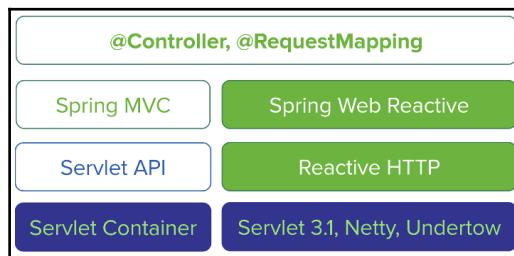
Changing the REST layer

We will make some fixes on the REST layer as well. We changed the service layer, and it caused some compilation problems in our resources classes.

We need to add the new dependency, `spring-web-reactive`. This supports the `@Controller` or `@RestController` annotations for the reactive non-blocking engine. The Spring MVC does not support the reactive extensions, and this module enables developers to use reactive paradigms, as they did before.

`spring-web-reactive` will change many contracts on the Spring MVC foundations, such as `HandlerMapping`, and `HandlerAdapter`, to enable reactive foundations on these components.

The following image can help us to better understand the Spring HTTP layers:



As we can see, `@Controller` and `@RequestMapping` can be used for different approaches in the Spring MVC traditional applications, or by using the Spring web reactive module.

Before we start to change our REST layer, we need to remove the Spring Fox dependencies and annotations in our project. At present, the Spring Fox has no support for reactive applications yet.

The dependencies to remove are:

```
<dependency>
  <groupId>io.springfox</groupId>
  <artifactId>springfox-swagger2</artifactId>
  <version>2.7.0</version>
</dependency>

<dependency>
  <groupId>io.springfox</groupId>
  <artifactId>springfox-swagger-ui</artifactId>
  <version>2.7.0</version>
</dependency>
```

After that, we need to remove the annotations from the Swagger packages, such as `@Api` and `@ApiOperation`.

Now, let's adjust our REST layer.

Adding the Spring WebFlux dependency

Before we start to change our REST layer, we need to add the new dependency to our `pom.xml`.

First, we will remove the Spring MVC traditional dependencies. To do this, we need to remove the following dependency:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

We do not need this dependency anymore. Our application will be reactive now. Then, we need to add the new dependencies described in the following code snippet:

```
<dependency>
  <groupId>io.netty</groupId>
  <artifactId>netty-transport-native-epoll</artifactId>
</dependency>

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
```

`spring-boot-starter-webflux` is a kind of syntax sugar for dependencies. It has the `spring-boot-starter-reactor-netty` dependency, which is the Reactor Netty, as embedded in the reactive HTTP server.

Awesome, our project is ready to convert the REST layer. Let's transform our application into a fully reactive application.

Changing the CategoryResource

We will change the `CategoryResource` class. The idea is pretty simple. We will convert our `ResponseEntity`, which is parametrized with the `models` class to `ResponseEntity` using `Mono` or `Flux`.

The new version of the `CategoryResource` should be like this:

```
package springfive.cms.domain.resources;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;
import springfive.cms.domain.models.Category;
import springfive.cms.domain.service.CategoryService;
import springfive.cms.domain.vo.CategoryRequest;

@RestController
@RequestMapping("/api/category")
public class CategoryResource {

    private final CategoryService categoryService;

    public CategoryResource(CategoryService categoryService) {
        this.categoryService = categoryService;
    }

    @GetMapping(value = "/{id}")
    public ResponseEntity<Mono<Category>> findOne(@PathVariable("id")
```

```
String id){
    return ResponseEntity.ok(this.categoryService.findOne(id));
}

@GetMapping
public ResponseEntity<Flux<Category>> findAll(){
    return ResponseEntity.ok(this.categoryService.findAll());
}

@PostMapping
public ResponseEntity<Mono<Category>> newCategory(@RequestBody
CategoryRequest category){
    return new ResponseEntity<>(this.categoryService.create(category),
HttpStatus.CREATED);
}

@DeleteMapping("/{id}")
@ResponseStatus(HttpStatus.NO_CONTENT)
public void removeCategory(@PathVariable("id") String id){
    this.categoryService.delete(id);
}

@PutMapping("/{id}")
public ResponseEntity<Mono<Category>>
updateCategory(@PathVariable("id") String id,CategoryRequest
category){
    return new
ResponseEntity<>(this.categoryService.update(id,category),
HttpStatus.OK);
}

}
```

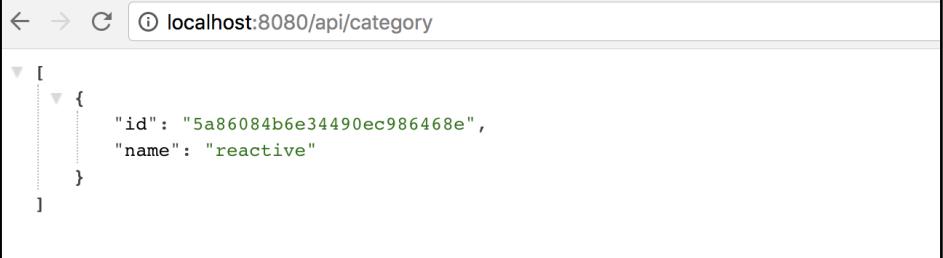
The code is quite similar to what we did before. We have used the `@RequestBody` annotation in the method argument; otherwise, the JSON converter will not work.

The other important characteristic here is the `return` method. It returns `Mono` or `Flux`, which are parameterized types for `ResponseEntity`.

We can test the reactive implementation by using the command line. It will persist the `Category` object on MongoDB. Type the following command on the Terminal:

```
curl -H "Content-Type: application/json" -X POST -d
'{"name":"reactive"}' http://localhost:8080/api/category
```

And then, we can use the following command to check the database. Using the browser, go to `http://localhost:8080/api/category`. The following result should be presented:



```
[{"id": "5a86084b6e34490ec986468e", "name": "reactive"}]
```

Awesome, our reactive implementation is working as expected. Well done!!!

Summary

In this chapter, we have learned a lot of Spring concepts. We have introduced you to Spring Data projects, which help developers to create data access layers as we have never seen before. We saw how easy it is to create repositories with this project.

Also, we presented some relatively new projects, such as Spring WebFlux, which permits developers to create modern web applications, applying the Reactive Streams foundations and reactive programming style in projects.

We have finished our CMS application. The application has the characteristics of a production-ready application, such as database connections, and services which have been well-designed with single responsibilities. Also, we introduced the `docker-maven-plugin`, which provides a reasonable way to create images using the `pom.xml` configurations.

In the next chapter, we will create a new application using the *Reactive Manifesto* based on message-driven applications. See you there.

4

Kotlin Basics and Spring Data Redis

Spring Boot allows developers to create different styles of application. In [Chapter 2, Starting in the Spring World – the CMS Application](#), and [Chapter 3, Persistence with Spring Data and Reactive Fashion](#), we have created a portal application, and now we will create an application based on message-driven architecture. It demonstrates how the Spring Framework fits well in a wide range of application architectures.

In this chapter, we will start to create an application which keeps the tracked hashtags on the Redis database. The application will get hashtags and put them in a couple of queues to our other projects, and consume and handle them appropriately.

As we have been doing in our previous projects, we will continue to use the Reactive Foundation to provide scalable characteristics in the application.

At the end of this chapter, we will have:

- Learned Kotlin basics
- Created the project structure
- Created the Reactive Redis repositories
- Applied some techniques in reactive programming, using the Reactive Redis Client

Let's start right now.

Learning Kotlin basics

The Kotlin language was released officially in February 2016. JetBrains created it and has been developing the language ever since. The company is the owner of the IntelliJ IDEA IDE.

In February 2012, JetBrains made the language open source under the Apache v2 license; the license allows developers to create applications.

The language is one option for **JVM (Java Virtual Machine)** languages such as Clojure and Scala, which means that the language can compile bytecode for JVM. As we will see, Kotlin has many similarities with Scala. Kotlin has the Scala language as a reference, but the JetBrains teams believe that Scala has problems with the compilation time.

Kotlin was becoming an adopted language in the Android world and because of this, in the Google I/O, 2017, the Google Team announced official support for the Android ecosystem. Since then, the language has been growing year by year and increasing in popularity.

Main characteristics of Kotlin

The Kotlin language was designed to maintain the interoperability with Java code. It means we can start to code with Java idioms in the Kotlin file.

The language is statically-typed, and it is an excellent attribute because it can help us find some problems at compilation time. Also, statically-typed languages are much faster than dynamic languages. The IDEs can help developers much better than dynamic languages, as well.

Syntax

The syntax is different from Java syntax. At first glance, it can be a problem but after some hours of playing with Kotlin, it is not a problem at all.

There are two interesting reserved words to understand the usage and concepts:

- **var**: This is a variable declaration. It indicates the variable is mutable and can be reassigned, as developers need.
- **val**: This is a variable declaration which indicates the variable is immutable and cannot be reassigned anymore. This definition is like a final declaration in the Java language.

The variable declarations have a name, and after the desired data type, the colon is necessary in the middle as a separator. If the variable is initialized, the type is not necessary because the compiler can infer the correct data type. Let's try it out to understand it better.

Here is a variable with the data type specified:

```
var bookName: String
```

In this case, we need to keep the data type because the variable is not initialized, then the compiler cannot infer the type. The variable, `bookName`, can be reassigned because of the modifier `var`.

Here is a variable without the data type:

```
val book = "Spring 5.0 by Example"
```

It is not a necessity to declare the data type because we have initialized the variable with the value, `Spring 5.0 by Example`. The compiler can infer the type is a kind of *syntactic sugar*. The variable cannot be reassigned because of the modifier `val`. If we try to reassign the instruction, we will get a compilation error.

The semicolons are optional in Kotlin, the compiler can detect the statement terminator. This is another point where Kotlin diverges from the Java programming language:

```
val book = "Spring 5.0 by Example"  
var bookName: String  
println("Hello, world!")
```

The semicolons were not provided, and the instructions were compiled.



Immutable programming in the Kotlin language is recommended. It performs better on the multi-core environments. Also, it makes the developer's life easier to debug and troubleshoot scenarios.

Semantics

In Kotlin, there are classes and functions. However, there is no method anymore. The `fun` keyword should be used to declare a function.

Kotlin gets some concepts of the Scala language and brings some special classes such as Data classes and Object classes (which we will learn soon). Before that, we will understand how to declare a function in Kotlin. Let's do that!

Declaring functions in Kotlin

There are many variations in function declarations. We will create some declarations to understand the slight difference from Java methods.

Simple function with parameters and return type

This simple function has two parameters and a String as a return type. Take a look at a parameter declaration and observe the order, name and data type.

```
fun greetings(name:String, greeting:String) :String{  
    return greeting + name  
}
```

As we can see, the type of argument which comes after the variable name is the same as on the variable declarations. The return type comes after the arguments list is separated with semicolons. The same function can be declared in the following way in Java:

```
public String greetings(String name, String greeting){  
    return greeting + name;  
}
```

There are some differences here. Firstly, there are semicolons in the Java code, and we can see the order of the methods and functions declarations.

Simple function without return

Let's understand how we can construct functions without a return value, the following function will not return any value:

```
fun printGreetings(name:String, greeting:String) :Unit{  
    println(greeting + name)  
}
```

There is one difference, in this case, the `Unit` was introduced; this type of object corresponds to `void` in Java language. Then, in the preceding code, we have a function without a return. The `Unit` object can be removed if you want the compiler to understand the function has no return value.

Single expressions functions

When the function has a single expression we can remove the curly braces, the same as in Scala, and the function body should be specified after the `=` symbol. Let's refactor our first function, as follows:

```
fun greetings(name:String, greeting:String) = greeting + name
```

We can remove the `return` keyword, as well. Our function is pretty concise now. We removed `return` and the type of return as well. As we can see, the code is more readable now. If you want, the return type can be declared too.

Overriding a function

To override a function on Kotlin, it is necessary to put an `override` keyword on the function declaration, and the base function needs to have the `open` keyword as well.

Let's look at an example:

```
open class Greetings {  
    open fun greeting() {}  
}  
  
class SuperGreeting() : Greetings() {  
    override fun greeting() {  
        // my super greeting  
    }  
}
```

This way is more explicit than Java, it increases the legibility of the code as well.

Data classes

Data classes are the right solution when we want to hold and transfer data between system layers. Like in Scala, these classes offer some built-in functionalities such as getters/setters, equals and hashCode, toString method and the copy function.

Let's create an example for that:

```
data class Book(val author:String, val name:String, val  
description:String, val new:Boolean = false)
```

We have some interesting things in the code. The first thing we notice is that all of the attributes are immutable. It means there are no setters for all of them. The second is that in the class declaration, we can see a list of attributes. In this case, Kotlin will create a constructor with all attributes present in this class and because they are `val` it means final attributes.

In this case, there is no default constructor anymore.

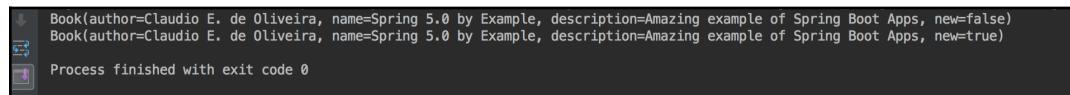
Another interesting feature in Kotlin is that it enables developers to have default values on constructors, in our case the `new` attribute, if omitted, will assume the `false` value. We can get the same behavior in the parameters list in functions as well.

Finally, there is a fantastic way to copy objects. The `copy` method allows developers to copy objects with named parameters. This means we can change only attributes as we need. Let's take a look at an example:

```
fun main(args : Array<String>) {
    val springFiveOld = Book("Claudio E. de Oliveira", "Spring 5.0 by Example", "Amazing example of Spring Boot Apps", false)
    val springFiveNew = springFiveOld.copy(new = true)
    println(springFiveOld)
    println(springFiveNew)
}
```

In the first object, we have created a book instance with `false` for the `new` attribute, then we copied a new object with `true` for the `new` attribute, and the other attributes are not changed. Goodbye to the complex clone logic and nice to meet the new way to copy objects.

The output of this code should look like the following:



```
Book(author=Cláudio E. de Oliveira, name=Spring 5.0 by Example, description=Amazing example of Spring Boot Apps, new=false)
Book(author=Cláudio E. de Oliveira, name=Spring 5.0 by Example, description=Amazing example of Spring Boot Apps, new=true)
Process finished with exit code 0
```

As we can see, only the `new` attribute is changed and the `toString` function was generated in good shape as well.

There are some restrictions on Data classes. They cannot be abstract, open, sealed, or inner.

Objects

The singleton pattern is commonly used in applications, and Kotlin provides an easy way to do that without much boilerplate code.

We can instruct Kotlin to create a singleton object using the `object` keyword. Once again, Kotlin used Scala as a reference because there are the same functionalities in the Scala language.

Let's try it:

```
object BookNameFormatter{
    fun format(book: Book):String = "The book name is" + book.name
}
```

We have created a formatter to return a message with the book name. Then, we try to use this function:

```
val springFiveOld = Book("Claudio E. de Oliveira","Spring 5.0 by
Example","Amazing example of Spring Boot Apps",false)
BookNameFormatter.format(springFiveOld)
```

The function format can be called in a static context. There is no instance to call the function because it is a singleton object.

Companion objects

A **companion object** is an object which is common for all instances of that class. It means there are many instances of a book, for example, but there is a single instance of their companion object. Usually, the developers use companion objects as a factory method. Let's create our first companion object:

```
data class Book(val author:String,val name:String,val
description:String,val new:Boolean = false{

    companion object {
        fun create(name:String,description: String,author: String):Book{
            return Book(author,name,description)
        }
    }
}
```

If the name of the companion object was omitted, the function could be called in a singleton way, without an instance, like this:

```
val myBookWithFactory = Book.create("Claudio E. de Oliveira","Spring
5.0 by Example","Amazing example of Spring Boot Apps")
```

It is like an object behavior. We can call it in a static context.

Kotlin idioms

Kotlin idioms are a kind of syntax sugar for Java programmers. It is a collection of pieces of code which help developers to create a concise code in Kotlin languages. Let's take a look at common Kotlin idioms.

String interpolation

Kotlin supports string interpolation, it is a little bit complex to do it in the Java language but it is not a problem for Kotlin. We do not require a lot of code to do this task as Kotlin supports it natively. It makes the code easier to read and understand. Let's create an example:

```
val bookName = "Spring 5.0"
val phrase = "The name of the book is $bookName"
```

As we can see, it is a piece of cake to interpolate strings in Kotlin. Goodbye `String.format()` with a lot of arguments. We can use `$bookName` to replace the `bookName` variable value. Also, we can access the functions present in objects, but for that, we need to put curly braces. Check the following code:

```
val springFiveOld = Book("Claudio E. de Oliveira", "Spring 5.0 by
Example", "Amazing example of Spring Boot Apps", false)
val phrase = "The name of the book is ${springFiveOld.name}"
```

Thanks, Kotlin we appreciate this feature.

Smart Casts

Kotlin supports the feature called Smart Casts which enables developers to use the cast operators automatically. After checking the variable type, in Java, the cast operator must be explicit. Let's check it out:

```
fun returnValue(instance: Any): String {
    if (instance is String) {
        return instance
    }
    throw IllegalArgumentException("Instance is not String")
}
```

As we can see, the cast operator is not present anymore. After checking the type, Kotlin can infer the expected type. Let's check the Java version for the same piece of code:

```
public String returnValue(Object instance) {  
    if (instance instanceof String) {  
        String value = (String) instance;  
        return value;  
    }  
    throw new IllegalArgumentException("Instance is not String");  
}
```

It makes the cast safer because we do not need to check and apply the cast operator.

Range expressions

Range expressions permit developers to work with ranges in `for` loops and `if` comparison. There are a lot of ways to work with ranges in Kotlin. We will take a look at most of the common ones here.

Simple case

Let's look at one simple case:

```
for (i in 1..5){  
    println(i)  
}
```

It will iterate from 1 to 5 inclusive because we have used them in the `in` keyword.

The until case

We also can use the `until` keyword in `for` loops, in this case, the end element will be excluded from the interaction. Let's see an example:

```
for (i in 1 until 5) {  
    println(i)  
}
```

In this case, the 5 value will not be printed on the console, because the end element is not included in the interaction.

The downTo case

The `downTo` keyword enables developers to interact with the numbers in reverse order. The instruction is self-explanatory, as well. Let's see it in practice:

```
for (i in 5 downTo 1) {  
    println(i)  
}
```

It is pretty easy as well. The interaction will occur in the reverse order, in this case, the value 1 will be included. As we can see, the code is pretty easy to understand.

Step case

Sometimes we need to interact over values but with the arbitrary steps, not one by one, for example. Then we can use the `step` instruction. Let's practice:

```
for (i in 1..6 step 2) {  
    print(i)  
}
```

Here, we will see the following output: 135, because the interaction will start on the 1 value and will be increased by two points.

Awesome. The Kotlin ranges can add more readability to our source code and help to increase the quality of code as well.

Null safety

Kotlin has amazing stuff to work with null references. The null reference is a nightmare for Java developers. The Java 8 has an `Optional` object, which helps developers work with nullable objects, but is not concise like in Kotlin.

Now, we will explore how Kotlin can help developers to avoid the `NullPointerException`. Let's understand.

The Kotlin type system makes a distinction between references which can hold null and those which cannot hold null. Due to this, the code is more concise and readable because it is a kind of advice for developers.

When the reference does not allow null, the declaration should be:

```
var myNonNullString:String = "my non null string"
```

The preceding variable cannot be assigned to a null reference, if we do this, we will get a compilation error. Look how easy the code is to understand.

Sometimes, we need to allow for a variable to have null references, in these cases, we can use the ? as an operator, such as follows:

```
var allowNull:String? = "permits null references"
```

Easy. Pay attention to a variable declaration on the ? operator, it makes the variable accept null references.

There are two different ways to avoid the `NullPointerException` in Kotlin. The first one can be called **safe calls**, and the other can be called the **Elvis Operator**. Let's take a look at those.

Safe calls

The safe call can be written using the `.?`. It can be called when the reference holds a non-null value when the value holds a null reference then the null value will be returned:

```
val hash:TrackedHashTag? = TrackedHashTag(hashTag="java", queue="java")
val queueString = hash?.queue
```

When the `hash?` holds null, the null value will be assigned to a `queueString` attribute. If the `hash?` has a valid reference, the `queue` attribute will be assigned to a `queueString` attribute.

Elvis operator

It can be used when developers expect to return a default value when the reference is null:

```
val hash:TrackedHashTag? = TrackedHashTag(hashTag="java", queue="java")
val queueString = hash?.queue ?: "unrecognized-queue"
```

When the value holds null, the default value will be returned.

Time to use Kotlin in the real world. Let's begin.

Wrapping it up

Now, we can use the basics of the Kotlin language. We saw some examples and practiced a little bit.

We looked at the main concepts of Kotlin. We have learned how data classes can help developers to transfer data between application layers. Also, we learned about singleton and companion objects. Now we can try to create a real project with the pretty new support from Spring Framework.

In the next sections, we will create a project using the Kotlin language, for now, we can forget about the Java language.

Creating the project

Now, we have a good idea about how we can use programming in Kotlin language. In this section, we will create the basic structure for our new project in which the main feature is consuming the Twitter stream. Let's do that.

Project use case

Before we start to code, we need to track the application requirements. The application is message-driven, we will use a broker to provide the messaging infrastructure. We choose the RabbitMQ broker because it provides reliability, high availability, and clustering options. Also, the RabbitMQ is a popular choice for the modern message-driven applications.

The software is powered by the Pivotal company, the same company which maintains Spring Framework. There is a huge community which supports the project.

We will have three projects. These three projects will collect the Twitter stream and send it to a recipient to show Tweets in a formatted way to the end user.

The first one, which will be created in this chapter, will be responsible for keeping the tracked hashtags on the Redis cache.

When the new hashtags are registered, it will send a message to the second project which will start to consume the Twitter stream and redirect it to the desired queue. This queue will be consumed by the other project which will format the Tweet, and finally, show them to the end user.

We will have three microservices. Let's create these things.

Creating the project with Spring Initializr

We have learned how to use the Spring Initializr page. We will go to the page and then select the following modules:

- Reactive Web
- Reactive Redis

The page content should look like this:

SPRING INITIALIZR bootstrap your application now

Generate a with and Spring Boot

Project Metadata

Artifact coordinates

Group

Artifact

Dependencies

Add Spring Boot Starters and dependencies to your application

Search for dependencies

Selected Dependencies

Don't know what to look for? Want more options? [Switch to the full version.](#)

We can choose the group and artifact. There is no problem with using the different name. Then, we can click on **Generate Project** and wait until the download ends.

Adding Jackson for Kotlin

We need to add Jackson for Kotlin dependencies for Maven projects. In fact, we need a Kotlin standard library on our `pom.xml`. Also, we need to put `jackson-module-kotlin`, it allows us to work with JSON on Kotlin, there are some differences from Java in these parts.

This part is pretty simple, and we will add these following dependencies in the dependencies sections in `pom.xml`. The dependencies are as follows:

```
<dependency>
    <groupId>com.fasterxml.jackson.module</groupId>
    <artifactId>jackson-module-kotlin</artifactId>
    <version>${jackson.version}</version>
</dependency>
```

Now, we have the dependencies configured, and we can set the plugins to compile the Kotlin source code. In the next section, we will do that.

Looking for the Maven plugins for Kotlin

The project was created with Kotlin configured successfully. Now, we will take a look at the Maven plugin in our `pom.xml`. The configuration is necessary to instruct Maven on how to compile the Kotlin source code and add in the artifacts.

We will add the following plugins in the plugins section:

```
<plugin>
    <artifactId>kotlin-maven-plugin</artifactId>
    <groupId>org.jetbrains.kotlin</groupId>
    <version>${kotlin.version}</version>
    <configuration>
        <jvmTarget>1.8</jvmTarget>
    </configuration>
    <executions>
        <execution>
            <id>compile</id>
            <phase>process-sources</phase>
            <goals>
                <goal>compile</goal>
            </goals>
        </execution>
        <execution>
            <id>test-compile</id>
```

```
<phase>process-test-sources</phase>
<goals>
    <goal>test-compile</goal>
</goals>
</execution>
</executions>
</plugin>
```

There is one more thing to do. Take a look how Maven configures the path for our Kotlin code. It is easy peasy. Look at the following:

```
<build>

<sourceDirectory>${project.basedir}/src/main/kotlin<
/sourceDirectory<testSourceDirectory>${project.basedir}/src/
test/kotlin</testSourceDirectory>

....<

</build>
```

We added our Kotlin folders in the source paths.

Awesome, the project structure is ready, and we can start coding!

Creating a Docker network for our application

To create isolation for our application, we will create a custom Docker network. This network was created using the bridge driver. Let's do that using the following command:

```
docker network create twitter
```

Good, now we can check the network list by typing the following command:

```
docker network list
```

The Twitter network should be on the list, like the following:

```
ubuntu@ubuntu-xenial:~$ docker network list
NETWORK ID     NAME      DRIVER      SCOPE
d2bb065f5d06   bridge    bridge      local
5a8485d8da42   cms-application    bridge      local
1d4b5dc3ec8b   host      host      local
46b59abc89c2   none      null      local
fb27a7381539   twitter    bridge      local
```

The last one is our Twitter network. Let's pull the Redis image from the Docker Hub. Take a look at the next section.

Pulling the Redis image from the Docker Hub

The first thing we need to do is download the Redis image from the Docker Hub. To do that, it is necessary to execute the following command:

```
docker pull redis:4.0.6-alpine
```

We have used the alpine version from Redis because it is smaller than the others and has a reasonable security. While the image is downloaded, we can see the downloading status progress.

We can check the result using the following command:

```
docker images
```

The result should look like the following:



A terminal window showing the output of the docker images command. It lists three images: redis, postgres, and mongo, along with their tags, image IDs, creation dates, and sizes.

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
redis	4.0.6-alpine	ed8544cc83de	5 days ago	26.9MB
postgres	9.6.6-alpine	e20de7998161	3 weeks ago	37.8MB
mongo	3.4.10	d22888af0ce0	5 weeks ago	361MB

Take a look at the images downloaded. The Redis must be on the list.

Awesome, now we will start the Redis instance.

Running the Redis instance

The image was downloaded, then we will start the Redis instance for our application. The command can be:

```
docker run -d --name redis --net twitter -p 6379:6379 redis:4.0.6-alpine
```

We have interesting attributes here. We named our Redis instance with `redis`, it will be useful for running our application in containers in the next chapters. Also, we exposed the Redis container ports to the host machine, the command argument used for that is `-p`. Finally, we attached the container to our Twitter network.

Good, the Redis instance is ready to use. Let's check out the Spring Data Reactive Redis stuff.

Configuring the redis-cli tool

There is an excellent tool to connect with the Redis instance which is called `redis-cli`. There are some Docker images for that, but we will install it on our Linux machine.

To install it, we can execute the following command:

```
sudo apt-get install redis-tools -y
```

Excellent, now we can connect and interact with our Redis container. The tool can perform the read and write instructions, then we need to be careful to avoid instructions unintentionally.

Let's connect. The default configuration is enough for us because we have exported the port 6379 on the `run` instruction. Type the following command in the Terminal:

```
redis-cli
```

Then we will connect with our running instance. The command line should display the Redis host and port, like the following screenshot:

```
ubuntu@ubuntu-xenial:~$ redis-cli  
127.0.0.1:6379> 
```

Excellent, the client is configured and tested.

Now, we will execute some Redis commands on our container.

Understanding Redis

Redis is an open source in-memory data structure. Redis fits well for a database cache and is not common, but it can be used as a message broker using the publish-subscribe feature, it can be useful to decouple applications.

There are some interesting features supported by Redis such as transactions, atomic operations, and support for time-to-live keys. Time-to-live is useful for giving a time for the key, the eviction strategy is always hard to implement, and Redis has a built-in solution for us.

Data types

There are a lot of supported data types by Redis. The most common ones are strings, hashes, lists, and sorted sets. We will understand each of these a little bit because it is important to help us to choose the correct data type for our use case.

Strings

Strings are the more basic data type of Redis. The string value can be at max 512 MB in length. We can store it as a JSON in the value of the key, or maybe as an image as well because the Redis is binary safe.

Main commands

Let's look at some important commands we would need:

- **SET**: It sets the key and holds the value. It is a simple and basic command of Redis. Here's an example:

```
SET "user:id:10" "joe"
```

The return of the command should be `OK`. It indicates the instruction has been executed with success.

- **GET**: This command gets the value of the requested key. Remember `GET` can only be used with a string data type:

```
GET "user:id:10"
```

As we can see, the return of that command should be `joe`.

- **INCR**: The `INCR` command increments the key by one. It can be useful to handle sequential numbers atomically in distributed systems. The number increment will be returned as a command output:

```
SET "users" "0"
INCR "users"
GET "users"
```

As we can see, the `INCR` command returned `1` as a command output and then we can check this using the `GET` and obtain the value.

- **DECR:** The DECR command is opposite of INCR, it will decrement the value atomically as well:

```
GET "users"  
DECR "users"  
GET "users"
```

The value of the `users` key was decremented by one and then transformed to 0.

- **INCRBY:** It will increment the value of the key by the argument. The new incremented value will be returned:

```
GET "users"  
INCRBY "users" 2  
GET "users"
```

The new value was returned as a command output.

Lists

Lists are simple lists of strings. They are ordered by the insertion order. Redis also offers instructions to add new elements at the head or tail of the list.

Lists can be useful for storing groups of things, groups of categories, for example, grouped by the `categories` key.

Main commands

LPUSH: Insert the new element at the head of the key. The command also supports multiple arguments, in this case, the values will be stored in the reverse order as we passed on the arguments.

Here are some command examples:

```
LPUSH "categories" "sports"  
LPUSH "categories" "movies"  
LRANGE "categories" 0 -1
```

Take a look at the `LRANGE` output, as we can see the value of the `movie` is the first one on the list because the `LPUSH` inserted the new element on the head.

RPUSH: Insert the new element at the tail of the key. The command supports multiple arguments as well, in this case, the values will respect the respective order.

Here are some command examples:

```
RPUSH "categories" "kitchen"
RPUSH "categories" "room"
LRANGE "categories" 0 -1
```

As we can see, in the LRANGE output, the new values are inserted at the tail of the values. It is the behavior of the RPUSH command.

LSET: It sets the element on the requested index.

Here are some command examples:

```
LSET "categories" 0 "series"
LRANGE "categories" 0 -1
```

The new value of the zero index is series. The LSET command does that for us.

LRANGE: It returns the specified elements of the key. The command arguments are the key, the start index, and finally the stop element. The -1 on the stop argument will return the whole list:

```
LRANGE "categories" 0 2
LRANGE "categories" 0 -1
```

As we can see, the first command will return three elements because the zero index will be grouped.

Sets

A **set** is a collection of strings. They have a property which does not allow repeated values. It means that if we add the pre-existing value on the sets, it will result in the same element, in this case, the advantage is not necessary to verify if the element exists on the set. Another important characteristic is that the sets are unordered. This behavior is different from the Redis lists. It can be useful in different use cases such as count the unique visitor, track the unique IPs, and much more.

Main commands

The following are the main commands listed with their usages:

- **SADD**: It adds the element in a requested key. Also, the return of this command is the number of the element added to the set:

```
SADD "unique-visitors" "joe"  
SADD "unique-visitors" "mary"
```

As we can see, the command returned one because we added one user each time.

- **SMEMBERS**: It returns all the members of a requested key:

```
SMEMBERS "unique-visitors"
```

The command will return `joe` and `mary` because those are the values stored in the `unique-visitors` key.

- **SCARD**: It returns the numbers of elements of a requested key:

```
SCARD "unique-visitors"
```

The command will return the number of elements stored in the requested keys, in this case, the output will be 2.

Spring Data Reactive Redis

Spring Data Redis provides an easy way to interact with the Redis Server from Spring Boot Apps. The project is part of the Spring Data family and provides high-level and low-level abstractions for the developers.

The Jedis and Lettuce connectors are supported as a driver for this project.

The project offers a lot of features and facilities to interact with Redis. The Repository interfaces are supported as well. There is a CrudRepository for Redis like in other implementations, Spring Data JPA, for example.

The central class for this project is the `RedisTemplate` which provides a high-level API to perform Redis operations and serialization support. We will use this class to interact with set data structures on Redis.

The Reactive implementation is supported by this project, these are important characteristics for us because we are looking for Reactive implementations.

Configuring the `ReactiveRedisConnectionFactory`

To configure the `ReactiveRedisConnectionFactory`, we can use the `application.yaml` file, because it is easier to maintain and centralize our configuration.

The principle is the same as other Spring Data Projects, we should provide the host and port configurations in the `application.yaml` file, as follows:

```
spring:  
  redis:  
    host: localhost  
    port: 6379
```

In the preceding configuration file, we point the Redis configuration to the `localhost`, as we can see. The configuration is pretty simple and easy to understand as well.

Done. The connection factory is configured. The next step is to provide a `RedisTemplate` to interact with our Redis instance. Take a look at the next section.

Providing a `ReactiveRedisTemplate`

The main class from Spring Data Redis is the `ReactiveRedisTemplate`, then we need to configure and provide an instance for the Spring container.

We need to provide an instance and configure the correct serializer for the desired `ReactiveRedisTemplate`. `Serializers` is the way Spring Data Redis uses to serialize and deserialize objects from raw bytes stored in Redis in the `Key` and `Value` fields.

We will use only the `StringRedisSerializer` because our `Key` and `Value` are simple strings and the Spring Data Redis has this serializer ready for us.

Let's produce our `ReactiveRedisTemplate`. The implementation should look like the following:

```
package springfive.twittertracked.infra.redis

import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import
org.springframework.data.redis.connection.ReactiveRedisConnectionFactory
ry
import org.springframework.data.redis.core.ReactiveRedisTemplate
import
org.springframework.data.redis.serializer.RedisSerializationContext

@Configuration
open class RedisConfiguration {

    @Bean
    open fun
reactiveRedisTemplate(connectionFactory:ReactiveRedisConnectionFactory
):
                    ReactiveRedisTemplate<String, String>
    {
        return ReactiveRedisTemplate(connectionFactory,
RedisSerializationContext.string())
    }
}
```

Awesome. That is our first code using Kotlin in the Spring Framework. The keyword `open` is the opposite of Java's `final` keyword. It means this function can be inherited from this class. By default, all classes in Kotlin are final. Spring Framework requires non-final functions on `@Bean` on the `@Configuration` class and then we need to insert `open`.

We received `ReactiveRedisConnectionFactory` as a parameter. Spring knows which we produced in the `application.yaml` file using the configurations for Redis. Then the container can inject the factory.

Finally, we declare `ReactiveRedisTemplate<String, String>` as a return value for our function.

Interesting work, we are ready to work with our Redis template. Now, we will implement our first repository for Redis. See you in the next section.

Creating Tracked Hashtag repository

We have created the `ReactiveRedisTemplate`, then we can use this object in our repository implementation. We will create a simple repository to interact with Redis, remember the repository should be reactive, it is an important characteristic of our application. Then we need to return `Mono` or `Flux` to make the repository Reactive. Let's look at our repository implementation:

```
package springfive.twittertracked.domain.repository

import org.springframework.data.redis.core.ReactiveRedisTemplate
import org.springframework.stereotype.Service
import reactor.core.publisher.Flux
import reactor.core.publisher.Mono
import springfive.twitterconsumer.domain.TrackedHashTag

@Service
class TrackedHashTagRepository(private val redisTemplate:
ReactiveRedisTemplate<String, String>) {

    fun save(trackedHashTag: TrackedHashTag): Mono<TrackedHashTag>?
    {
        return this.redisTemplate
            .opsForSet().add("hash-tags",
        "${trackedHashTag.hashTag}:${trackedHashTag.queue}")
            .flatMap { Mono.just(trackedHashTag) }
    }

    fun findAll(): Flux<TrackedHashTag> {
        return this.redisTemplate.opsForSet().members("hash-
tags").flatMap { el ->
            val data = el.split(":")
            Flux.just(TrackedHashTag(hashTag = data[0],queue = data[1]))
        }
    }
}
```

We received the `ReactiveRedisTemplate<String, String>` as an injection on our class, the Spring Framework can detect the constructor and inject the correct implementation.

For now, we need these two functions. The first one is responsible for inserting our entity, `TrackedHashTag` on the set structure from Redis. We add the value of the hash-tags key on Redis. This function returns a `Mono` with the `TrackedHashTag` value. Pay attention to the `save` function. We have created a pattern for our value, the pattern follows the hashtag, queue where the hashtag is the value to gather Tweets and the queue we will use in the next sections to send to a RabbitMQ queue.

The second function returns all values from the hash-tags key, it means all tracked hashtags from our system. Moreover, we need to do some logic to create our model, `TrackedHashTag`, as well.

The repository is finished, now we can create our service layer to encapsulate the repository. Let's do that in the next section.

Creating the service layer

Our repository is ready to use, now we can create our service layer. This layer is responsible for orchestrating our repository calls. In our case, it is pretty simple but in some complex scenarios, it can help us to encapsulate the repository calls.

Our service will be called `TrackedHashTagService`, which will be responsible for interacting with our repository created previously. The implementation should look like the following:

```
package springfive.twittertracked.domain.service

import org.springframework.stereotype.Service
import springfive.twitterconsumer.domain.TrackedHashTag
import
springfive.twitterconsumer.domain.repository.TrackedHashTagReposit
ory

@Service
class TrackedHashTagService(private val repository:
TrackedHashTagRepository) {

    fun save(hashTag:TrackedHashTag) = this.repository.save(hashTag)

    fun all() = this.repository.findAll()

}
```

Well done. Here, there is basic stuff. We have the construct which injects our repository to interact with Redis. The interesting point here is the function declarations. There is not a body and return type because the Kotlin compiler can infer the return type, it helps the developer to avoid writing boilerplate code.

Exposing the REST resources

Now, we have created the repository and service layer, and we are ready to expose our service through HTTP endpoints:

```
package springfive.twittertracked.domain.resource

import org.springframework.web.bind.annotation.*
import springfive.twitterconsumer.domain.TrackedHashTag
import
springfive.twitterconsumer.domain.service.TrackedHashTagService

@RestController
@RequestMapping("/api/tracked-hash-tag")
class TrackedHashTagResource(private val
service:TrackedHashTagService) {

    @GetMapping
    fun all() = this.service.all()

    @PostMapping
    fun save(@RequestBody hashTag:TrackedHashTag) =
this.service.save(hashTag)

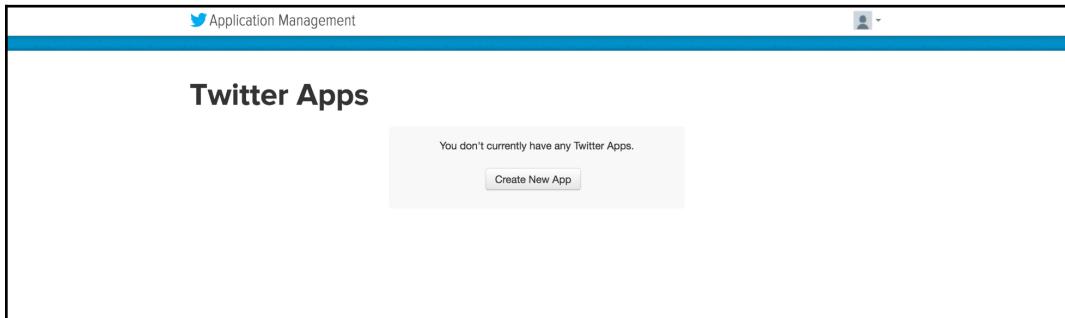
}
```

The code is pretty concise and simple. Take a look at how concise this piece of code is. The preceding code is an example of how Kotlin helps developers to create readable codes. Thanks, Kotlin.

Creating a Twitter application

For this project, we will need to configure an application on the Twitter platform. It is necessary, because we will use Twitter's API to search Tweets, for example, and the Twitter account is the requirement for that. We will not explain how to create a Twitter account. There are plenty of articles about that on the internet.

After the Twitter account is created, we need to go to <https://apps.twitter.com/> and create a new app. The page is quite similar to the following screenshot:



We will click on the **Create New App** button to start the creation process. When we click on that button, the following page will be displayed. We need to fill the required fields and accept the Twitter agreements:

A screenshot of a 'Create an application' form. The title is 'Create an application'. The 'Application Details' section contains fields for 'Name' (placeholder: 'My New App'), 'Description' (placeholder: 'A new application'), 'Website' (placeholder: 'http://myapp.com'), and 'Callback URL' (placeholder: 'http://myapp.com/callback'). Below these fields are detailed descriptions of their purposes. The 'Agreements' section is partially visible at the bottom.

We can choose the application name, fill in the description, and website. These details are up to you.

Then, we need to accept the agreements and click on **Create your Twitter application**:



Awesome job. Our Twitter application is almost ready to use.

Now, we just need to configure the application for usage.

We need to check if our Keys and Access Tokens are correctly configured. Let's click on the **Keys and Access Tokens** tab and check the values, shown as follows:

The screenshot shows the 'Keys and Access Tokens' tab for the 'springfivebyexample' application. It displays the following configuration:

Setting	Value
Consumer Key (API Key)	gupfxwn43NBTdxCD3Tsf1JgMu
Consumer Secret (API Secret)	pH4uM5LIYxKzfJ7huYRwfbaFXn7ooK01LmqCP69QV9a9kZrHw5
Access Level	Read-only (modify app permissions)
Owner	springfivebyexa
Owner ID	940015005860290560

As we can see, there are some important configurations in the preceding screenshot. The **Consumer Key** and **Consumer Secret** are mandatory to authenticate with Twitter APIs. Another important point here is the **Access Level**; be sure it is configured as read-only, as in the preceding screenshot, we will not do write actions on Twitter.

Let's Dockerize it.

Awesome. We have the system which keeps the tracked hashtags on the Redis instance. The application is fully Reactive and has no blocking threads.

Now, we will configure the Maven plugin to generate the Docker images. The configuration is quite similar to what we did in Chapter 3, *Persistence with Spring Data and Reactive Fashion*. However, now we will create a first container which we will run with the Kotlin language. Let's do that.

Configuring pom.xml

Now, we will configure our `pom.xml` to be able to generate our Docker image. The first thing we need to change is our final name artifact because Docker images do not allow the `-` character, then we need to configure properly.

The configuration is pretty simple, put the `<finalName>` tag on the `<build>` node. Let's do that:

```
<build>
    <finalName>tracked_hashtag</finalName>
    ...
</build>
```

Good. We have configured the final name properly to generate the Docker image correctly. Now, we will configure the Maven Docker plugin to generate the Docker image by the Maven goal.

In the plugins section inside the build node, we should put in the following plugin configuration:

```
<plugin>
    <groupId>io.fabric8</groupId>
    <artifactId>docker-maven-plugin</artifactId>
    <version>0.21.0</version>
    <configuration>
        <images>
            <image>
                <name>springfivebyexample/${project.build.finalName}</name>
                <build>
                    <from>openjdk:latest</from>
                    <entryPoint>java -Dspring.profiles.active=container -jar
                        /application/${project.build.finalName}.jar</entryPoint>
                    <assembly>
```

```
<basedir>/application</basedir>
<descriptorRef>artifact</descriptorRef>
<inline>
    <id>assembly</id>
    <files>
        <file>
<source>target/${project.build.finalName}.jar</source>
        </file>
    </files>
</inline>
</assembly>
<tags>
    <tag>latest</tag>
</tags>
<ports>
    <port>9090</port>
</ports>
</build>
<run>
    <namingStrategy>alias</namingStrategy>
</run>
    <alias>${project.build.finalName}</alias>
</image>
</images>
</configuration>
</plugin>
```

The configuration is pretty simple. We did this before. In the configuration section, we configured from the image, in our case the `openjdk:latest`, Docker entry point and exposed ports as well.

Let's create our Docker image in the next section.

Creating the image

Our project was previously configured with the Maven Docker plugin. We can generate the Docker image with the Maven Docker plugin using the `docker:build` goal. Then, it is time to generate our Docker image.

To generate the Docker image, type the following command:

```
mvn clean install docker:build
```

Now, we must wait for the Maven build and check if the Docker image was generated with success.

Check the Docker images and we should see the new image generated. To do this, we can use the `docker images` command:

```
docker images
```

Right, we should see the `springfivebyexample/tracked_hashtag:latest` on the image list, like the following screenshot:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
springfivebyexample/tracked_hashtag	latest	54d51eba299a	About an hour ago	766MB
redis	4.0.6-alpine	ed8544cc83de	8 days ago	26.9MB
springfivebyexample/cms	latest	a2609f25ded1	2 weeks ago	773MB

Awesome, our Docker image is ready to run with our first Spring Boot Application in the Kotlin language. Let's run it right now.

Running the container

Let's run our container. Before that, we need to keep in mind some things. The container should be run on the Twitter network to be able to connect to our Redis instance which is running on the Twitter network as well. Remember the `localhost` address for Redis does not work anymore when running in the containers infrastructure.

To run our container, we can execute the following command:

```
docker run -d --name hashtag-tracker --net twitter -p 9090:9090  
springfivebyexample/tracked_hashtag
```

Congratulations, our application is running in the Docker container and connected to our Redis instance. Let's create and test our APIs to check the desired behaviors.

Testing APIs

Our container is running. Now, we can try to call the APIs to check the behaviors. In this part, we will use the `curl` command line. The `curl` allows us to call APIs by the command line on Linux. Also, we will use `jq` to make the JSON readable on the command line, if you do not have these, look at the Tip Box to install these tools.

Let's call our create API, remember to create we can use the `POST` method in the base path of API. Then type the following command:

```
curl -H "Content-Type: application/json" -X POST -d
'{"hashTag": "java", "queue": "java"}' \
http://localhost:9090/api/tracked-hash-tag
```

There are interesting things here. The `-H` argument instructs `curl` to put it in the request headers and `-d` indicates the request body. Moreover, finally, we have the server address.

We have created the new `tracked-hash-tag`. Let's check our `GET` API to obtain this data:

```
curl 'http://localhost:9090/api/tracked-hash-tag' | jq .'
```

Awesome, we called the `curl` tool and printed the JSON value with the `jq` tool. The command output should look like the following screenshot:

```
% Total    % Received % Xferd  Average Speed   Time     Time      Time  Current
          Dload Upload Total Spent   Left Speed
100  35    0   35    0     0  1415      0 --:--:-- --:--:-- --:--:-- 1458
[
  {
    "hashTag": "java",
    "queue": "java"
  }
]
```



To install `curl` on Ubuntu, we can use `sudo apt-get install curl -y`. Moreover, to install `jq`, we can use `sudo apt-get install jq -y`.

Summary

In this chapter, we have been introduced to the Kotlin language, which is the most prominent language for the JVM, because it has a super-fast compiler, if we compare it to Scala, for example. It also brings the simplicity of code and helps developers to create more concise and readable code.

We have also created our first application in the Spring Framework using Kotlin as the basic concepts of the language, and we saw how Kotlin helps the developers in a practical way.

We have introduced Redis as a cache and Spring Data Reactive Redis, which supports Redis in a Reactive paradigm.

In the last part of the chapter, we learned how to create a Twitter application which required us to create our next application, and start to consume the Twitter API in reactive programming with a Reactive Rest Client.

Let's jump to the next chapter and learn more about Spring Reactive.

5

Reactive Web Clients

Until now, we have created the whole project infrastructure to consume the Twitter stream. We have created an application which stores the tracked hashtags.

In this chapter, we will learn how to use the Spring Reactive Web Client and make HTTP calls using the reactive paradigm, which is one of the most anticipated features of Spring 5.0. We will call the Twitter REST APIs asynchronously and use the Project Reactor to provide an elegant way to work with streams.

We will be introduced to Spring Messaging for the RabbitMQ. We will interact with the RabbitMQ broker using the Spring Messaging API and see how Spring helps developers use the high-level abstractions for that.

At the end of this chapter, we will wrap up the application and create a docker image.

In this chapter, we will learn about:

- Reactive web clients
- Spring Messaging for RabbitMQ
- RabbitMQ Docker usage
- Spring Actuator

Creating the Twitter Gathering project

We learned how to create Spring Boot projects with the amazing Spring Initializr. In this chapter, we will create a project in a different way, to show you an alternative way of creating a Spring Boot project.

Create the `tweet-gathering` folder, in any directory. We can use the following command:

```
mkdir tweet-gathering
```

Then, we can access the folder created previously and copy the `pom.xml` file located at GitHub: <https://github.com/PacktPublishing/Spring-5.0-By-Example/blob/master/Chapter05/tweet-gathering/pom.xml>.

Open the `pom.xml` on IDE.

There are some interesting dependencies here. The `jackson-module-kotlin` helps to work with JSON in Kotlin language. Another interesting dependency is `kotlin-stdlib`, which provides the Kotlin standard libraries in our classpath.

In the plugin sections, the most important plugin is the `kotlin-maven-plugin`, which permits and configures the build for our Kotlin code.

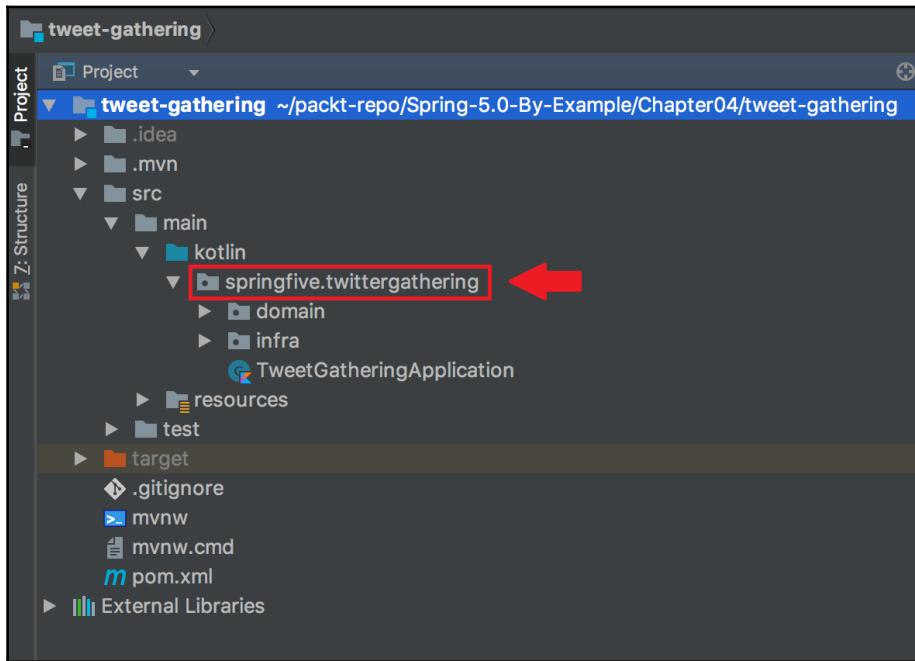
In the next section, we will create a folder structure to start the code.

Let's do it.

Project structure

The project structure follows the maven suggested pattern. We will code the project in the Kotlin language, then we will create a `kotlin` folder to store our code.

We made that configuration on the `pom.xml` created before, so it will work fine. Let's take a look at the correct folder structure for the project:



As we can see, the base package is the `springfive.twittergathering` package. Then, we will start to create sub-packages in this package as soon.

Let's create our infrastructure for the microservice.



The full source code can be found at GitHub: <https://github.com/PacktPublishing/Spring-5.0-By-Example/tree/master/Chapter05/tweet-gathering>.

Starting the RabbitMQ server with Docker

We can use Docker to spin up the RabbitMQ server. We do not want to install the server on our developer machines as it can create library conflicts and a lot of files. Let's understand how to start RabbitMQ in a Docker container.

Let's do that in the next couple of sections.

Pulling the RabbitMQ image from Docker Hub

We need to pull the RabbitMQ image from Docker Hub. We will use the image from the official repository as it is more safe and reliable.

To get the image, we need to use the following command:

```
docker pull rabbitmq:3.7.0-management-alpine
```

Wait for the download to end and then we can move forward to the next section. In the next section, we will learn how to set up the RabbitMQ server.

Starting the RabbitMQ server

To start the RabbitMQ server, we will run the Docker command. There are some considerations which we need to pay attention to; we will run this container on the Twitter Docker network created previously, but we will expose some ports on the host, as it makes it easier to interact with the broker.

Also, we will use the management image because it provides a page which enables us to manage and see the RabbitMQ information on something similar to a control panel.

Let's run:

```
docker run -d --name rabbitmq --net twitter -p 5672:5672 -p  
15672:15672 rabbitmq:3.7.0-management-alpine
```

Wait for a few seconds so that RabbitMQ establishes the connections and then we can connect to the management page. To do that, go to <http://localhost:15672> and log on to the system. The default user is **guest**, and the password is **guest** as well. The control panel looks like this:

The screenshot shows the RabbitMQ Management Console's Overview page. At the top, it displays the version as 3.7.0 Erlang 20.1.7. The main area has tabs for Overview, Connections, Channels, Exchanges, Queues, and Admin. The Overview tab is selected. It shows various metrics: Queued messages (last minute), Currently idle, Message rates (last minute), Currently idle, and Global counts. Below these are buttons for Connections: 0, Channels: 0, Exchanges: 8, Queues: 0, and Consumers: 0. A table titled 'Nodes' lists one node: rabbit@3eb5c73508d8, showing values for File descriptors (25), Socket descriptors (0), Erlang processes (364), Memory (79MB), Disk space (54GB), Uptime (18m 6s), and Info (basic, disc 1, rss). Buttons for 'This node' and 'All nodes' are present. At the bottom of the page are links for HTTP API, Server Docs, Tutorials, Community Support, Community Slack, Commercial Support, Plugins, GitHub, and Changelog.

There is a lot of interesting information on the panel, but for now, we are going to explore the channels and some interesting parts.

Awesome. Our RabbitMQ server is up and running. We will use the infrastructure soon.

Spring Messaging AMQP

This project supports the AMQP-based messaging solutions. There is a high-level API to interact with desired brokers. These interactions can send and receive messages from a broker.

Like in the other Spring projects, these facilities are provided by the *template* classes, which expose the core features provided by the broker and implemented by the Spring Module.

This project has two parts: `spring-amqp` is the base abstraction, and `spring-rabbit` is the RabbitMQ implementation for RabbitMQ. We will use `spring-rabbit` because we are using the RabbitMQ broker.

Adding Spring AMQP in our pom.xml

Let's add the `spring-amqp` jars to our project. `spring-amqp` has a starter dependency which configures some common things for us, such as `ConnectionFactory` and `RabbitTemplate`, so we will use that. To add this dependency, we will configure our `pom.xml` follows:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
```

The next step is to configure the connections; we will use the `application.yaml` file because we are using the starter. In the next section, we will do the configuration.

Integrating Spring Application and RabbitMQ

We have configured the `spring-amqp` dependencies in our project. Now, it is time to configure the RabbitMQ connections properly. We will use the `RabbitMQTemplate` to send messages to the broker; this has some converters which help us convert our domain models into JSON and vice versa.

Let's configure our RabbitMQ connections. The configurations should be in the `application.yaml` file and should look like this:

```
spring:
  rabbitmq:
    host: localhost
    username: guest
    password: guest
    port: 5672
```

As we can see, some Spring configurations are quite similar to others, the same style, and the node in `yaml` is the name of the technology followed by a couple of attributes.

We are using the default credentials for the RabbitMQ. The host and port are related to the RabbitMQ Broker address. The configuration is quite simple but does a lot of things for us such as `ConnectionFactory`.

Understanding RabbitMQ exchanges, queues, and bindings

We are doing some interesting things with RabbitMQ. We configured connections successfully. There are some other things that we have not done yet, such as configuring the exchanges, queue, and bindings, but before we do that, let's understand a little bit more about these terms.

Exchanges

Exchanges are RabbitMQ entities where the messages are sent. We can make an analogy with a river where the water is flowing; the river is the course of the messages. There are four different kinds of exchanges which we will understand in the following sections.

Direct exchanges

The direct exchanges allow for route messages based on the routing key. The name is self-explanatory, it permits to send the messages directly to the specified customer, who is the one listening to the exchange. Remember, it uses the routing key as the argument to route the message to the customers.

Fanout exchanges

The fanout exchanges route the messages for all the queues bound independently of the routing key. All the bound queues will receive the message sent to fanout exchanges. They can be used to have the topic behavior or distributed listings.

Topic exchanges

The topic exchanges are similar to direct exchanges, but topic exchanges enable us to use pattern matching as compared to the direct exchanges, which permit only the exact routing key. We will use this exchange in our project.

Header exchanges

Header exchanges are self-explanatory, the behavior is like the topic exchange, but instead of using the routing key, it uses the header attributes to match the correct queue.

Queues

Queues are the buffer where the exchanges will write the messages respecting the routing key. Queues are the place where consumers get the messages which are published to exchanges. Messages are routed to queues depending on the exchange type.

Bindings

Binding can be thought of as a link between exchanges and queues. We can say that it is a kind of traffic cop which instructs the messages where they should be redirected based on the configuration, in this case, links.

Configuring exchanges, queues, and bindings on Spring AMQP

The Spring AMQP project has abstractions for all the RabbitMQ entities listed previously, and we need to configure it to interact with the broker. As we did in other projects, we need a `@Configuration` class, which will declare the beans for the Spring container.

Declaring exchanges, queues, and bindings in yaml

We need to configure the entity names to instruct the framework to connect with the broker entities. We will use the `application.yaml` file to store these names, since it is easier to maintain and is the correct way to store application infrastructure data.

The section with the entity names should look like this snippet:

```
queue:  
    twitter: twitter-stream  
exchange:  
    twitter: twitter-exchange  
routing_key:  
    track: track.*
```

The properties are self-explanatory, the `exchange` node has the name of the exchange, the `queue` node has the queue name, and finally, the `routing_key` node has the routing argument.

Awesome. The properties are configured, and now we will create our `@Configuration` class. Let's do that in the next section. We are almost ready to interact with the RabbitMQ broker.

Declaring Spring beans for RabbitMQ

Now, let's create our configuration class. The class is pretty simple and as we will see with the Spring abstraction, they are easy to understand too, especially because the class names allude to the RabbitMQ entities.

Let's create our class:

```
package springfive.twittergathering.infra.rabbitmq

import com.fasterxml.jackson.databind.ObjectMapper
import com.fasterxml.jackson.module.kotlin.KotlinModule
import org.springframework.amqp.core.Binding
import org.springframework.amqp.core.BindingBuilder
import org.springframework.amqp.core.Queue
import org.springframework.amqp.core.TopicExchange
import
org.springframework.amqp.support.converter.Jackson2JsonMessageConverte
r
import org.springframework.beans.factory.annotation.Value
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration

@Configuration
open class RabbitMQConfiguration(@Value("\${queue.twitter}") private
val queue:String,
                                  @Value("\${exchange.twitter}")
private val
exchange:String,
                                  @Value("\${routing_key.track}")
private val routingKey:String){

    @Bean
    open fun queue():Queue{
        return Queue(this.queue, false)
    }

    @Bean
    open fun exchange():TopicExchange{
        return TopicExchange(this.exchange)
    }
}
```

```
    @Bean
    open fun binding(queue: Queue, exchange: TopicExchange): Binding {
        return
    BindingBuilder.bind(queue).to(exchange).with(this.routingKey)
    }

    @Bean
    open fun converter(): Jackson2JsonMessageConverter {
        return
    Jackson2JsonMessageConverter(ObjectMapper().registerModule(KotlinModul
e()))
    }

}
```

There are interesting things to pay attention to here. In the `RabbitMQConfiguration` constructor, we injected the values configured in the `application.yaml` file to name the entities. After that, we started to configure the Spring beans for the container to allow it to inject them into the Spring-managed classes. The key point here is that if they do not exist in the RabbitMQ broker, Spring will create them. Thanks, Spring, we appreciate that and love how helpful that is.

We can see the DSL to declare `Binding`, it makes the developer's life easier and prevents errors in the code.

On the last part of the class, we declared the `Jackson2JsonMessageConverter`. These converters are used to convert the domain models in JSON and vice versa. It enables us to receive the domain object on Listener instead of an array of bytes or strings. The same behavior can be used in the `Producers`, we are able to send the domain object instead of JSON.

We need to supply the `ObjectMapper` to `Jackson2JsonMessageConverter`, and we have used the Kotlin module because of the way Kotlin handles data classes, which do not have no-args constructors.

Excellent job! Our infrastructure is fully configured. Let's code the producers and consumers right now!

Consuming messages with Spring Messaging

Spring AMQP provides the `@RabbitListener` annotation; it will configure the subscriber for the desired queue, it removes a lot of infrastructure code, such as connect to `RabbitListenerConnectionFactory`, and creates a consumer programmatically. It makes the creation of queue consumers really easy.

The `spring-boot-starter-amqp` provides some automatic configurations for us. When we use this module, Spring will automatically create a `RabbitListenerConnectionFactory` for us and configure the Spring converters to convert JSON to domain classes automatically.

Pretty simple. Spring AMQP really provides a super high-level abstraction for developers.

Let's see an example which will be used in our application soon:

```
@RabbitListener(queues = ["twitter-track-hashtag"])
fun receive(hashTag:TrackedHashTag) {
    ...
}
```



The full source code can be found at GitHub: <https://github.com/PacktPublishing/Spring-5.0-By-Example/blob/master/Chapter05/tweet-gathering/src/main/kotlin/springfive/twittergathering/domain/service/TwitterGatherRunner.kt>.

A piece of cake. The code is really easy to understand and it makes it possible to pay attention only to the business rules. The infrastructure is not a good thing to maintain because this does not bring real value to the business, as it is only a piece of technology. Spring tries to abstract the whole infrastructure code to help developers write business code. It is a real asset provided by the Spring Framework.

Thanks, Spring Team.

Producing messages with Spring Messaging

The `spring-amqp` module provides a `RabbitTemplate` class, which abstracts high-level RabbitMQ driver classes. It improves the developer performance and makes the application void of bugs because the Spring modules are a very well-tested set of codes. We will use the `convertAndSend()` function which permits to pass exchange, the routing key, and the message object as parameters. Remember this function uses Spring converters to convert our model class into a JSON string.

There are a lot of overloaded functions for `convertAndSend()`, and depending on the use case, others could be more appropriate. We will use the simple one as we saw before.

Let's see the piece of code which sends the message to the broker:

```
this.rabbitTemplate.convertAndSend("twitter-exchange", "track.${hashTag.queue}", it)
```

Good. The first parameter is the Exchange name, and the second is the RoutingKey. Finally, we have the message object, which will be converted into a JSON string.

We will see the code in action soon.

Enabling Twitter in our application

In this section, we will enable the use of Twitter APIs on our Twitter Gathering application. This application should get Tweets based on the query specified by the user. This query was registered on the previous microservice that we created in the previous chapter.

When the user calls the API to register `TrackedHashTag`, the microservice will store the `TrackedHashTag` on the Redis database and send the message through the RabbitMQ. Then, this project will start to gather Tweets based on that. This is the data flow. In the next chapter, we will do a reactive stream and dispatch Tweets through our Reactive API. It will be amazing.

However, for now, we need to configure the Twitter credentials; we will do that using Spring beans – let's implement it.

Producing Twitter credentials

We will use the `@Configuration` class to provide our Twitter configuration objects. The `@Configuration` class is really good to provide infrastructure beans, if we do not have starter projects for the required module.

Also, we will use the `application.yaml` file to store the Twitter credentials. This kind of configuration should not be kept in the source code repository because it is sensitive data and should not be shared with others. Then, the Spring Framework enables us to declare properties in the `yaml` file and configures the environment variables to fill these properties at runtime. It is an excellent way to keep sensitive data out of the source code repository.

Configuring Twitter credentials in `application.yaml`

To start configuring the Twitter API in our application, we must provide the credentials. We will use the `yaml` file for this. Let's add credentials in our `application.yaml`:

```
twitter:  
    consumer-key: ${consumer-key}  
    consumer-secret: ${consumer-secret}  
    access-token: ${access-token}  
    access-token-secret: ${access-token-secret}
```

Easy peasy. The properties have been declared and then we used the `$` to instruct the Spring Framework that this value will be received as an environment variable. Remember, we configured the Twitter account in the previous chapter.

Modelling objects to represent Twitter settings

We must create abstractions and an amazing data model for our applications. This will create some models which make the developer's life easier to understand and code. Let's create our Twitter settings models.

Twittertoken

This class represents the application token previously configured in Twitter. The token can be used for the application authentication only. Our model should look like this:

```
data class TwitterToken(val accessToken: String, val  
accessTokenSecret: String)
```

I love the Kotlin way to declare data classes—totally immutable and without boilerplate.

TwitterAppSettings

TwitterAppSettings represents the consumer key and consumer secret. It is a kind of identity for our application, from Twitter's perspective. Our model is pretty simple and must look like this:

```
data class TwitterAppSettings(val consumerKey: String, val  
consumerSecret: String)
```

Good job, our models are ready. It is time to produce the objects for the Spring Container. We will do that in the next section.

Declaring Twitter credentials for the Spring container

Let's produce our Twitter configuration objects. As a pattern we have been using, we will use the `@Configuration` class for that. The class should be as follows:

```
package springfive.twittergathering.infra.twitter  
  
import org.springframework.beans.factory.annotation.Value  
import org.springframework.context.annotation.Bean  
import org.springframework.context.annotation.Configuration  
  
@Configuration  
open class TwitterConfiguration(@Value("\${twitter.consumer-key}")  
private val consumerKey: String,  
                                @Value("\${twitter.consumer-secret}")  
private val consumerSecret: String,  
                                @Value("\${twitter.access-token}")  
private val accessToken: String,  
                                @Value("\${twitter.access-token-")
```

```
secret}") private val accessTokenSecret: String) {  
  
    @Bean  
    open fun twitterAppSettings(): TwitterAppSettings {  
        return TwitterAppSettings(consumerKey, consumerSecret)  
    }  
  
    @Bean  
    open fun twitterToken(): TwitterToken {  
        return TwitterToken(accessToken, accessTokenSecret)  
    }  
  
}
```

Pretty simple and a Spring way to declare beans. We are improving how we use Spring step by step. Well done!

Now, we are done with Twitter configurations. We will consume the Twitter API using the WebClient from the Spring WebFlux, which supports the reactive programming paradigm. Let's understand something before we run the code.

Spring reactive web clients

This is a pretty new feature which was added in Spring Framework 5. It enables us to interact with HTTP services, using the reactive paradigm.

It is not a replacement for a `RestTemplate` provided by Spring, however, it is an addition to working with reactive applications. Do not worry, the `RestTemplate` is an excellent and tested implementation for interaction with HTTP services in traditional applications.

Also, the `WebClient` implementation supports the `text/event-stream` mime type which can enable us to consume server events.

Producing WebClient in a Spring Way

Before we start to call the Twitter APIs, we want to create an instance of `WebClient` in a Spring way. It means we are looking for a way to inject the instance, using the Dependency Injection Pattern.

To achieve this, we can use the `@Configuration` annotation and create a `WebClient` instance, using the `@Bean` annotation to declare the bean for the Spring container. Let's do that:

```
package springfive.twittergathering.infra.web

import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.web.reactive.function.client.WebClient

@Configuration
open class WebClientProducer {

    @Bean
    open fun webClient(): WebClient? {
        return WebClient.create()
    }

}
```

There are a couple of known annotations in this class; this is a pretty standard way to declare bean instances in a Spring way. It makes it possible to inject an instance of `WebClient` in other Spring-managed classes.

Creating the models to gather Tweets

If we want to consume the Twitter APIs asynchronously and reactively, then we should create the API client. Before we code the client, we need to create our classes for modeling, according to our requirements.

We do not need all Tweets' attributes. We expect the following attributes:

- `id`
- `text`
- `createdAt`
- `user`

Then, we will model our class based on the attributes listed.

Let's start with the user attribute. This attribute is a JSON attribute, and we will create a separated class for that. The class should look like this:

```
@JsonIgnoreProperties(ignoreUnknown = true)
data class TwitterUser(val id:String, val name:String)
```

We have used the Kotlin `data class`, it fits our use case well, and we want to use that as a data container. Also, we need to put `@JsonIgnoreProperties(ignoreUnknown = true)` because this annotation instructs the Spring converters to ignore the attribute when it is missing in the JSON response. That is the important part of this portion of code.

We have created the `TwitterUser` class, which represents the user who created the Tweet. Now, we will create the `Tweet` class which represents the Tweet. Let's create our class:

```
@JsonIgnoreProperties(ignoreUnknown = true)
data class Tweet(val id:String, val text:String,
    @JsonProperty("created_at") val createdAt:String, val
    user:TwitterUser)
```

There are some common things for us and one that's new. The `@JsonProperty` permits developers to customize the attribute name on the class which has a different attribute name in JSON; this is common for Java developers because they usually use *CamelCase* as a way to name attributes, and in JSON notation, people usually use *SnakeCase*. This annotation can help us to solve this mismatch between the programming language and JSON.



We can find a more detailed explanation of snake case here: https://en.wikipedia.org/wiki/Snake_case. Also, we can find a full explanation of camel case here: https://en.wikipedia.org/wiki/Camel_case.

Good. Our API objects are ready. With these objects, we are enabled to interact with the APIs. We will create a service to collect the Tweets. We will do that in the next section.

Authentication with Twitter APIs

With our objects ready, we need to create a class to help us handle the Twitter authentication. We will use the Twitter Application Only Auth authentication model. This kind of authentication should be used for backend applications.

The application using this kind of authentication can:

- Pull user timelines
- Access friends and followers of any account
- Access lists and resources
- Search in Tweets
- Retrieve any user information

As we can see, the application is a read-only Twitter API consumer.



We can use the Twitter documentation to understand this kind of authentication in detail. The documentation can be found here: <https://developer.twitter.com/en/docs/basics/authentication/guides/authorizing-a-request>.

We will follow the Twitter documentation to authorize our request, which is a kind of cooking recipe, so we must follow all the steps. The final class should look like this:

```
package springfive.twittergathering.infra.twitter

import org.springframework.util.StringUtils
import
springfive.twittergathering.infra.twitter.EncodeUtils.computeSignature
import springfive.twittergathering.infra.twitter.EncodeUtils.encode
import java.util.*

object Twitter {

    private val SIGNATURE_METHOD = "HMAC-SHA1"

    private val AUTHORIZATION_VERIFY_CREDENTIALS = "OAuth " +
        "oauth_consumer_key=\\"{key}\\"", " +
        "oauth_signature_method=\\"" + SIGNATURE_METHOD + "\\", " +
        "oauth_timestamp=\\"{ts}\\"", " +
        "oauth_nonce=\\"{nonce}\\"", " +
        "oauth_version=\\"1.0\\"", " +
        "oauth_signature=\\"{signature}\\"", " +
        "oauth_token=\\"{token}\\""
}
```

```
    fun buildAuthHeader(appSettings: TwitterAppSettings, twitterToken: TwitterToken, method: String, url: String, query: String):String{
        val ts = "" + Date().time / 1000
        val nounce = UUID.randomUUID().toString().replace("-".toRegex(), "")
        val parameters =
            "oauth_consumer_key=${appSettings.consumerKey}&oauth_nonce=$nounce&oauth_signature_method=$SIGNATURE_METHOD&oauth_timestamp=$ts&oauth_token=${encode(twitterToken.accessToken)}&oauth_version=1.0&track=${encode(query)}"
        val signature = "$method&" + encode(url) + "&" +
        encode(parameters)
        var result = AUTHORIZATION_VERIFY_CREDENTIALS
        result = StringUtils.replace(result, "{nonce}", nounce)
        result = StringUtils.replace(result, "{ts}", "" + ts)
        result = StringUtils.replace(result, "{key}",
        appSettings.consumerKey)
        result = StringUtils.replace(result, "{signature}",
        encode(computeSignature(signature,
        "${appSettings.consumerSecret}&${encode(twitterToken.accessTokenSecret)}")))
        result = StringUtils.replace(result, "{token}",
        encode(twitterToken.accessToken))
        return result
    }

    data class TwitterToken(val accessToken: String, val accessTokenSecret: String)

    data class TwitterAppSettings(val consumerKey: String, val consumerSecret: String)
```

It is a recipe. The function, `buildAuthHeader`, will create the authorization header using the rules to authorize the request. We have signed some request headers combined with a request body. Moreover, replace the template values with our Twitter credentials objects.

Some words about server-sent events (SSE)

Server-sent events (SSE) is a technology where the server sends events to the client, instead of the client polling the server to check the information availability. The message flow will not get interrupted until the client or server closes the stream.

The most important thing to understand here is the direction of the information flow. The server decides when to send data to a client.

It is very important to handle resource load and bandwidth usage. The client will receive the chunk of data instead to apply load on the server through the polling techniques.

Twitter has a stream API and the Spring Framework WebClient supports SSE. It is time to consume the Twitter stream.

Creating the gather service

The TweetGatherService will be responsible for interacting with Twitter APIs and collecting the request tweets according to the requested hashtag. The service will be a Spring bean with some inject attributes. The class should look like this:

```
package springfive.twittergathering.domain.service

import com.fasterxml.jackson.annotation.JsonIgnoreProperties
import com.fasterxml.jackson.annotation.JsonProperty
import org.springframework.http.MediaType
import org.springframework.stereotype.Service
import org.springframework.web.reactive.function.BodyInserters
import org.springframework.web.reactive.function.client.WebClient
import reactor.core.publisher.Flux
import springfive.twittergathering.infra.twitter.Twitter
import springfive.twittergathering.infra.twitter.TwitterAppSettings
import springfive.twittergathering.infra.twitter.TwitterToken

@Service
class TweetGatherService(private val twitterAppSettings:
TwitterAppSettings,
                        private val twitterToken: TwitterToken,
                        private val webClient: WebClient) {

    fun streamFrom(query: String): Flux<Tweet> {
        val url =
"https://stream.twitter.com/1.1/statuses/filter.json"
        return this.webClient.mutate().baseUrl(url).build()
            .post()
            .body(BodyInserters.fromFormData("track", query))
            .header("Authorization",
Twitter.buildAuthHeader(twitterAppSettings, twitterToken, "POST", url,
query))
```

```
        .accept(MediaType.TEXT_EVENT_STREAM)
        .retrieve().bodyToFlux(Tweet::class.java)
    }

}

@JsonIgnoreProperties(ignoreUnknown = true)
data class Tweet(val id: String = "", val text: String = "",
@JsonProperty("created_at") val createdAt: String = "", val user:
TwitterUser = TwitterUser("", ""))

@JsonIgnoreProperties(ignoreUnknown = true)
data class TwitterUser(val id: String, val name: String)
```

There are some important points here. The first is the function declaration; take a look at `Flux<Tweet>`, it means the data can never get interrupted because it represents the N values. In our case, we will consume the Twitter stream until the client or server interrupts the data flow.

After that, we configured the HTTP request body with our desired track to get events. After that, we configured the Accept HTTP header; it is essential to instruct the WebClient what kind of mime type it needs to consume.

Finally, we have used our `Twitter.buildAuthHeader` function to configure the Twitter authentication.

Awesome, we are ready to start to consume the Twitter API, and we only need to code the trigger to use that function. We will do that in the next section.

Listening to the Rabbit Queue and consuming the Twitter API

We will consume the Twitter API, but when?

We need to start to get Tweets when the request for tracking the hashtags comes to our application. To reach that goal, we will implement the RabbitMQ Listener when the `TrackedHashTag` gets registered on our microservice. The application will send the message to the broker to start consuming the Twitter stream.

Let's take a look at the code and step by step understand the behaviors; the final code should look like this:

```
package springfive.twittergathering.domain.service

import org.springframework.amqp.rabbit.annotation.RabbitListener
import org.springframework.amqp.rabbit.core.RabbitTemplate
import org.springframework.stereotype.Service
import reactor.core.publisher.Mono
import reactor.core.scheduler.Schedulers
import springfive.twittergathering.domain.TrackedHashTag
import java.util.concurrent.CompletableFuture
import java.util.concurrent.TimeUnit

@Service
class TwitterGatherRunner(private val twitterGatherService:
TweetGatherService, private val rabbitTemplate: RabbitTemplate) {

    @RabbitListener(queues = ["twitter-track-hashtag"])
    fun receive(hashTag:TrackedHashTag) {
        val streamFrom =
this.twitterGatherService.streamFrom(hashTag.hashTag).filter({
            return@filter it.id.isNotEmpty() && it.text.isNotEmpty()
        &&
            it.createdAt.isNotEmpty()
        })
        val subscribe = streamFrom.subscribe({
            println(it.text)
            Mono.fromFuture(CompletableFuture.runAsync {
                this.rabbitTemplate.convertAndSend("twitter-
                    exchange", "track.${hashTag.queue}", it)
            })
        })
        Schedulers.elastic().schedule({ subscribe.dispose()
}, 10L, TimeUnit.SECONDS)
    }
}
```

Keep calm. We will cover the whole code. In the `@RabbitListener`, we configured the name of the queue we want to consume. The Spring AMQP module will configure our listener automatically for us and start to consume the desired queue. As we can see, we received the `TrackedHashTag` object; remember the converters on the previous sections.

The first instruction will start to consume the Twitter stream. The stream returns a flux and can have a lot of data events there. After the consumer, we want to filter the data on the flow. We want Tweet in which the `id`, `text`, and `createdAt` are not null.

Then, we subscribe this stream and start to receive the data in the flow. Also, the `subscribes` function returns the disposable object which will be helpful in the next steps. We have created an anonymous function which will print the `Tweet` on the console and send the Tweet to the RabbitMQ queue, to be consumed in another microservice.

Finally, we use the schedulers to stop the data flow and consume the data for 10 seconds.

Before you test the Twitter stream, we need to change the Tracked Hashtag Service to send the messages through the RabbitMQ. We will do that in the next sections. The changes are small ones and we will do them quickly.

Changing the Tracked Hashtag Service

To run the whole solution, we need to make some changes to the Tracked Hashtag Service project. The changes are simple and basic; configure the RabbitMQ connection and change the service to send the messages to the broker.

Let's do that.

Adding the Spring Starter RabbitMQ dependency

As we did before in the Twitter Gathering project, we need to add `spring-boot-starter-amqp` to provide some auto-configuration for us. To do that, we need to add the following snippet to our `pom.xml`:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
```

Right. Now, it is time to configure the RabbitMQ connections. We will do this in the next section.

Configuring the RabbitMQ connections

We will use the application.yaml to configure the RabbitMQ connections. Then, we need to create a couple of properties in it and the Spring AMQP module will use that provided configuration to start the connection factory.

It is pretty simple to configure it. The final yaml file for Tracked Hashtag should look like this:

```
spring:
  rabbitmq:
    host: localhost
    username: guest
    password: guest
    port: 5672
  redis:
    host: 127.0.0.1
    port: 6379

  server:
    port: 9090

  queue:
    twitter: twitter-track-hashtag
  exchange:
    twitter: twitter-track-exchange
  routing_key:
    track: "*"
---
spring:
  profiles: docker
  rabbitmq:
    host: rabbitmq
    username: guest
    password: guest
    port: 5672
  redis:
    host: redis
    port: 6379

  server:
    port: 9090

  queue:
    twitter: twitter-track-hashtag
  exchange:
    twitter: twitter-track-exchange
```

```
routing_key:  
  track: "*"
```

There are two profiles in this yaml. Take a look at the different host for the RabbitMQ. In the default profile, we are able to connect the localhost because we exposed the RabbitMQ ports on the host. But on the Docker profile, we are not able to connect the localhost, we need to connect to the `rabbitmq` host, which is the host for the Twitter network.

Our RabbitMQ connection is ready to use. Let's try it in the next section. Let's go.

Creating exchanges, queues, and bindings for the Twitter Hashtag Service

Let's declare our RabbitMQ entities for the Tracked Hashtag usage. We will do that using the `@Configuration` class.

The RabbitMQ connection should look like this:

```
package springfive.twittertracked.infra.rabbitmq

import com.fasterxml.jackson.databind.ObjectMapper
import com.fasterxml.jackson.module.kotlin.KotlinModule
import org.springframework.amqp.core.Binding
import org.springframework.amqp.core.BindingBuilder
import org.springframework.amqp.core.Queue
import org.springframework.amqp.core.TopicExchange
import
org.springframework.amqp.support.converter.Jackson2JsonMessageConverte
r
import org.springframework.beans.factory.annotation.Value
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration

@Configuration
open class RabbitMQConfiguration(@Value("\${queue.twitter}") private
val queue:String,
                                  @Value("\${exchange.twitter}")
private val exchange:String,
                                  @Value("\${routing_key.track}")
private val routingKey:String) {

    @Bean
    open fun queue():Queue{
        return Queue(this.queue, false)
```

```
}

@Bean
open fun exchange(): TopicExchange{
    return TopicExchange(this.exchange)
}

@Bean
open fun binding(queue: Queue, exchange: TopicExchange): Binding {
    return
BindingBuilder.bind(queue).to(exchange).with(this.routingKey)
}

@Bean
open fun converter(): Jackson2JsonMessageConverter {
    return
Jackson2JsonMessageConverter(ObjectMapper().registerModule(KotlinModul
e()))
}

}
```

Pretty straightforward. We declared one exchange, queue, and binding, as we did before.

Sending the messages to the broker

This is the most interesting part now. When we want to save the `TrackedHashTag`, we must send the pretty new entity to the RabbitMQ. This process will send the message, and then the Twitter Gathering microservice will start to consume the stream in ten seconds.

We need to change the `TrackedHashTagService` a little bit; the final version should look like this:

```
package springfive.twittertracked.domain.service

import org.springframework.amqp.rabbit.core.RabbitTemplate
import org.springframework.beans.factory.annotation.Value
import org.springframework.stereotype.Service
import reactor.core.publisher.Mono
import springfive.twittertracked.domain.TrackedHashTag
import
springfive.twittertracked.domain.repository.TrackedHashTagRepository
import java.util.concurrent.CompletableFuture
```

```
@Service
class TrackedHashTagService(private val repository:
TrackedHashTagRepository,
                           private val rabbitTemplate:
RabbitTemplate,
                           @Value("\${exchange.twitter}") private val
exchange: String,
                           @Value("\${routing_key.track}") private
val routingKey: String) {

    fun save(hashTag: TrackedHashTag) {
        this.repository.save(hashTag).subscribe { data ->
            Mono.fromFuture(CompletableFuture.runAsync {
                this.rabbitTemplate.convertAndSend(this.exchange,
this.routingKey,
hashTag)
            })
        }
    }

    fun all() = this.repository.findAll()
}

}
```

Awesome job. When the new entity comes, it will be sent to the broker. We have finished our changes on the Tracked Hashtag Service.

Finally, we are able to test the whole flow. Let's start to play and perceive the real power of our built application.

It's showtime!!!

Testing the microservice's integrations

Now, we are ready to test the whole solution. Before you start, we need to check the following infrastructure items:

- Redis
- RabbitMQ

If the items are up and running, we can jump to the next section.



We can use the `docker ps` command, and the command should list the Redis and RabbitMQ containers in running mode.

Running Tracked Hashtag Service

There is no special thing to run this application. It includes the infrastructure connections which are configured in the default profile in `application.yaml`.

Run the main function present on the `TrackedHashTagApplication`. We can use the IDE or command line to do that.

Check the console output; the output will be presented on the IDE or command line. We want to find the following line:

```
[main] o.s.c.support.DefaultLifecycleProcessor : Starting beans in phase 2147483647
[ctor-http-nio-1] r.ipc.netty.tcp.BlockingNettyContext : Started HttpServer on /0:0:0:0:0:0:0:9090
[main] o.s.b.web.embedded.netty.NettyWebServer : Netty started on port(s): 9090
[main] s.t.TrackedHashTagApplication$Companion : Started TrackedHashTagApplication.Companion in 4.179 seconds (JVM running for 5.059)
```

It means the first application is fully operational and we are able to run Twitter Gathering. Please keep the application running as it is required.

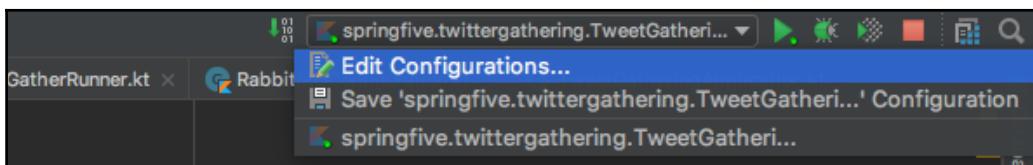
Let's run Twitter Gathering!!!

Running the Twitter Gathering

This application is a little bit more complicated to run. We need to configure some environment variables for that. It is required because we do not want the Twitter application credentials in our repository.

It is pretty simple to do in the IDE. To do that, we can configure the run configuration. Let's do it:

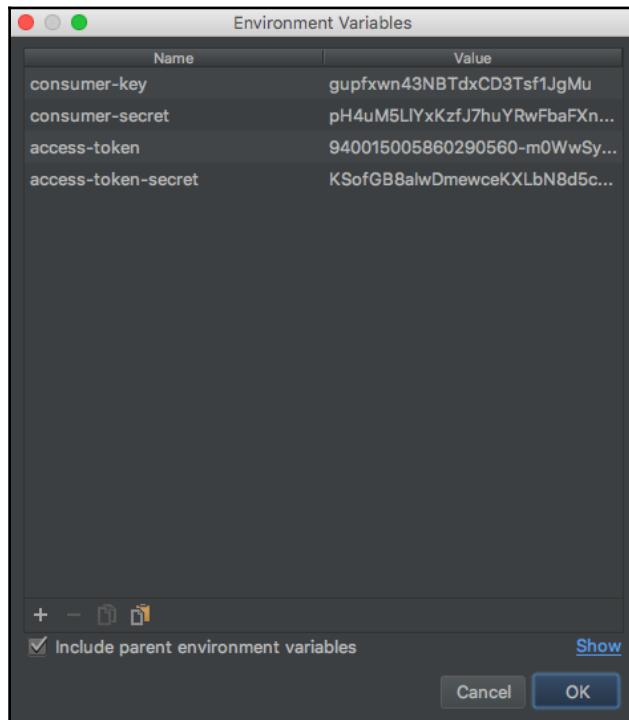
1. Click on the **Edit Configurations...** like in the following image:



Then, we are able to see the **Environment variables** like this:



2. We need to click on ..., as highlighted in the proceeding image.
3. The next screen will be shown and we can configure the **Environment Variable**:



4. We need to configure the following environment variables:

- **consumer-key**
- **consumer-secret**
- **access-token**
- **access-token-secret**

These values should be filled with the **Twitter Application Management** values.

Then, we can run the application. Run it!!

Now, we should see the following lines in the console, which means the application is running:

```
2017-12-23 15:58:46.362 INFO 1635 --- [ctor-http-nio-1] r.i.p.c.netty.tcp.BlockingNettyContext : Started HttpServer on /0:0:0:0:0:0:0:8081  
2017-12-23 15:58:46.363 INFO 1635 --- [           main] o.s.d.web.embedded.netty.NettyWebServer : Netty started on port(s): 8081  
2017-12-23 15:58:46.369 INFO 1635 --- [           main] s.t.TweetGatheringApplication$Companion : Started TweetGatheringApplication.Companion in 7.012 seconds (JVM running for 7.645)
```

Awesome, our two microservices are running. Let's trigger the Twitter stream. We will do that in the next section.



There are other ways to run the application, for example, with the maven Spring Boot goals or Java command line. If you prefer to run in the Java command line, keep in mind the `-D` argument to pass environment variables.

Testing stuff

We are excited to test the full integration. We can use the `curl` tool to send request data to the Tracked Hashtag Service. We want to track the "bitcoin" from Twitter.

We can execute the following command line:

```
curl -H "Content-Type: application/json" -X POST -d  
'{"hashTag":"bitcoin", "queue":"bitcoin"}' \  
http://localhost:9090/api/tracked-hash-tag
```

Check the HTTP status code; it should be HTTP status 200. After that, we can check the console from the Twitter Gathering project, and there should be a lot of Tweets logged.

Take a look at the log, the log must have Tweets like this:

```
RT @PayperExnet: Bitcoin is going down? GOOD! buy PAX and join the last days of the sale! buy PAX TOKEN NOW!!! @PayperExnet @Bitcoin https://t.co/...  
RT @AttWorldNews: Bitcoin Goes on Wild Journey and it Could Solely Get Crazier #Bitcoin #Blockchain #LengthyIslandIceTeaCorp https://t.co/...  
@BBrianKelly #Listen to #Bitcoin #audio #voxpop #BitcoinMadness #LetMeAsk @AmaroufmediaAsk #YouTube https://t.co/InqCHH0wDw  
The blockchain that wouldn't die #crypto https://t.co/oXFMg0efl  
RT @rajneeshchhabra: No Place Like Home: The Internet Of Things And Its Promise For Consumers https://t.co/vAwghGoEca #IoT #InternetOfThing...  
RT @ltsHoover: *gets one dollar profit with bitcoin* https://t.co/PB6DyP7hyd  
3 Cryptocurrencies to Consider Buying Over Bitcoin #themotleyfool #stocks $SAN, $IBM, $AXP, $BP https://t.co/gfIAVE87wo.  
So in order to buy 1 bit you'll need to pay 3500 bits in fees. That sounds like fun! That's what the Core developer.. https://t.co/un50HfvNec  
RT @LifeInvestasset: #cryptocurrencyLifeInvest (Change 24h):  
-#Bitcoin $15.584,80 (-7,82%).  
-#BitcoinCash $3.212,84 (-16,70%).  
-#Ether $809..  
RT @HealthRanger: What do you think: Do #China and #Russia really have this much control over #Bitcoin ? https://t.co/yjKPzQG0sW  
RT @ToshiDesk: Guys - Next #ICO to watch out for!  
  
https://t.co/Vfol0rWYce  
  
They already have 10+ courses up on their Educational Site..  
RT @AlexanderHaxton: One of my favorite ICOs right now: Trade.io is revolutionising the banking industry. Many #ICO's will be launched on T...  
RT @yahapErenTR: Dear @Poloniex !  
My friend's account @urselkaraslan has been frozen for 10 days. Please respond the #580589 ticket which..  
@krios_io Bitcoin Diamond (SBCD) successfully launch mainnet and 28 global exchanges start trades! If you have bala.. https://t.co/06uRaCVDY1  
Zum ständigen auf und ab der #Bitcoin ein sehr leserwarter Artikel von @SPIEGELONLINE – danke für das Interview... https://t.co/Y7zhRlbcB  
RT @JamesGRickards: OK, let's make this simple. Bitcoin is a multiplayer game dressed up as a real world experience. Enjoy! https://t.co/lx...  
RT @rajneeshchhabra: No Place Like Home: The Internet Of Things And Its Promise For Consumers https://t.co/vAwghGoEca #IoT #InternetOfThing...  
Bitcoin value tumbles by 30 per cent as investors face 'reality check' https://t.co/tdBkaTEZr/  
Uh what? 😱 I guess he forgot to show where the Nasdaq is today 😂😂😂 https://t.co/mChT4uMugM
```

Awesome!

Great work guys, we have the full application integrated with RabbitMQ and the Twitter stream.

Spring Actuator

The Spring Boot Actuator is a kind of helper when the application is running in production. The project provides built-in information of a deployed application.

In the microservices world, monitoring instances of applications are the key point to getting success. In these environments, there are usually many applications calling the other applications over the network protocols such as HTTP. The network is an unstable environment and sometimes it will fail; we need to track these incidents to make sure the application is up and fully operational.

The Spring Boot Actuator helps developers in these situations. The project exposes a couple of HTTP APIs with application information, such as the memory usage, CPU usage, application health check, and the infrastructure components of the application, such as a connection with databases and message brokers, as well.

One of the most important points is that the information is exposed over HTTP. It helps integrations with external monitor applications such as Nagios and Zabbix, for instance. There is no specific protocol for exposing this information.

Let's add it to our project and try a couple of endpoints.

Adding Spring Boot Actuator in our pom.xml

Spring Boot Actuator is pretty simple to configure in our `pom.xml`. We extended the parent pom of Spring Boot, so it is not necessary to specify the version of the dependency.

Let's configure our new dependency:

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
</dependencies>
```

Awesome, really easy. Let's understand a little bit more before we test.

Actuator Endpoints

The projects have a lot of built-in endpoints and they will be up when the application started. Remember, we have used the starter project, which is the one that configures it automatically for us.

There are several endpoints for different requirements, and we will take a look at the most used in production microservices.

- `/health`: The most known actuator endpoint; it shows the application's health, and usually, there is a `status` attribute
- `/configprops`: Displays a collapse `@ConfigurationProperties`
- `/env`: Exposes properties from the Spring `ConfigurableEnvironment`
- `/dump`: Shows the thread dump
- `/info`: We can put some arbitrary information at this endpoint
- `/metrics`: Metrics from the running application
- `/mappings`: `@RequestMapping` endpoints from the current application

There is another important endpoint to show the application logs over the HTTP interface. The `/logfile` endpoint can help us visualize logfiles.



The list of endpoints created by the Spring Boot Actuator can be found at: <https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-endpoints.html>.

Application custom information

There is one particular endpoint which we can use to expose custom information from our application. This information will be exposed to `/info` endpoint.

To configure that, we can use the `application.yaml` file and put the desired information respecting the pattern, as follows:

```
info:  
  project: "twitter-gathering"  
  kotlin: @kotlin.version@
```

The desired properties must be preceded by the `info.*`. Then, we can test our first actuator endpoint and check our `/info` resource.

Let's try to access the `http://localhost:8081/info`. The information filled on `application.yaml` should be displayed, as shown here:

A screenshot of a web browser window. The address bar shows the URL `localhost:8081/actuator/info`. The page content displays a JSON object:

```
{  
  "project": "twitter-gathering",  
  "kotlin": "1.2.0"  
}
```

As we can see, the properties are exposed from the HTTP endpoint. We can use that to put the application version, for instance.

Testing endpoints

In version 2 of Spring Boot, the Spring Actuator management endpoints are disabled by default, because these endpoints can have sensitive data of a running application. Then, we need to configure to enable these endpoints properly.

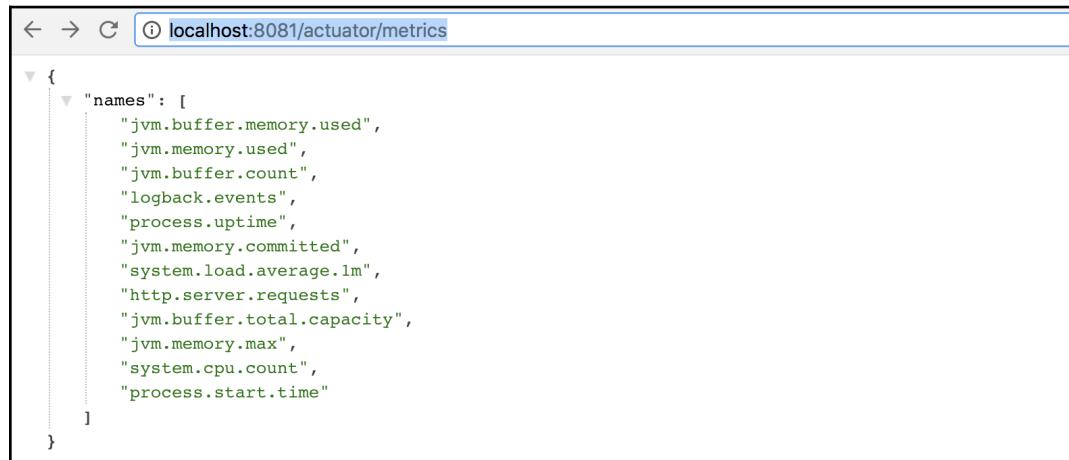
There is a special point to pay attention to. If the application is exposed publicly, you should protect these endpoints.

Let's enable our management endpoints:

```
management:  
  endpoints:  
    web:  
      expose: "*"
```

In the preceding configuration, we enabled all the management endpoints, and then we can start to test some endpoints.

Let's test some endpoints. First, we will test the metrics endpoints. This endpoint shows the metrics available for the running application. Go to <http://localhost:8081/actuator/metrics> and check the result:





We are using port 8081 because we configured the property `server.port` in `application.yaml`. The port can be changed as you desire.

There are a lot of metrics configured automatically for us. That endpoint exposes only the available metrics. To check the metric value, we need to use another endpoint. Let's check the value of the `http.server.requests`.

The base endpoint to check the value is: `http://localhost:8081/actuator/metrics/{metricName}`. Then, we need to go to: `http://localhost:8081/actuator/metrics/http.server.requests`. The result should be:

A screenshot of a web browser window displaying a JSON response. The URL in the address bar is `localhost:8081/actuator/metrics/http.server.requests`. The JSON data shows the following structure:

```
{
  "name": "http.server.requests",
  "measurements": [
    {
      "statistic": "Count",
      "value": 8
    },
    {
      "statistic": "TotalTime",
      "value": 281213374
    },
    {
      "statistic": "Max",
      "value": 281213374
    }
  ],
  "availableTags": [ ... ] // 4 items
}
```

As you can see, the server received eight calls. Try to hit a few more times to see the metrics changing.

Awesome job. Our microservice is ready for production. We have the docker image and endpoints for monitoring our services.

Summary

In this chapter, we learned and put into practice a lot of Spring Advanced concepts, such as RabbitMQ integration.

We have created a fully reactive WebClient and took advantage of the reactive paradigm; it enables resource computational optimization and increases performance for the application.

Also, we have integrated two microservices through the RabbitMQ broker. This is an excellent solution to integrating applications because it decouples the applications and also permits you to scale the application horizontally really easily. Message-driven is one of the required characteristics to build a reactive application; it can be found at Reactive Manifesto (<https://www.reactivemanifesto.org/en>).

In the next chapter, we will improve our solution and create a new microservice to stream the filtered Tweets for our clients. We will use RabbitMQ one more time.

6

Playing with Server-Sent Events

In Chapter 4, Kotlin Basics and Spring Data Redis and Chapter 5, *Reactive Web Clients*, we created two microservices. The first one is responsible for keeping tracked data on Redis and triggering the second microservice which one will consume the Twitter stream. This process happens asynchronously.

In this chapter, we will create another microservice which will consume the data produced by Twitter Gathering and expose it via a REST API. It will be possible to filter Tweets by text content.

We have consumed the Twitter stream using the **Server-Sent Events (SSE)**; we created a reactive REST client to consume that. Now, it is time to create our implementation for SSE. We will consume the RabbitMQ queue and push the data to our connected clients.

We will take a look at the SSE and understand why this solution fits well for our couple of microservices.

At the end of the chapter, we will be confident about using SSE in the Spring ecosystem.

In this chapter, we will learn the following:

- Implementation of SSE endpoints with the Spring Framework
- Consuming RabbitMQ using the Reactor Rabbit client

Creating the Tweet Dispatcher project

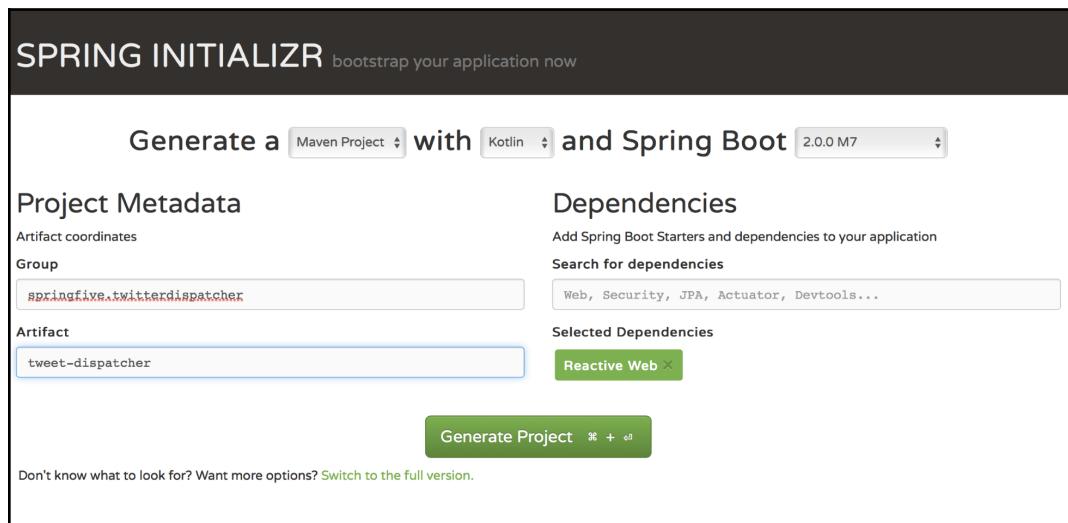
Now, we will create our last microservice. It will push the Tweets filtered by Twitter Gathering for our connected clients, in this case, consumers.

In this chapter, we will use the Spring Initializr page to help us create our pretty new project. Let's create.

Using Spring Initializr once again

As you can see, the Spring Initializr page is a kind of partner for creating Spring projects. Let's use it one more time and create a project:

Go to <https://start.spring.io> and fill in the data using the following screenshot:



We have selected the **Reactive Web** dependencies; we will also keep using Kotlin as a programming language. Finally, click on the **Generate Project** button. Good, it is enough for us.

There are some missing dependencies which are not displayed in the Spring Initializr. We need to set these dependencies manually. We will do that task in the next section. Let's go there.

Additional dependencies

We need to use the Jackson Kotlin Module as a dependency to handle JSON properly in our new microservice. Also, we will use the Reactor RabbitMQ dependency, which allows us to interact in the reactive paradigm with the RabbitMQ Broker.

To add these dependencies, we need to add the following snippet to `pom.xml`:

```
<dependency>
    <groupId>com.fasterxml.jackson.module</groupId>
    <artifactId>jackson-module-kotlin</artifactId>
    <version>${jackson.version}</version>
</dependency>

<dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-test</artifactId>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>io.projectreactor.rabbitmq</groupId>
    <artifactId>reactor-rabbitmq</artifactId>
    <version>1.0.0.M1</version>
</dependency>
```

Awesome. Our dependencies are configured. Our project is ready to start.

Before we start, we need to understand, in depth, the concept of SSE. We will learn this in the next section.

Server-Sent Events

Server-Sent Events (SSE) is a standard way to send data streams from a server to clients. In this next section, we will learn how to implement it using the Spring Framework.

Also, we will understand the main differences between SSE and WebSockets.

A few words about the HTTP protocol

HTTP is an application layer protocol in the OSI model. The application layer is the last layer represented in the OSI model. It means this layer is closer to the user interface. The main purpose of this layer is to send and receive the data input by the user. In general, it happens by the user interface, also known as applications, such as file transfer and sending an email.

There are several protocols on the application layer such as Domain Name Service (DNS), which translates the domain names to IP address, or SMTP, whose main purpose is to deliver an email to a mail manager application.

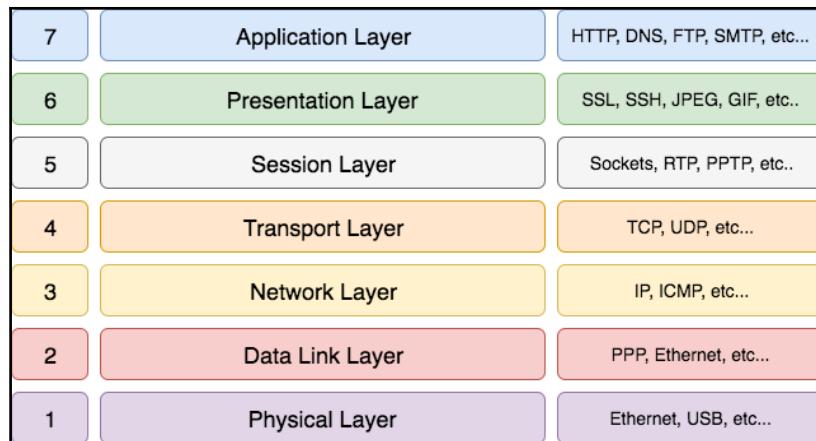
The application layer interacts directly with software such as email clients, for instance; there are no interactions with the hardware parts. It is the last layer of the OSI model and the closest to the end user as well.

All these layers deal with software, which means there are no concerns about the physical parts represented in the OSI model.



A more detailed explanation of the OSI model can be found
at: <https://support.microsoft.com/en-us/help/103884/the-osi-model-s-seven-layers-defined-and-functions-explained>.

The following is an OSI model representation:



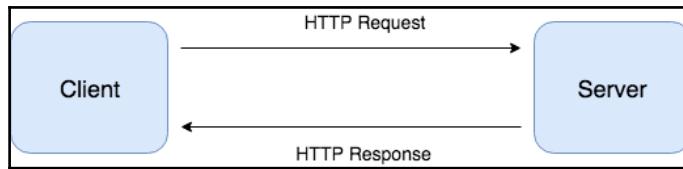
The HTTP protocol uses the TCP protocol as a transportation channel. Then, it will establish a connection and start to flow the data on the channel.

The TCP protocol is a stream protocol and a full duplex channel. This means the server and clients can send data across the connection.

HTTP and persistent connections

The HTTP protocol is a request-response model, where the client submits the message (HTTP Request) and the server processes this message and sends the response (HTTP Response) to the client. The connection will be closed after the response is sent.

Look at the following diagram:



It's pretty simple to understand. The client will send the request, and in this case, the connection will be opened. After that, the server will receive the request to process something and it will send the answer to the client. The connection will be closed after the whole process. If the client needs to send a new request, the connection should be opened again and the flow happens in the same order.

There is a perceived drawback here, the clients need to open the new connection per-request. From the server's eyes, the server needs to process a lot of new connections simultaneously. This consumes a lot of CPU and memory.

On HTTP's 1.0 version, the connections are not persistent. To enable it, the `keep-alive` header should be included on the request. The header should look like this:

Connection: keep-alive

This is the only way to make an HTTP connection persistent on the 1.0 version, as described previously; when it happens, the connection will not be dropped by the server and the client is able to reuse the opened connection.

On HTTP 1.1, the connections are persistent by default; in this case, as opposed to the first version, the connection is kept opened and the client can use it normally.

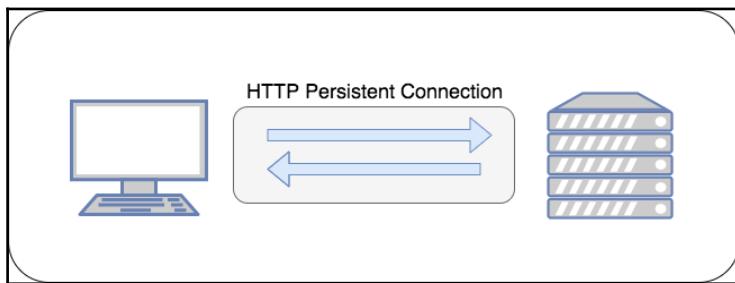
There is a perceived improvement here and it can bring some advantages. The server needs to manage fewer connections, and it reduces a lot of CPU time. The HTTP Requests and Responses can be pipelined in the same connection.

As we know, *there is no such thing as a free lunch*. There are some disadvantages to this as well; the server needs to keep the connection opened and the server will reserve the required connection for the client. This may cause server unavailability in some scenarios.

Persistent connections can be useful to maintain a stream between the server and clients.

WebSockets

In the HTTP protocol, the communication supports full-duplex, which means the client and server can send data through the channel. The standard way to support this kind of communication is WebSockets. In this specification, both client and server can send data to each other in the persistent connection. Look at the following diagram:



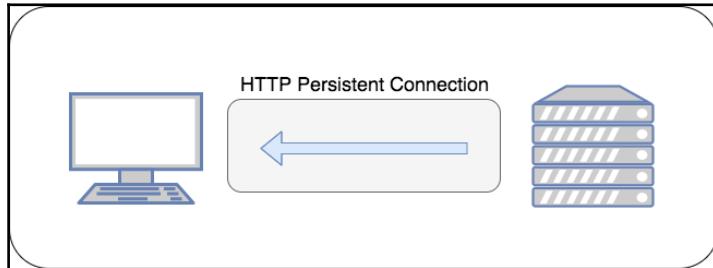
As we can see, the data can be sent and received by the two actors, client, and server—this is how WebSockets works.

In our case, we do not need to send any data to the server during the connection. Because of this characteristic, we will choose SSE. We will learn about them in the following section.

Server-Sent Events

As opposed to the full-duplex communication implemented by WebSockets, the SSE uses a half-duplex communication.

The client sends a request to the server, and when necessary, the server will push the data to the client. Remember the active actor here is the server; the data can be sent only by the server. This is a half-duplex behavior. Look at the following diagram:



A piece of cake. It is the base of the SSE technology. SSE is self-explanatory. We will use it with the Spring Framework. However, before we do that, let's look at a Reactor RabbitMQ project.

Reactor RabbitMQ

Our solution is fully reactive, so we need to use Reactor RabbitMQ, which allows us to interact with the RabbitMQ broker using the reactive paradigm.

On this new microservice, we do not need to send messages through the message broker. Our solution will listen to the RabbitMQ queues and push the received Tweets for the connected clients.

Understanding the Reactor RabbitMQ

The Reactor RabbitMQ tries to provide a reactive library to interact with the RabbitMQ broker. It enables developers to create non-blocking applications based on the reactive stream, using RabbitMQ as a message-broker solution.

As we learned before, this kind of solution, in general, does not use a lot of memory. The project was based on the RabbitMQ Java client and has similar functionalities, if we compare it to the blocking solution.

We are not using the `spring-amqp-starter`, so the magic will not happen. We will need to code the beans declarations for the Spring context and we will do that in the following section.

Configuring RabbitMQ Reactor beans

In this section, we will configure the RabbitMQ infrastructure classes in the Spring context. We will use a `@Configuration` class to declare it.

The configuration class should look like the following:

```
package springfive.twitterdispatcher.infra.rabbitmq

import com.fasterxml.jackson.databind.ObjectMapper
import com.fasterxml.jackson.module.kotlin.KotlinModule
import com.rabbitmq.client.ConnectionFactory
import org.springframework.beans.factory.annotation.Value
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import reactor.rabbitmq.ReactorRabbitMq
import reactor.rabbitmq.Receiver
import reactor.rabbitmq.ReceiverOptions

@Configuration
class RabbitMQConfiguration(private
    @Value("\${spring.rabbitmq.host}") val host:String,
    private
    @Value("\${spring.rabbitmq.port}") val port:Int,
    private
    @Value("\${spring.rabbitmq.username}") val username:String,
    private
    @Value("\${spring.rabbitmq.password}") val password:String){

    @Bean
    fun mapper(): ObjectMapper =
        ObjectMapper().registerModule(KotlinModule())

    @Bean
    fun connectionFactory():ConnectionFactory{
        val connectionFactory = ConnectionFactory()
        connectionFactory.username = this.username
        connectionFactory.password = this.password
        connectionFactory.host = this.host
        connectionFactory.port = this.port
        connectionFactory.useNio()
        return connectionFactory
    }

    @Bean
    fun receiver(connectionFactory: ConnectionFactory):Receiver{
        val options = ReceiverOptions()
        options.connectionFactory(connectionFactory)
```

```
    return ReactorRabbitMq.createReceiver(options)
}

}
```

There are two important things here. The first one is that we configured the Jackson support for Kotlin. It allows us to inject the `ObjectMapper` into our Spring beans. The next important thing is related to the RabbitMQ connections' configuration.

We have declared a `ConnectionFactory` bean for the Spring Context. We injected the configurations with `@Value` annotations and received the values on the constructor. We can set the value directly in the attributes, in the Kotlin language; look at the `ConnectionFactory` attributes assignments.

After the `ConnectionFactory` configuration, we are able to declare a receiver, which is a Reactive abstraction to consume the queues, using reactive programming. We receive the `ConnectionFactory` previously created and set it as the `ReceiverOptions`.

That is all for the Reactor RabbitMQ configuration.

Consuming the RabbitMQ queues reactively

Now, we will consume the RabbitMQ queues. The implementation is quite similar to what we have seen in the blocking implementation, and the names of the functions are similar as well.

We have consumed some RabbitMQ messages in the previous chapters, but this solution is quite different. Now, we will use the Reactive RabbitMQ implementation. The main idea here is to consume the stream of events; these events represent the messages that have arrived in the broker. These messages arrive and the Reactor RabbitMQ converts these messages to `Flux`, to enable us to consume in the reactive paradigm.

In the reactive paradigm, the representation of a stream of events (we can think of messages in the queue), is the `Flux`.

Then our function, which is listening to the RabbitMQ, should return `Flux`, an infinite representation of events. The Receiver implementation returns the `Flux` of messages, which is enough for us and fits well with our needs.

Our implementation should look like the following:

```
package springfive.twitterdispatcher.domain.service

import com.fasterxml.jackson.annotation.JsonIgnoreProperties
import com.fasterxml.jackson.annotation.JsonProperty
import com.fasterxml.jackson.databind.ObjectMapper
import com.fasterxml.jackson.module.kotlin.readValue
import org.springframework.beans.factory.annotation.Value
import org.springframework.stereotype.Service
import reactor.core.publisher.Flux
import reactor.core.publisher.Mono
import reactor.rabbitmq.Receiver

@Service
class TwitterDispatcher(private @Value("\${queue.twitter}") val queue: String,
    private val receiver: Receiver,
    private val mapper: ObjectMapper) {

    fun dispatch(): Flux<Tweet> {
        return this.receiver.consumeAutoAck(this.queue).flatMap { message ->
            Mono.just(mapper.readValue<Tweet>(String(message.body)))
        }
    }

    @JsonIgnoreProperties(ignoreUnknown = true)
    data class Tweet(val id: String = "",
        val text: String = "", @JsonProperty("created_at")
        val createdAt: String = "", val user: TwitterUser =
        TwitterUser("", ""))
}

@JsonIgnoreProperties(ignoreUnknown = true)
data class TwitterUser(val id: String, val name: String)
```

Let's understand a little bit more. We received the `Receiver` as an injection in our constructor. When someone invokes the `dispatch()` function, the `Receiver` will start to consume the queue, which was injected in the constructor as well.

The Receiver produces `Flux<Delivery>`. Now, we need to convert the instance of `Flux<Delivery>`, which represents a message abstraction, to our domain model `Tweet`. The `flatMap()` function can do it for us, but first, we will convert the `message.body` to string and then we have used Jackson to read JSON and convert to our `Tweet` domain model.

Take a look at how simple the code is to read; the API is fluent and really readable.

The consumer will not terminate until the connected client disconnects. We will be able to see this behavior soon.

Filtering streams

We are receiving the messages from RabbitMQ. Now, we need to return the messages to the connected customer.

For that, we will use SSE with Spring WebFlux. The solution is a good fit for us because we will produce a `Flux<Tweet>` and start to push the Tweets for our clients. The clients will send a query to filter the desired Tweets.

The application will be fully reactive. Let's take a look at our code:

```
package springfive.twitterdispatcher.domain.controller

import org.springframework.http.MediaType
import org.springframework.web.bind.annotation.GetMapping
import org.springframework.web.bind.annotation.RequestMapping
import org.springframework.web.bind.annotation.RequestParam
import org.springframework.web.bind.annotation.RestController
import reactor.core.publisher.Flux
import springfive.twitterdispatcher.domain.service.Tweet
import springfive.twitterdispatcher.domain.service.TwitterDispatcher

@RestController
@RequestMapping("/tweets")
class TweetResource(private val dispatcher: TwitterDispatcher) {

    @GetMapping(produces = [MediaType.TEXT_EVENT_STREAM_VALUE])
    fun tweets(@RequestParam("q") query: String): Flux<Tweet> {
        return dispatcher.dispatch()
            .filter({ tweet: Tweet? ->
                tweet!!.text.contains(query, ignoreCase = true)
            })
    }
}
```

Pretty easy and simple to understand. We have declared the `tweets()` function; this function is mapped to a GET HTTP Request and produces a `MediaType.TEXT_EVENT_STREAM_VALUE`. When the client connects to the endpoint, the server will start to send Tweets accordingly with the desired argument.

When the client disconnects, the Reactor RabbitMQ will close the requested RabbitMQ connection.

Dockerizing the whole solution

Now, it is time to wrap the whole solution and create a Docker image for all projects. It is useful to run the projects anywhere we want.

We will configure all the projects step by step and then run the solution in Docker containers. As a challenge, we can use `docker-compose` to orchestrate the whole solution in a single `yaml` file.

For the Tracked Hashtag Service, we have created the docker image. Then, we will start to configure the Tweet Gathering, and the last one is Tweet Dispatcher. Let's do that right now.



You can find more `docker-compose` project details at: <https://docs.docker.com/compose/>. Also, in the new versions, `docker-compose` supports Docker Swarm to orchestrate the stack between cluster nodes. It can be really useful to deploy Docker containers in production.

Tweet Gathering

Let's configure our `pom.xml` for the Tweet Gathering project.

The build node should look like the following:

```
<plugin>
    <groupId>io.fabric8</groupId>
    <artifactId>docker-maven-plugin</artifactId>
    <version>0.21.0</version>
    <configuration>
        <images>
            <image>
                <name>springfivebyexample/${project.build.finalName}</name>
```

```
<build>
    <from>openjdk:latest</from>
    <entryPoint>java -Dspring.profiles.active=container -jar
/application/${project.build.finalName}.jar</entryPoint>
    <assembly>
        <basedir>/application</basedir>
        <descriptorRef>artifact</descriptorRef>
        <inline>
            <id>assembly</id>
            <files>
                <file>
<source>target/${project.build.finalName}.jar</source>
                </file>
            </files>
        </inline>
    </assembly>
    <tags>
        <tag>latest</tag>
    </tags>
    <ports>
        <port>8081</port>
    </ports>
</build>
<run>
    <namingStrategy>alias</namingStrategy>
</run>
    <alias>${project.build.finalName}</alias>
</image>
</images>
</configuration>
</plugin>
```

Take a look at the port configuration; it should be the same as what we have configured in the `application.yaml`. The configuration is done, so let's create our Docker image:

```
mvn clean install docker:build
```

The command output should look like the following screenshot:

```
[INFO] DOCKER-> [springfivebyexample/tweet_gathering:latest] "tweet_gathering": Created docker-build.tar in 196 milliseconds
[INFO] DOCKER-> [springfivebyexample/tweet_gathering:latest] "tweet_gathering": Built image sha256:e1973
[INFO] DOCKER-> [springfivebyexample/tweet_gathering:latest] "tweet_gathering": Tag with latest
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 13.714 s
[INFO] Finished at: 2018-01-04T22:51:58-02:00
[INFO] Final Memory: 65M/524M
```

There is an image recently created and tagged as a latest; the image is ready to run. Let's do the same thing for our Tweet Dispatcher project.

Tweet Dispatcher

Our new plugin entry should look like this:

```
<plugin>
    <groupId>io.fabric8</groupId>
    <artifactId>docker-maven-plugin</artifactId>
    <version>0.21.0</version>
    <configuration>
        <images>
            <image>
                <name>springfivebyexample/${project.build.finalName}</name>
                <build>
                    <from>openjdk:latest</from>
                    <entryPoint>java -Dspring.profiles.active=container -jar
                        /application/${project.build.finalName}.jar</entryPoint>
                    <assembly>
                        <basedir>/application</basedir>
                        <descriptorRef>artifact</descriptorRef>
                        <inline>
                            <id>assembly</id>
                            <files>
                                <file>
                                    <source>target/${project.build.finalName}.jar</source>
                                    </file>
                                </files>
                            </inline>
                        </assembly>
                        <tags>
                            <tag>latest</tag>
                        </tags>
                        <ports>
                            <port>9099</port>
                        </ports>
                    </build>
                    <run>
                        <namingStrategy>alias</namingStrategy>
                    </run>
                    <alias>${project.build.finalName}</alias>
                </image>
            </images>
        </configuration>
    </plugin>
```

Take a look at the port configuration, one more time. It will be used by Docker to expose the correct port. Now, we can run the image creation command:

```
mvn clean install docker:build
```

Then, we can see the command's output, as shown in the following screenshot:

```
[INFO] DOCKER> [springfivebyexample/tweet_dispatcher:latest] "tweet_dispatcher": Created docker-build.tar in 164 milliseconds
[INFO] DOCKER> [springfivebyexample/tweet_dispatcher:latest] "tweet_dispatcher": Built image sha256:19317
[INFO] DOCKER> [springfivebyexample/tweet_dispatcher:latest] "tweet_dispatcher": Tag with latest
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 14.020 s
[INFO] Finished at: 2018-01-04T23:07:29-02:00
[INFO] Final Memory: 61M/528M
[INFO] -----
```

Awesome, all images are ready. Let's run it.

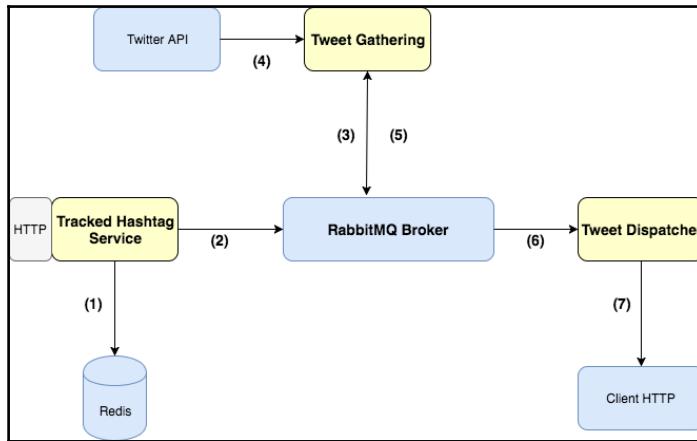


We need to create Docker images for all the projects. The process is the same; configure the maven Docker plugin and then use `mvn clean install docker:build` on the project. The full source code can be found at GitHub. The Tracked Hashtag Service can be found here (<https://github.com/PacktPublishing/Spring-5.0-By-Example/tree/master/Chapter04>), the Tweet Gathering can be found here (<https://github.com/PacktPublishing/Spring-5.0-By-Example/tree/master/Chapter05>) and finally, the Tweet Dispatcher can be found here (<https://github.com/PacktPublishing/Spring-5.0-By-Example/tree/master/Chapter06>).

Running the containerized solution

We are ready to run the solution in Docker containers. We have been running the solution with the IDE or command line, but now we will spin up some container and test the solution and Spring profiles as well.

Before that, let's do a quick recap of the solution:



1. The first operation, the **Tracked Hashtag Service**, will persist the hashtag in the **Redis** database.
2. After that, the **Tracked Hashtag Service** will send the newly tracked hashtag to a queue in the **RabbitMQ Broker**.
3. **Tweet Gathering** is listening to the queue to track Tweets and trigger the event and starts by listening to the **Twitter stream**.
4. **Tweet Gathering** starts to get Tweets from the **Twitter stream**.
5. **Tweet Gathering** publishes Tweets to a queue in the **RabbitMQ broker**.
6. **Tweet Dispatcher** consumes the message.
7. **Tweet Dispatcher** sends the message to the **Client** using SSE.

Now that we have understood the solution, let's start the containers.

Running the Tracked Hashtag Service container

The image has been created in the previous section, so now we are able to spin up the container. The command to start the container should look like this:

```
docker run -d --name tracked --net twitter -p 9090:9090
springfivebyexample/tracked_hashtag
```

Let's explain the instruction. `-d` tells the Docker engine to run the container in background mode or detached. The other important parameter is `--net`, which attaches the container to the desired network.

We can use the following command to tail the container logs at runtime:

```
docker logs tracked -f
```

This command is like the `tail -f` command on Linux, which looks at the last part of the log stream. We can remove the flag `-f` to see the last lines of the log.

The output of docker logs should look like this:

Look at the profile selected, in the logs:

```
INFO 7 --- [           main] s.t.TrackedHashTagApplication$Companion : The following profiles are active: docker
```

Remember, we have parameterized it in the pom.xml file from the Tracked Hash Tag Service. Let's look at the following snippet:

```
<entryPoint>java -Dspring.profiles.active=docker -jar  
/application/${project.build.finalName}.jar</entryPoint>
```

Awesome job. Our first service is running properly. Let's run Tweet Gathering; there is some interesting configuration here.



We have created the Twitter network in chapter 4, *Kotlin Basics and Spring Data Redis*, and we need to use this network to enable the containers to see each other by container name in our custom network.

Running the Tweet Gathering container

To run the **Tweet Gathering** application is slightly different. This container needs environment variables which are used to interact with the Twitter API. We can use the `-e` argument on the `docker run` command. Let's do that:

```
docker run -d --name gathering --net twitter -e  
CONSUMER_KEY=gupfxwn43NBTDxCD3Tsf1JgMu \  
-e CONSUMER_SECRET=pH4uM5L1YxKzfJ7huYRwFbaFXn7ooK01LmqCP69QV9a9kZrHw5 \  
\  
-e ACCESS_TOKEN=940015005860290560-m0WwSyxGvp5ufff9KW2zm5LGXLafLov \  
-e ACCESS_TOKEN_SECRET=KSofGB8aIwDmewceKXLbN8d5chvZkZyB31Vza09pNBhLo \  
-p 8081:8081 springfivebyexample/tweet_gathering
```

Take a look at the environment variables we have configured in the `application.yaml` file. The Docker run command will inject these variables into the system and then we can use them in our Java application.

Let's inspect our container logs. We can do that using the following command:

```
2018-01-15 23:54:29.084 INFO 6 --- [cTaskExecutor-1] o.s.a.r.c.CachingConnectionFactory      : Created new connection: rabbitConnectionFactory#57d7f78ca0/SimpleConnection@50ecfb4a [delegate=amqp://guest@172.19.0.6:5672/, localPort= 48016]  
2018-01-15 23:54:29.088 INFO 6 --- [cTaskExecutor-1] o.s.amqp.rabbit.core.RabbitAdmin       : Auto-declaring a non-durable, auto-delete, or exclusive Queue (twitter-stream) durable:false, auto-delete:false, exclusive:false. It will be redeclared if the broker stops and is restarted while the connection factory is alive, but all messages will be lost.  
2018-01-15 23:54:29.088 INFO 6 --- [cTaskExecutor-1] o.s.amqp.rabbit.core.RabbitAdmin       : Auto-declaring a non-durable, auto-delete, or exclusive Queue (twitter-track-hashtag) durable:false, auto-delete:false, exclusive:false. It will be redeclared if the broker stops and is restarted while the connection factory is alive, but all messages will be lost.  
2018-01-15 23:54:29.235 INFO 6 --- [           main] r.ipc.netty.tcp.BlockingNettyContext    : Started HttpServer on /0.0.0.0:8081  
2018-01-15 23:54:29.242 INFO 6 --- [           main] o.s.b.web.embedded.netty.NettyWebServer  : Netty started on port(s): 8081  
2018-01-15 23:54:29.246 INFO 6 --- [           main] s.t.TweetGatheringApplication$Companion  : Started TweetGatheringApplication.Companion in 20.599 seconds (JVM running for 21.708)
```

Awesome, our application is up and running. As you can see, the application is connected to the RabbitMQ Broker.



RabbitMQ and Redis should be running to enable you to run Tweet Gathering. We can check it using the `docker ps` command; it will list the running containers, RabbitMQ and Redis need to be on this list.

Now, we can run the Dispatcher application to complete the whole solution. Let's do that.

Running the Tweet Dispatcher container

There is no secret to running the Tweet Dispatcher container. We can use the following command to run it:

```
docker run -d --name dispatcher --net twitter -p 9099:9099
springfivebyexample/tweet_dispatcher
```

It will spin up the container, it is a good idea to name the container during the run. It can help us manage the container with command-line tools, such as `docker container ls` or `docker ps`, because it shows the container name in the last column. Then, let's check if our container is running, so type the following command:

```
docker container ls
```

Or, you can run the following command:

```
docker ps
```

We should be able to see the Gathering container running, like in the following output:

CONTAINER ID	IMAGE NAMES	COMMAND	CREATED	STATUS	PORTS
e266302dd499	springfivebyexample/tweet_dispatcher	"/bin/sh -c 'java -D..."	2 seconds ago	Up 3 seconds	0.0.0.0:9099->9099/tcp
282c1a3be5c2	springfivebyexample/tweet_gathering	"/bin/sh -c 'java -D..."	14 seconds ago	Up 16 seconds	0.0.0.0:8081->8081/tcp
c937b5cf2b03	gathering	springfivebyexample/tracked_hashtag	"/bin/sh -c 'java -D..."	31 seconds ago	0.0.0.0:9090->9090/tcp
b0e0ebb8219	rabbitmq:3.7.0-management-alpine	"docker-entrypoint.s..."	About a minute ago	Up About a minute	4369/tcp, 5671/tcp, 0.0.0.0:5672->5672/tcp, 15671/tcp, 25672/tcp, 0.0.0.0:15672->15672/tcp
1572a789b288	rabbitmq	"docker-entrypoint.s..."	2 minutes ago	Up 2 minutes	0.0.0.0:6379->6379/tcp
	redis	"docker-entrypoint.s..."			

There are five containers, three applications, and two infrastructure services, **RabbitMQ** and **Redis**.

At any time, we can stop the desired container using the following command:

```
docker stop gathering
```

The `docker stop` will only stop the container; the information will be kept in the container volume. We can use the container name or container ID as well, we named it before. It is easy for us. If we use the `docker ps` command, the image recently stopped will never appear on the list. To show all the containers, we can use `docker ps -a` or `docker container ls -a`.

Now, we will start the container again; the command is self-explanatory:

```
docker start gathering
```

The container is running again. We have practiced more with Docker.

Awesome job, guys. The whole application is containerized. Well done.



We can use the Linux instruction and execute some batch instructions. For instance, we can use `docker stop $(docker ps -q)` — it will stop all containers running. The `docker ps -q` command will bring only the container's IDs.

The docker-compose tool

In the microservices architectural style, the whole solution is decoupled in small and well-defined services. Usually, when we adopt these styles, we have more than one artifact to deploy.

Let's analyze our solution; we have three components to deploy. We have used the Docker containers and we have run these containers using the `docker run` command. One by one, we have used `docker run` three times. It is quite complex and very hard to do in the development routine.

`docker-compose` can help us in this scenario. It is a tool which helps to orchestrate Docker containers in complex scenarios like ours.

Let's imagine our application is growing fast and we need to build four more microservices to achieve the desired business case, it will implicate on four more `docker run` commands and will probably be painful to maintain, especially during the development life cycle. Sometimes, we need to promote the artifacts to test the environment and we probably need to modify our command line to achieve this.

`docker-compose` enables us to deploy multiple containers with a single `yml` file. This `yml` file has a defined structure which allows us to define and configure several containers in the same file. Moreover, we can run the solution configured in this `yml` file with a single command, it makes development life easy.

The tool can work on the local machine or we can integrate it with the Docker Swarm tool which can manage clusters of Docker hosts.

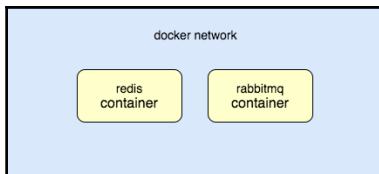


Docker Swarm is a native tool to manage docker clusters. It makes it easy to deploy a container on the Docker cluster. In the new version, `docker-compose` is fully integrated with Docker Swarm. We can define it from Docker Swarm properties in `docker-compose.yaml`. The Docker Swarm documentation can be found at: <https://docs.docker.com/engine/swarm/>.

The `docker-compose.yaml` has a defined structure to follow; the documentation can be found here: <https://docs.docker.com/compose/compose-file/#compose-and-docker-compatibility-matrix>. We will create a simple file to understand the `docker-compose` behaviors. Let's create our simple `yaml`—the `yaml` should look like this:

```
version: '3'
services:
  rabbitmq:
    image: rabbitmq:3.7.0-management-alpine
    ports:
      - "5672:5672"
      - "15672:15672"
  redis:
    image: "redis:alpine"
    ports:
      - "6379:6379"
```

The `yaml` in the preceding code will create the structure detailed in the following diagram:



It simplifies the development time. Now, we will learn how to install `docker-compose`.

Installing docker-compose

The `docker-compose` installation is pretty simple and well-documented. We are using Linux, so we will use the Linux instructions.

Open the terminal and use the following command:

```
sudo curl -L  
https://github.com/docker/compose/releases/download/1.18.0/docker-compose-`uname -s`-`uname -m` -o /usr/local/bin/docker-compose
```

Wait for the download and then we can execute the following instructions to give executable permissions for the program. Let's do this by executing the following command:

```
sudo chmod +x /usr/local/bin/docker-compose
```

As you may know, you may be asked for the administrator password. Our docker-compose is now installed. Let's check it:

```
docker-compose --version
```

The prompt will display the installed version, like the following screenshot:

```
docker-compose version 1.18.0, build 8dd22a9
```

docker-compose is up and running, so let's jump to the next section and start to create our yaml file and deploy the whole stack with one single command.



For different operating systems, the instructions can be found here: <https://docs.docker.com/compose/install/#install-compose>. Then, you can navigate around the instructions and click on the desired operating system.

Creating a docker-compose file

Now, we have docker-compose installed and we can try to work with the tool. We want to run the whole stack with a single command. We will create the yaml file to represent the stack. Our yaml file should have the Redis container, the RabbitMQ container, the Tracked Hashtag application, the Gathering application, and finally, the Dispatcher application.

We can create a docker-compose.yaml file wherever we want, there is no restriction for that.

Our docker-compose.yaml file should look like the following:

```
version: '3'
services:
  rabbitmq:
    image: rabbitmq:3.7.0-management-alpine
    hostname: rabbitmq
    ports:
      - "5672:5672"
      - "15672:15672"
    networks:
      - solution
  redis:
    image: "redis:4.0.6-alpine"
    hostname: redis
    ports:
      - "6379:6379"
    networks:
      - solution
  tracked:
    image: springfivebyexample/tracked_hashtag
    ports:
      - "9090:9090"
    networks:
      - solution
  gathering:
    image: springfivebyexample/tweet_gathering
    ports:
      - "8081:8081"
    networks:
      - solution
    environment:
      - CONSUMER_KEY=gupfxwn43NBTDxCD3Tsf1JgMu
      -
      CONSUMER_SECRET=pH4uM5LlYxKzfJ7huYRwFbaFXn7ooK01LmqCP69QV9a9kZrHw5
      - ACCESS_TOKEN=940015005860290560-
      m0WwSyxGvp5ufff9KW2zm5LGXLafLov
      -
      ACCESS_TOKEN_SECRET=KSofGB8aIwDmewceKXLbN8d5chvZkZyB31Vza09pNBhLo
    dispatcher:
      image: springfivebyexample/tweet_dispatcher
      ports:
        - "9099:9099"
      networks:
        - solution
networks:
  solution:
    driver: bridge
```

As you can see, we have defined the whole stack in the `yaml`. Something to note is that we can find some similarities with the `docker run` command, in fact, it will use the Docker engine to run. The `environment` node in `yaml` has the same behavior as `-e` in the Docker run command.

We have defined the application ports, docker images, and have also connected the containers to the same network. This is really important because when we use the `docker-compose` file name on the network, it can find that the container name has a kind of DNS behavior.

For instance, inside the defined network `solution`, the container can find the Redis container instance by the name `redis`.

Running the solution

`docker-compose` simplifies the process to run the whole stack. Our `yaml` file was configured and defined properly.

Let's start the solution. Run the following command:

```
docker-compose up -d
```

The command is pretty simple, the `-d` parameter instructs Docker to run the command in the background. As we did on the Docker run command.

The output of this command should be the following:

```
Creating network "compose_solution" with driver "bridge"
Creating compose_gathering_1 ... done
Creating compose_redis_1 ... done
Creating compose_tracked_1 ... done
Creating compose_rabbitmq_1 ... done
Creating compose_dispatcher_1 ... done
```

Take a look, `docker-compose` has created a network for our stack. In our case, the network driver is a bridge, after the network creation, the containers are started.

Testing the network

Let's test it, find the Gathering container – the container name in docker-compose is prefixed by the folder name, where docker-compose was started.

For instance, I have started my docker-compose stack in the compose folder. My container name will be `compose_gathering_1` because of the folder name.

Then, we will connect the Gathering container. It can be achieved using the following command:

```
docker exec -it compose_gathering_1 /bin/bash
```

The `docker exec` command allows us to execute something inside the container. In our case, we will execute the `/bin/bash` program.

The command structure is like this:

```
docker exec -it <container name or container id> <program or instruction>
```

Awesome, pay attention to the command line. It should be changed because now we are in the container command line:

```
root@cc6520b2bdc5:/# ls -l
total 68
drwxr-xr-x  2 root root 4096 Jan 11 00:11 application
drwxr-xr-x  1 root root 4096 Sep 14 04:18 bin
```

We are not connected as a root on our host, but now we are a root on the container. This container is on the same network as the Redis container instance, which is called `redis`.

Let's test with the `ping` command; we should be able to find the `redis` container by the name `redis`, let's do it. Type the following:

```
ping redis
```

The command output should be the following:

```
root@cc6520b2bdc5:/# ping redis
PING redis (172.19.0.2): 56 data bytes
64 bytes from 172.19.0.2: icmp_seq=0 ttl=64 time=0.280 ms
64 bytes from 172.19.0.2: icmp_seq=1 ttl=64 time=0.368 ms
64 bytes from 172.19.0.2: icmp_seq=2 ttl=64 time=0.221 ms
64 bytes from 172.19.0.2: icmp_seq=3 ttl=64 time=0.255 ms
64 bytes from 172.19.0.2: icmp_seq=4 ttl=64 time=0.310 ms
^C--- redis ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.221/0.287/0.368/0.050 ms
```

Awesome, our container can find the Redis container by the name. The `yaml` file is fully working.

Summary

In this chapter, we completed our second solution. We were introduced to the RabbitMQ Reactor library, which enables us to connect to RabbitMQ, using the reactive paradigm.

We have prepared the whole solution in Docker containers and connected it to the same network to enable the applications to talk to each other.

We also learned the important pattern for pushing data from server to client through the HTTP persistent connection, and we learned the difference between WebSockets and Server-Sent Events, as well.

Finally, we learned how `docker-compose` helps us to create the stack and run the whole solution with a couple of commands.

In the following chapters, we will build a fully microservice solution, using some important patterns such as Service Discovery, API Gateway, Circuit Breakers, and much more.

7

Airline Ticket System

Our last projects—Twitter Consumers, Twitter Gathering, and Twitter Dispatcher—were excellent. We learned several exciting features, and they were implemented using the new features present in Spring 5.0. All of them are implemented in Reactive Streams and use Kotlin as the programming language. They are the hottest features in Spring 5.0; it was an impressive progression.

However, there are notably missing parts on these projects; we have microservice needs in mind. There are no infrastructure services such as service discovery, distributed configurations, API Gateway, distributed tracing, and monitoring. These kinds of services are mandatory in distributed systems such as microservice architectures.

There are several reasons for that. Firstly, we can think of the configuration management. Let's imagine the following scenario – in the development cycle, we have three environments: DEV, TST, and PROD. This is a pretty simple standard found in companies. Also, we have an application decoupled in 4 microservices, then with the minimum infrastructure, we have 12 instances of services; remember, this is a good scenario because in a real situation, we will probably have several instances of microservice applications.

In the earlier scenario, we will maintain at least three configuration files per microservice, remember there are three environments for which we need to keep the configurations. Then, we will have 12 *versions* of settings. It is a hard task to maintain the configurations, to keep the files synchronized and updated. These files probably contain sensitive information, such as database passwords and message brokers' configurations, and it is not recommended that you put these files on the host machines.

In this case, the distributed configuration can solve our problems easily. We will learn about configuration servers in this chapter, and other infrastructure services as well.

Let's summarize what we will learn in this chapter:

- How to create a Config Server
- Implementing a service discovery with Eureka
- Monitoring applications with Spring Cloud Zipkin
- Exposing the applications with the Spring Cloud Gateway

The Airline Ticket System

In these last few chapters, we will work on the Airline Ticket System. The solution is quite complex and involves a lot of HTTP integrations and message-based solutions. We will explore what we have learned from the book journey.

We will use Spring Messaging, Spring WebFlux, and Spring Data components to create the solution. The application will split up into several microservices to guarantee the scalability, elasticity, and fault tolerance for the system.

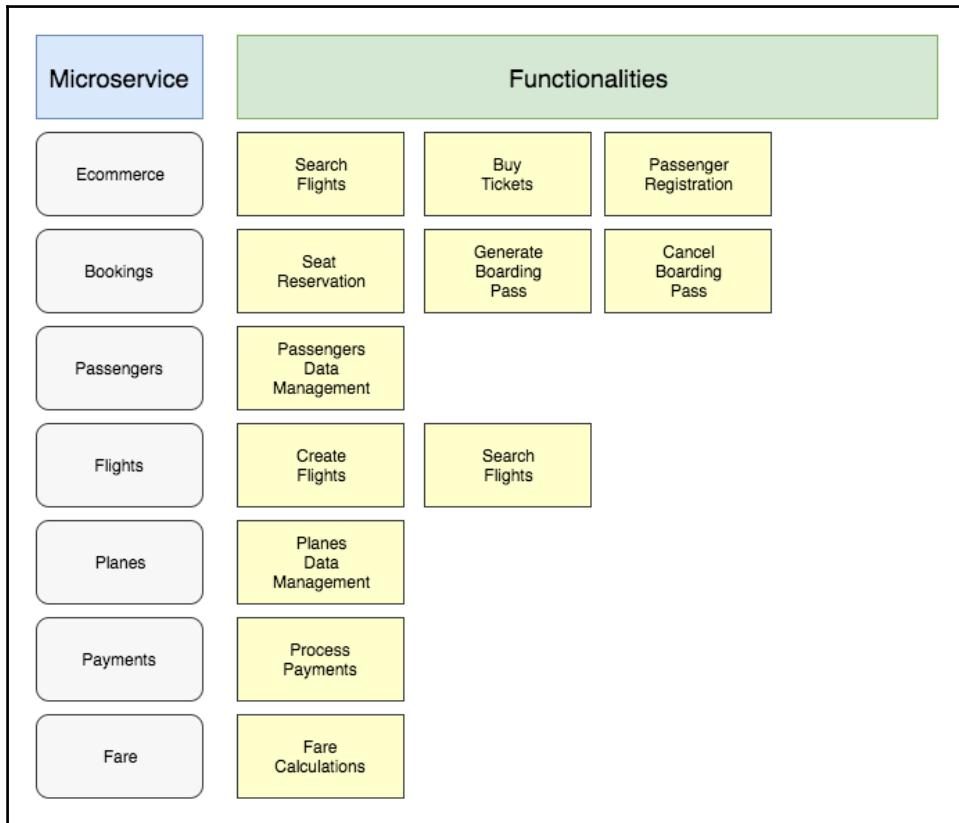
Also, we will have some infrastructure services to help us deliver an efficient system. Some new patterns will be introduced, such as circuit breakers and OAuth. In the infrastructure layer, we will use the Netflix OSS components integrated with the Spring Framework ecosystem.

The main purpose of our application is to sell airline tickets, but to achieve this task, we need to build an entire ecosystem. We will build a microservice which will manage the seats and planes' characteristics. There will also be a microservice to manage available company flights; the basic idea is to manage flight dates and routes. Of course, we will have a microservice to manage passengers, fares, bookings, and payments. Finally, we will have an e-commerce API with which end users will buy airline tickets.

Airline functionalities

We will create some microservices to compose the solution and then we will decompose the solution into small pieces, that is, microservices. For that, we will use the Bounded Context pattern which is an essential part of the **Domain-Driven Design (DDD)**.

Let's look at the following diagram to have an idea about what we will build:

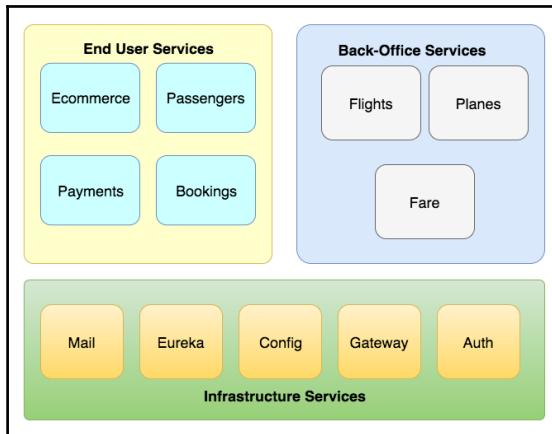


It is a summary of what we will do in these few chapters; we have defined the basic functionalities for each microservice.

Now, we will take a look at components; let's go to the next section.

Solution diagram

The following diagram illustrates the whole solution, which we will implement in the following chapters:



As we can see, there are different kinds of components. Some components will be exposed through the **Gateway** for end users, in our case, our customers. There is a category which the company users will use to register flights, for instance, where these microservices will be exposed on **Gateway** as well.

The infrastructure category will not be exposed over the internet, except the **Gateway** service. These services help the solution infrastructure and should be not exposed because there is sensitive data in there.

There a lot of things to do; let's get on with the show.



DDD enables us to deal easily with microservices. Some DDD patterns fit well for the microservices architectural style. There are many interesting books in the Packt catalog.

Spring Cloud Config Server

When we adopt the microservices architectural style, there are some challenges to solve. One of the first problems to solve is how to manage the microservices configurations in the cluster, and how to make them easy and distributed, as well?

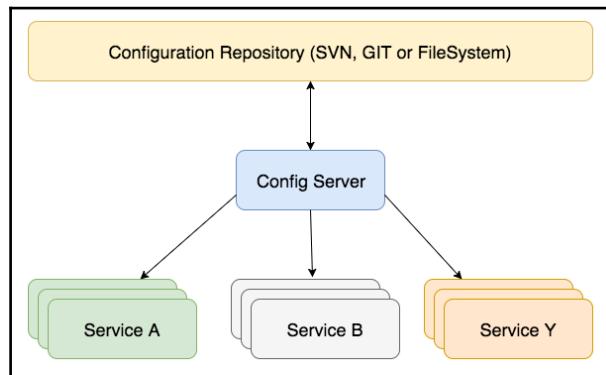
Spring Cloud Config provides a Spring way, based on annotations and Spring beans. It is an easy way to solve this problem in a production-ready module. There are three main components in this module, the Configuration Repository, that is, version control system, the Config Server, which will provide the configurations, and finally, the Configuration Client, which will consume the configuration from the Config Server.

This module supplies the configuration files over an HTTP interface. It is the main feature provided by this project and it acts as a central repository for configuration in our architecture.

We want to remove the `application.yaml` file from our classpath; we do not need this file in classpath anymore, and so we will use the Config Server to serve this file for our application.

Now, our microservices will not have the configuration file, that is, `application.yaml`. During the application bootstrap, the application will look at the Config Server to get the correct configuration, and after that, the application will finish the bootstrap to get them up and into running status.

The following diagram explains the **Config Server** and Config Client:



As we can see, the basic idea here is to try to distribute the configuration through the **Config Server**. There are some advantages to using this approach. The first one keeps the configuration in the central repository. It makes the configuration easy to maintain. The second one is that the configurations are served with a standard protocol, such as HTTP. Most of the developers know the protocol and make the interaction easy to understand. Finally, and most importantly, when the properties change, it can reflect immediately in other microservices.

Time to implement it. Let's go there.



The Config Server is usually maintained on private networks, if we are deploying in cloud environments, although the Spring Cloud Config supports encrypt and decrypt based on symmetric or asymmetric keys. Keep in the mind that the microservices configurations should not be published on public networks.

Creating the Config Server project

Let's create our project with Spring Initializr. Go to Spring Initializr (<https://start.spring.io/>) and follow the image instructions:

The screenshot shows the Spring Initializr interface. At the top, it says "SPRING INITIALIZR bootstrap your application now". Below that, it asks "Generate a [Maven Project] with [Java] and Spring Boot [2.0.0 M7]". On the left, under "Project Metadata", it shows "Artifact coordinates" with "Group" set to "springfive.airline" and "Artifact" set to "config-server". On the right, under "Dependencies", it says "Add Spring Boot Starters and dependencies to your application" and "Search for dependencies" with "Web, Security, JPA, Actuator, Devtools...". Below this, "Selected Dependencies" include "Config Server", "Reactive Web", and "Actuator". At the bottom center is a green "Generate Project" button. A small note at the bottom says "Don't know what to look for? Want more options? Switch to the full version."

Click on **Generate Project** and then we can open the project on the IDE.

Enabling Spring Cloud Config Server

We will use the Git repository as a property source, and then we need to create a repository to keep these files. However, before that, let's navigate to the `pom.xml` file and see some interesting stuff. We can find the following dependency:

```
<dependency>
```

```
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-config-server</artifactId>
</dependency>
```

It is a Config Server dependency. It enables us to use the Config Server in our application. Remember, we need to put this into the `pom.xml` file to achieve the required Config Server.

Using GitHub as a repository

The Spring Cloud Config Server enables us to use different datastore technologies to work as a properties repository. There are some options such as Git repository, filesystem, or SVN and others, provided by the community.

We will choose the Git repository, and use GitHub as a host.



We will use the Git repository that has the source code of the book.

The repository is located at: <https://GitHub.com/PacktPublishing/Spring-5.0-By-Example/tree/master/config-files>.

The Spring Cloud Config Server also supports private repositories. For that purpose, we need to supply the private/public keys.

Configuring the Spring Boot application

It's a piece of cake to enable and run the Config Server and provide our configuration HTTP protocol. To achieve it, we need to put the following annotation in our Spring Boot starter class. The implementation is as follows:

```
package springfive.airline.configserver;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.config.server.EnableConfigServer;

@EnableConfigServer
@SpringBootApplication
public class ConfigServerApplication {

    public static void main(String[] args) {
        SpringApplication.run(ConfigServerApplication.class,
        args);
    }
}
```

```
    }  
}  
}
```

Awesome. `@EnableConfigServer` does the magic for us. It will stand up the Config Server and make the application ready to connect.

Configuring the Git repository as a properties source

Our Config Server needs to be configured. For that purpose, we will use the `application.yaml` file. This file should be simple and with minimal configurations as well. The configuration file should look like this:

```
server:  
  port: 5000  
  
spring:  
  cloud:  
    config:  
      name: configserver  
      server:  
        git:  
          uri:  
            https://github.com/PacktPublishing/Spring-5.0-By-Example  
          search-paths: config-files*
```

We have configured the application port, which is a common task. We named our Config Server, and the most important part is the `server.git.uri` configuration property which instructs the Spring Framework to get the configurations files.

Another configuration is `search-paths`; it allows us to search the configuration in `git` repository folders, instead of a root address in the repository.

Running the Config Server

Awesome job; our configuration server is ready to use. Then let's run it. We can use the JAR file, or through IDE as well, it is up to you to choose the desired way.

We can use the Java command line or IDE to run it. I prefer to use IDE because it enables us to debug and make some code changes.

Run it.

The output should look like this:

```
2018-01-06 13:38:46.115 INFO 7815 — [           main] o.s.j.e.a.AnnotationMBeanExporter : Bean with name 'configurationPropertiesRebinder' has been autodetected for JMX exposure
2018-01-06 13:38:46.116 INFO 7815 — [           main] o.s.j.e.a.AnnotationMBeanExporter : Bean with name 'environmentManager' has been autodetected for JMX exposure
2018-01-06 13:38:46.117 INFO 7815 — [           main] o.s.j.e.a.AnnotationMBeanExporter : Bean with name 'refreshScope' has been autodetected for JMX exposure
2018-01-06 13:38:46.119 INFO 7815 — [           main] o.s.j.e.a.AnnotationMBeanExporter : Located managed bean 'environmentManager' as MBean [org.springframework.cloud.config.server.environment.MBean]
2018-01-06 13:38:46.120 INFO 7815 — [           main] o.s.j.e.a.AnnotationMBeanExporter : Located managed bean 'refreshScope' as MBean [org.springframework.cloud.config.server.RefreshScopeMBean]
2018-01-06 13:38:46.126 INFO 7815 — [           main] o.s.j.e.a.AnnotationMBeanExporter : Located managed bean 'configurationPropertiesRebinder' as MBean [org.springframework.cloud.config.server.configurationPropertiesRebinderMBean]
2018-01-06 13:38:46.133 INFO 7815 — [           main] o.s.j.e.a.AnnotationMBeanExporter : Located managed bean 'configurationPropertiesRebinder' : registering with JMX server as MBean [org.springframework.cloud.config.server.configurationPropertiesRebinderMBean]
2018-01-06 13:38:46.226 INFO 7815 — [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 5000 (http) with context path ''
```

Tomcat started successfully; our Config Server is up and running. We can find some different endpoints in our Config Server. These endpoints are exposed to serve the configuration file.

The Spring Cloud Config Server supports profiles as well, providing different configurations for different environments is important.

The pattern supported by the Config Server is as follows:

```
<application-name>-<profile>. <properties|yaml>
```

It is really important to keep this in mind. Also, it makes it mandatory to declare the `application.name` property in our microservices, to identify the application.

We can find the endpoints provided by the Spring Cloud Config Server on the application bootstrap. Take a look at the log:

```
2018-01-06 13:38:44.734 INFO 7815 — [ost-startStop-1] s.w.s.m.a.RequestMappingHandlerMapping : Mapped “[{!decrypt},methods=[POST]}” onto public java.lang.String org.springframework.cloud.config.server.environment.RequestMappingHandlerMapping
2018-01-06 13:38:44.734 INFO 7815 — [ost-startStop-1] s.w.s.m.a.RequestMappingHandlerMapping : Mapped “[{!decrypt},methods=[PUT]}” onto public java.lang.String org.springframework.cloud.config.server.environment.RequestMappingHandlerMapping
2018-01-06 13:38:44.734 INFO 7815 — [ost-startStop-1] s.w.s.m.a.RequestMappingHandlerMapping : Mapped “[{!key}/{name}/profiles/{profiles},methods=[GET]}” onto public java.lang.String org.springframework.cloud.config.server.environment.RequestMappingHandlerMapping
2018-01-06 13:38:44.734 INFO 7815 — [ost-startStop-1] s.w.s.m.a.RequestMappingHandlerMapping : Mapped “[{!key}/{name}/profiles/{profiles}.yml || /{name}-profiles.yaml,methods=[GET]}” onto public java.lang.String org.springframework.cloud.config.server.environment.RequestMappingHandlerMapping
2018-01-06 13:38:44.741 INFO 7815 — [ost-startStop-1] s.w.s.m.a.RequestMappingHandlerMapping : Mapped “[{!name}-{profiles},properties,methods=[GET]}” onto public org.springframework.cloud.config.server.environment.RequestMappingHandlerMapping
2018-01-06 13:38:44.741 INFO 7815 — [ost-startStop-1] s.w.s.m.a.RequestMappingHandlerMapping : Mapped “[{!name}/{profiles},{label:,*},methods=[GET]}” onto public org.springframework.cloud.config.server.environment.RequestMappingHandlerMapping
2018-01-06 13:38:44.741 INFO 7815 — [ost-startStop-1] s.w.s.m.a.RequestMappingHandlerMapping : Mapped “[{!label}/{name}-{profiles},properties,methods=[GET]}” onto public org.springframework.cloud.config.server.environment.RequestMappingHandlerMapping
2018-01-06 13:38:44.742 INFO 7815 — [ost-startStop-1] s.w.s.m.a.RequestMappingHandlerMapping : Mapped “[{!label}/{name}-{profiles}.json},methods=[GET]}” onto public org.springframework.cloud.config.server.environment.RequestMappingHandlerMapping
2018-01-06 13:38:44.743 INFO 7815 — [ost-startStop-1] s.w.s.m.a.RequestMappingHandlerMapping : Mapped “[{!label}/{name}-{profiles}.yml || /{Label}/{name}-{profiles}.yaml},methods=[GET]}” onto public org.springframework.cloud.config.server.environment.RequestMappingHandlerMapping
2018-01-06 13:38:44.743 INFO 7815 — [ost-startStop-1] s.w.s.m.a.RequestMappingHandlerMapping : Mapped “[{!label}/{name}/{profiles}..{!*},methods=[GET]}” onto public org.springframework.cloud.config.server.environment.RequestMappingHandlerMapping
2018-01-06 13:38:44.747 INFO 7815 — [ost-startStop-1] s.w.s.m.a.RequestMappingHandlerMapping : Mapped “[{!name}/{profile}/{label}/*},methods=[GET]},produces=[application/octet-stream]}” onto public org.springframework.cloud.config.server.environment.RequestMappingHandlerMapping
2018-01-06 13:38:44.748 INFO 7815 — [ost-startStop-1] s.w.s.m.a.RequestMappingHandlerMapping : Mapped “[{!name}/{profile}/{label}/*},methods=[GET]}” onto public java.lang.String org.springframework.cloud.config.server.environment.RequestMappingHandlerMapping
2018-01-06 13:38:44.748 INFO 7815 — [ost-startStop-1] s.w.s.m.a.RequestMappingHandlerMapping : Mapped “[{!name}/{profile}/{label}/*},methods=[GET]},parameters=[multimap]}” onto public org.springframework.cloud.config.server.environment.RequestMappingHandlerMapping
2018-01-06 13:38:44.781 INFO 7815 — [ost-startStop-1] o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path ‘/**’ onto handler of type [class org.springframework.web.servlet.resource.ResourceHttpRequestHandler]
2018-01-06 13:38:44.781 INFO 7815 — [ost-startStop-1] o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path ‘/***favicon.ico’ onto handler of type [class org.springframework.web.ser
```

Remember the Config Server supports environments; because of this, there is a kind of regex on endpoints. Look at the `/ {name} - {profiles}.yml` endpoint.

Testing our Config Server

We are able to test our Config Server over the REST API.

Let's create a simple `yaml` file to create the test; the file should be called `dummy.yaml`:

```
info:
  message: "Testing my Config Server"
  status: "It worked"
```

Push it to GitHub – if you are using the GitHub book, this step is unnecessary. Then, we can call the Config Server API using the following command:

```
curl http://localhost:5000/dummy/default | jq
```

The command looks for the `dummy` configuration in the profile `default`; the URL is self-explanatory. The following output should be displayed:

```
% Total    % Received % Xferd  Average Speed   Time     Time      Time  Current
          Dload  Upload Total   Spent  Left Speed
100  309    0  309    0     0   261     0 --:--:--  0:00:01 --:--:--  261
{
  "name": "dummy",
  "profiles": [
    "default"
  ],
  "label": null,
  "version": "bca0b9ad5fdd1f853744d7dc2abf92411423e2b1",
  "state": null,
  "propertySources": [
    {
      "name": "https://github.com/PacktPublishing/Spring-5.0-By-Example/config-files/dummy.yaml",
      "source": {
        "info.message": "Testing my Config Server",
        "info.status": "It worked"
      }
    }
  ]
}
```

Our Config Server is fully operational. Now, we will configure our service discovery using Netflix Eureka.

Spring Cloud service discovery

The service discovery is one of the key points of the microservices architecture. The basis of the microservices architecture is to decouple the monolithic application into smaller pieces of software which have well-defined boundaries.

This impacts our system design in the monolithic application. In general, the application logic stays in a single place with regards to the code. It means the procedure or methods calls are invoked in the same context when the application is running.

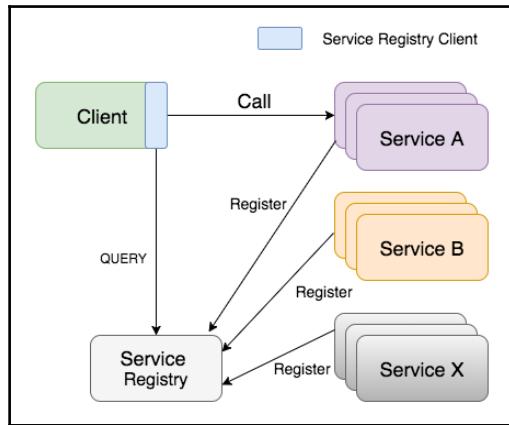
When we adopt the microservices architectural style, these invocations are typically external, in other words, they will invoke the service through HTTP calls, for example, in another application context or web server.

Then, the services need to call other services through HTTP, for instance, but how do the services call the others if the instances of these services change with a considerable frequency? Remember, we are creating distributed and scalable systems, where the instances of services can be increased according to the system usage.

The services need to know where the other services are running to be able to call them. Let's imagine that we are considering putting the services IPs in the configuration; it will be hard to manage and impossible to track the machine changes during that time.

The service discovery pattern addresses this challenge. In general, the solution involves a Service Registry, which knows the locations of all the running services. The client then needs to have a kind of Service Registry Client to be able to query this Service Registry to obtain the valid address for the desired service; the Service Registry will then return a healthy address, and finally, the client can invoke the desired service.

Let's look at the following diagram:



The full documentation of this pattern can be found at <http://microservices.io/patterns/client-side-discovery.html> and <https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/>. There are so many implementations for that pattern.

The Spring Cloud service discovery supports some service discovery implementations, such as Hashicorp Consul provided by the Spring Cloud Consul, and Apache Zookeeper provided by the Spring Cloud Zookeeper.

We are using the Netflix OSS stack where we will use the Eureka server, which was provided by the Spring Netflix OSS. It enables us to use the Eureka server as a managed Spring bean.

The Spring Eureka Client provides a client aware of the Service Registry, and it can be done with a couple of annotations and some configurations – we will do that soon.

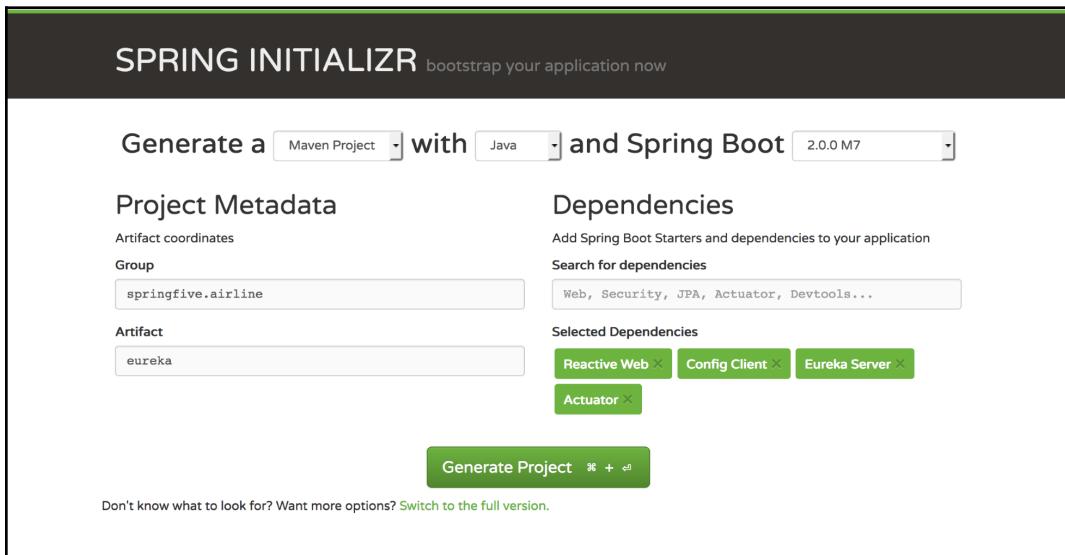
We will start to create and configure the Eureka server in the following sections. Let's do that.



The full documentation for the Spring Cloud Consul can be found at: <https://cloud.spring.io/spring-cloud-consul>, and the Spring Cloud Zookeeper can be found at: <https://cloud.spring.io/spring-cloud-zookeeper>.

Creating Spring Cloud Eureka

To enable service discovery in our infrastructure, we need to create an instance of a service which will act as a service discovery. The Spring Cloud Eureka server enables us to achieve this task. Let's create our project. Go to Spring Initializr and fill in the information, as shown in the following screenshot:



Take a look at the required dependencies. The Eureka server is the dependency which allows us to spin up a service discovery server.

Let's open the project on IDE and start to configure it. We will do this in the following section.

Creating the Eureka server main class

Before we start the configuration, we will create the main class. This class will start the Spring Boot application. The Eureka server is embedded in the application. It is a pretty standard Spring Boot application with a single annotation.

The main application class should look like this:

```
package springfive.airline.eureka;

import org.springframework.boot.SpringApplication;
import
org.springframework.boot.autoconfigure.SpringBootApplication;
import
org.springframework.cloud.netflix.eureka.server.EnableEurekaSe
rver;

@EnableEurekaServer
@SpringBootApplication
public class EurekaApplication {

    public static void main(String[] args) {
        SpringApplication.run(EurekaApplication.class, args);
    }

}
```

The `@EnableEurekaServer` annotation will start the embedded Eureka server in our application and make it ready to use. It will enable the service registry in our application as well.

Configuring the Spring Cloud Eureka server

Our Eureka server needs to be configured using the Spring Cloud Server configured in the previous sections. Then, we need to keep the `application.yaml` off our project, to use the Config Server properly. Instead of the `application.yaml`, we need to put the `bootstrap.yaml` and put the Config Server address on it.

Then, we need to:

- Create `discovery.yaml` on GitHub
- Create `bootstrap.yaml` file in the classpath project

Let's start with the `discovery.yaml` file. The file should look like this:

```
server:
  port: 8761

eureka:
  instance:
    hostname: localhost
```

```
    health-check-url-path: /actuator/health
    status-page-url-path: /actuator/info
  client:
    registerWithEureka: false
    fetchRegistry: false
  logging:
    level:
      com.netflix.discovery: 'ON'
      org.springframework.cloud: 'DEBUG'
```

There are some interesting things to explore. We are using the localhost as hostname because we are running on the developer machine. There are a couple of configurations about the URLs health check and status page – pay attention to the configurations that are related to the server. They are placed below the eureka.instance YAML node. The configurations are health-check-url-path and status-page-url-path. We can use the default values as well, but the new Spring Boot Actuator changes the URL for those two features, so we need to configure them properly.

The eureka.client YAML node is about the client configuration; in our case, we set registerWithEureka to false. We do not want the Eureka server to act as a client as well. The same is true for the fetchRegistry configuration, it is a client configuration and it will cache the Eureka registry's information.

The logging node is about logging configuration.

Awesome – our gateway.yaml is ready.

Let's create our bootstrap.yaml file in the Eureka server project classpath. The file should look like this:

```
spring:
  application:
    name: discovery
  cloud:
    config:
      uri: http://localhost:5000
      label: master
```

Easy peasy – we have configured spring.cloud.config. It instructs Spring of the Config Server address. Also, we have configured the label, which is the branch when we are using the **version control system (VCS)** as a repository.

Well done. The configuration is ready. Time to run it. Let's do it in the following section.

Running the Spring Cloud Eureka server

The Eureka server is ready to use. We will start the Spring Boot application and put our Eureka server online. We can use the Java command line or IDE to run it. I prefer to use IDE because it enables us to debug and make some code changes.



The Config Server needs to be running because the discovery will find the configuration file to bootstrap the server properly.

Run it!

We should see the following lines in the application bootstrap logs:

```
2018-01-07 14:42:42.602 INFO 11191 --- [           main] o.s.c.support.DefaultLifecycleProcessor : Starting beans in phase 0
2018-01-07 14:42:42.602 INFO 11191 --- [           main] o.s.c.m.e.s.EurekaServiceRegistry      : Registering application discovery with eureka with status UP
2018-01-07 14:42:42.605 DEBUG 11191 --- [           main] s.c.c.d.h.DiscoveryClientHealthIndicator : Discovery Client has been initialized
2018-01-07 14:42:42.608 INFO 11191 --- [           Thread-32] o.s.c.m.e.server.EurekaServerBootstrap  : Setting the eureka configuration..
2018-01-07 14:42:42.610 INFO 11191 --- [           Thread-32] o.s.c.m.e.server.EurekaServerBootstrap  : Eureka data center value eureka.datacenter is not set, defaulting to default
2018-01-07 14:42:42.612 INFO 11191 --- [           Thread-32] o.s.c.m.e.server.EurekaServerBootstrap  : Eureka environment value eureka.environment is not set, defaulting to test
2018-01-07 14:42:42.614 INFO 11191 --- [           Thread-32] o.s.c.m.e.server.EurekaServerBootstrap  : Aws region is false
2018-01-07 14:42:42.616 INFO 11191 --- [           Thread-32] o.s.c.m.e.server.EurekaServerBootstrap  : Ignored context context
2018-01-07 14:42:42.619 INFO 11191 --- [           Thread-32] c.n.e.r.PeerAwareInstanceRegistryImpl   : Got 1 instances from neighboring DS node
2018-01-07 14:42:42.622 INFO 11191 --- [           Thread-32] c.n.e.r.PeerAwareInstanceRegistryImpl   : Renew threshold is: 1
2018-01-07 14:42:42.624 INFO 11191 --- [           Thread-32] c.n.e.r.PeerAwareInstanceRegistryImpl   : Changing status to UP
2018-01-07 14:42:42.625 INFO 11191 --- [           Thread-32] c.n.e.r.PeerAwareInstanceRegistryImpl   : Started Eureka Server
2018-01-07 14:42:42.636 INFO 11191 --- [           Thread-32] e.s.EurekaServerInitializerConfiguration : Tomcat started on port(s): 8761 (http) with context path ''
2018-01-07 14:42:42.696 INFO 11191 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8761
2018-01-07 14:42:42.697 INFO 11191 --- [           main] s.c.m.e.s.EurekaAutoServiceRegistration : Updating port to 8761
2018-01-07 14:42:42.702 INFO 11191 --- [           main] s.airline.eureka.EurekaApplication     : Started EurekaApplication in 10.862 seconds (JVM running for 11.719)
2018-01-07 14:42:43.067 INFO 11191 --- [on(8)-127.0.0.1] c.c.CConfigServicePropertySourceLocator : Fetching config from server at: http://localhost:5000
```

Awesome. Look at the following line of the log:

```
2018-01-07 14:42:42.636 INFO 11191 --- [           Thread-32]
e.s.EurekaServerInitializerConfiguration : Started Eureka Server
```

It means our Eureka server is ready to use. To check the solution, we can go to the Eureka server home page. Go to <http://localhost:8761> and the following page will be displayed:

The screenshot shows the Spring Eureka dashboard. At the top, there's a header with the Eureka logo and the text "spring Eureka". On the right side of the header, it says "HOME" and "LAST 1000 SINCE STARTUP". Below the header, there's a section titled "System Status" which contains two tables. The first table has two rows: "Environment" (test) and "Data center" (default). The second table has five rows: "Current time" (2018-01-07T14:55:47 -0200), "Uptime" (00:13), "Lease expiration enabled" (true), "Renews threshold" (1), and "Renews (last min)" (2). Underneath these tables, there's a section titled "DS Replicas" with a sub-section "Instances currently registered with Eureka". A table header row is shown with columns "Application", "AMIs", "Availability Zones", and "Status". Below this, a message says "No instances available".

As we can see, there is no instance of service available yet. We can find some relevant information such as the server **Uptime**, the current **Data center**, and the **Current time**. There is some information in the **General Info** section, information regarding the server where the Eureka server is running.

Good job. Our service discovery service is running. We will use this infrastructure soon.

Spring Cloud Zipkin server and Sleuth

Our solution involves some microservices; it makes our solution easy to deploy and easy to write code. Each solution has a particular repository and codebase.

In the monolith solution, the whole problem is solved in the same artifact to be deployed. Usually, in Java, these artifacts are `.jar`, `.war`, or `.ear`, if the application was written in the Java EE 5/6 specifications.

The logging strategies for these kinds of applications is quite easy to work with (hence problems can be solved easily) because everything happens in the same context; the requests are received from the same application server or web server, which have the business components. Now, if we go to the logs, we will probably find the log entries we want. It makes the trace application easier to find errors and debug.

In the microservices solution, the application behaviors are split in the distributed systems; it increases the trace tasks substantially because the request probably arrives in the API Gateway and comes into microservices. They log the information in different sources. In this scenario, we need a kind of log aggregator and a way to identify the whole transaction between services.

For this purpose, the Spring Cloud Sleuth and Spring Cloud Zipkin can help us and make the trace features more comfortable for developers.

In this section, we will look at and understand how it works under the hood.

Infrastructure for the Zipkin server

Before we start to work, we need to configure a service which the Zipkin server needs. By default, the Zipkin server uses in-memory databases, but it is not recommended for production; usually, developers use this feature to demonstrate Zipkin features.

We will use MySQL as a data store. The Zipkin server also supports different sources, such as Cassandra and Elasticsearch.

Spring Cloud Sleuth supports synchronous and asynchronous operations. The synchronous operations are over the HTTP protocol and asynchronous can be done by RabbitMQ or Apache Kafka.

To use the HTTP, that is, REST API, we should use `@EnableZipkinServer`, it will delegate the persistence for REST tier through the `SpanStore` interface.

We will choose the asynchronous solution, since it fits well for our project, and we do not want the trace collector to cause some performance issues. The asynchronous solution uses the Spring Cloud Stream binder to store the Spans. We choose the RabbitMQ message broker to do that. It can be achieved using the `@EnableZipkinStreamServer` annotations which configure Spring Sleuth to use streams for store Spans.

Let's create our `docker-compose-min.yaml` to bootstrap our RabbitMQ and MySQL containers. The file should look like this:

```
version: '3'  
services:  
  
  rabbitmq:  
    hostname: rabbitmq
```

```
image: rabbitmq:3.7.0-management-alpine
ports:
  - "5672:5672"
  - "15672:15672"
networks:
  - airline

mysql:
  hostname: mysql
  image: mysql:5.7.21
  ports:
    - "3306:3306"
  environment:
    - MYSQL_ROOT_PASSWORD=root
    - MYSQL_DATABASE=zipkin
  networks:
    - airline

mongo:
  hostname: mongo
  image: mongo
  ports:
    - "27017:27017"
  networks:
    - airline

redis:
  hostname: redis
  image: redis:3.2-alpine
  ports:
    - "6379:6379"
  networks:
    - airline

networks:
  airline:
    driver: bridge
```



The docker-compose-min.yaml file can be found at [GitHub](#), there is a MongoDB and Redis – they will be used in the next chapter.

There is nothing special here. We have declared two containers—RabbitMQ and MySQL—and exposed the ports on the host machine. Also, we have created the airline network; we will use this network to attach our infrastructure microservices.

Now, we can create our Zipkin server, which we will do in the next section.

Creating the Spring Cloud Zipkin server

We will create our Zipkin panel structure in Spring Initializr, and then we need to follow the instructions:

The screenshot shows the Spring Initializr web interface. At the top, it says "SPRING INITIALIZR bootstrap your application now". Below that, there's a search bar with "Generate a [Maven Project] with [Java] and Spring Boot [1.5.9]". The "Project Metadata" section has "Artifact coordinates" fields for "Group" (set to "springfive.airline") and "Artifact" (set to "zipkin-server"). The "Dependencies" section shows "Selected Dependencies" including "Web", "Config Client", "Sleuth Stream", "Stream Binder Rabbit", "Zipkin Stream", "Sleuth", and "Zipkin UI". A "Generate Project" button is at the bottom. A note at the bottom left says "Don't know what to look for? Want more options? [Switch to the full version.](#)"

Awesome – take a look at the **Selected Dependencies** section, all of them are required. Pay attention to the Spring Boot version. We choose 1.5.9, because there is no support for Zipkin server in Spring Boot 2. It is not a problem because we do not need specific features from Spring Boot 2.

Click on the **Generate Project** button and wait for the download to finish. Afterwards, open the project in IDE.

In order to enable service discovery and store Spans on a database, we need to put the following dependencies in our `pom.xml`:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>

<dependency>
```

```
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>

<dependency>
<groupId>mysql</groupId>
<artifactId>mysql-connector-java</artifactId>
<version>6.0.6</version>
</dependency>
```

The first dependency is for the service discovery client and the others are to JDBC connections to MySQL. It makes our project dependencies fully configured.

Let's create our main class to start our Zipkin server. The class is pretty standard but with some new annotations:

```
package springfive.airline;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.EnableEurekaClient;
import org.springframework.cloud.stream.zipkin.EnableZipkinStreamServer;

@SpringBootApplication
@EnableZipkinStreamServer
@EnableEurekaClient
public class ZipkinServerApplication {

    public static void main(String[] args) {
        SpringApplication.run(ZipkinServerApplication.class, args);
    }

}
```

The `@EnableEurekaClient` annotation enables the application to connect to the Eureka server. The new annotation, `@EnableZipkinStreamServer`, instructs the framework to connect with the configured broker to receive the Spans. Remember, it can be done using the Spring Cloud Stream Binder.

Configuring bootstrap.yaml and application.yaml

In the section, we created our main class. Before we run it, we should create our two configuration files. The `bootstrap.yaml` inside the `src/main/resources` directory and the `application.yaml` on our GitHub repository. They will be downloaded via Config Server and provided by the Zipkin server project.

Let's start with `bootstrap.yaml`:

```
spring:
  application:
    name: zipkin
  cloud:
    config:
      uri: http://localhost:5000
      label: master
```

Nothing special, we have configured our Config Server address.

Let's jump to our `application.yaml`:

```
server:
  port: 9999

spring:
  rabbitmq:
    port: 5672
    host: localhost
  datasource:
    schema: classpath:/mysql.sql
    url:
      jdbc:mysql://${MYSQL_HOST:localhost}/zipkin?autoReconnect=true
      driver-class-name: com.mysql.cj.jdbc.Driver
      username: root
      password: root
      initialize: true
      continue-on-error: true
  sleuth:
    enabled: false

zipkin:
  storage:
    type: mysql

logging:
```

```
level:  
    ROOT: INFO  
  
eureka:  
    client:  
        serviceUrl:  
            defaultZone: http://localhost:8761/eureka/
```

There are some interesting things here. In the `spring.rabbitmq` node, we have configured our RabbitMQ broker connection. It will be used to receive Spans. In the `spring.datasource`, we have configured the MySQL connection. The Zipkin server will use it to store data. Also, we have configured how to execute the DDL script to create the `zipkin` database.

The `spring.sleuth` node was configured to not produce any Span because it is a server, not a client application, and we will not perform a trace on the Zipkin server.

The `zipkin` node had been used to configure the Zipkin server storage type, MySQL, in our case.

Let's run it!!!

Running the Zipkin server

We have configured the Zipkin server properly, so now we will be able to run it properly.

We can run the main class `ZipkinServerApplication`. We can use the IDE or Java command line, after running the following output:

```
2018-01-16 14:37:47.396 INFO [zipkin,,] 3715 — [main] o.s.i.endpoint.EventDrivenConsumer : Adding {message-handler:inbound.sleuth.sleuth} as a subscriber to the 'bridge.sleuth'  
2018-01-16 14:37:47.396 INFO [zipkin,,] 3715 — [main] o.s.i.endpoint.EventDrivenConsumer : started inbound.sleuth.sleuth  
2018-01-16 14:37:47.397 INFO [zipkin,,] 3715 — [main] o.s.c.support.DefaultLifecycleProcessor : Starting beans in phase 2147483647  
2018-01-16 14:37:47.474 INFO [zipkin,,] 3715 — [main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 9999 (http)  
2018-01-16 14:37:47.475 INFO [zipkin,,] 3715 — [main] s.c.n.e.s.EurekaAutoServiceRegistration : Updating port to 9999  
2018-01-16 14:37:47.479 INFO [zipkin,,] 3715 — [main] $airline.ZipkinServerApplication : Started ZipkinServerApplication in 13.782 seconds (JVM running for 14.068)
```

Good job – the Zipkin server is running now. We can take a look at the index page to see what it looks like.

Go to Zipkin page; the page should look like the following screenshot:

The screenshot shows the Zipkin web interface. At the top, there are three tabs: "Zipkin", "Investigate system behavior", "Find a trace", "Dependencies", and a "Go to trace" button. Below the tabs are several search and filter fields: "Start time" set to "01-09-2018 13:06", "End time" set to "01-16-2018 13:06", "Duration (μs) >= [empty]", "Limit" set to "10", and a "Find Traces" button. There is also a dropdown for "Annotations Query" with the placeholder "Annotations Query (e.g. "finagle.timeout", "error", "http.path=/foo/bar/ and cluster=foo and cache.miss")". Below these fields, it says "Showing: 0 of 0" and "Sort: Longest First". A "Services:" section is present, and a "JSON" button is located on the right. A large blue banner at the bottom states "Please select the criteria for your trace lookup."

Also, we can check the RabbitMQ panel to find the queue created by the Zipkin server. Go to the RabbitMQ Queues (<http://localhost:15672/#/queues>) section, the page should look like this:

The screenshot shows the RabbitMQ management interface. At the top, it displays "RabbitMQ 3.7.0 Erlang 20.1.7" and "Refreshed 2018-01-16 15:29:26 Refresh every 5 seconds". It shows "Virtual host: All", "Cluster rabbit@rabbitmq", and "User guest Log out". The navigation bar includes "Overview", "Connections", "Channels", "Exchanges", "Queues" (which is selected), and "Admin". The main area is titled "Queues" and shows "All queues (1)". Below this is a "Pagination" section with "Page 1 of 1" and a "Filter" input. A table follows, with columns: "Overview", "Messages", and "Message rates". The "Overview" column has sub-columns: "Name", "Features", "State", "Ready", "Unacked", "Total", "incoming", "deliver / get", and "ack". One row is shown: "sleuth.sleuth" with State "idle", Ready 0, Unacked 0, Total 0, incoming 0, deliver / get 0, and ack 0. At the bottom, there is a link "Add a new queue".

Looking at the queues, the project has created the `sleuth.sleuth` queue, well done.

The Zipkin server is ready. For now, we will not have any Span, because there is no application sending data to Zipkin. We will do that in the next chapter.

Spring Cloud Gateway

The API Gateway pattern helps us to expose our microservices through a single known entrypoint. Usually, it acts as an entrypoint to external access and redirects the call to internal microservices.

There are many benefits when we adopt the API Gateway in our application. The first one can be recognized easily, it makes the API consumption easy for the clients, which means the clients do not need to know the different microservices endpoints.

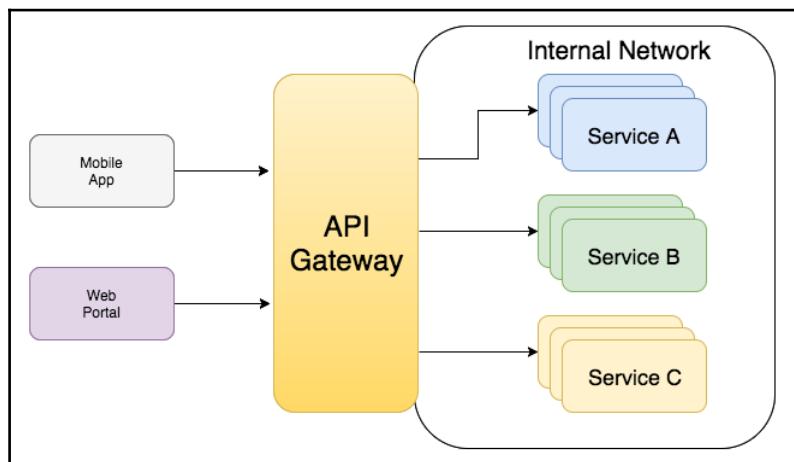
Other benefits are a consequence of the first one. When we have a unique entrypoint, we can address some cross-application concerns such as filtering, authentication, throttling, and rate limit, as well.

It is an essential part when we adopt the microservices architecture.

The Spring Cloud Gateway enables us to have these features in a Spring-managed bean, in a Spring way using Dependency Injection and other features provided by the Spring Framework.

The project was built on the Spring Framework 5, which uses the Project Reactor as a basis. There are some interesting features provided, such as Hystrix Circuit Breaker integration and with the Spring Cloud Discovery client, as well.

Look at the diagram to understand the benefits of the API Gateway:





The full documentation of the API Gateway Pattern can be found at: <http://microservices.io/patterns/apigateway.html>.

Creating the Spring Cloud Gateway project

We will use the Spring Initializr to create our Spring Cloud Gateway project; we will need to add some dependencies manually. Let's go to the [Spring Initializr](#) page and create our project:

The screenshot shows the Spring Initializr interface. At the top, it says "SPRING INITIALIZR bootstrap your application now". Below that, there's a search bar with the placeholder "Generate a [Maven Project] with [Java] and Spring Boot [2.0.0 M7]".

On the left, under "Project Metadata", there are fields for "Group" (set to "springfive.airline") and "Artifact" (set to "gateway").

On the right, under "Dependencies", there's a section titled "Add Spring Boot Starters and dependencies to your application" with a "Search for dependencies" input field containing "Web, Security, JPA, Actuator, Devtools...". Below this, a "Selected Dependencies" section shows three green buttons: "Reactive Web", "Actuator", and "Config Client", each with a close button ("X"). A single green button labeled "Gateway" also has a close button ("X").

At the bottom center is a large green "Generate Project" button. Below it, a small note says "Don't know what to look for? Want more options? [Switch to the full version](#)".

There is a brand new dependency **Gateway**, it enables us to work with Spring Cloud Gateway. Then click on **Generate Project** and wait for the download to complete.

After that, we need to add a missing dependency. The missing dependency is required by the Gateway to interact with the Eureka server; the name of the dependency is `spring-cloud-starter-netflix-eureka-client`. Then, let's add the dependency on our `pom.xml`, we will need to add the following snippet:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

Excellent, our project is configured correctly to work with the Eureka server. In the following section, we will configure the project to work with the Config Server as well.

Creating the Spring Cloud Gateway main class

There is no secret to this part. The Spring Cloud Gateway works in the same way as the common Spring Boot applications. There is a main class which will start the embedded server and starts the whole application.

Our main class should look like this:

```
package springfive.airline.gateway;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.EnableEurekaClient;

@EnableEurekaClient
@SpringBootApplication
public class GatewayApplication {

    public static void main(String[] args) {
        SpringApplication.run(GatewayApplication.class, args);
    }

}
```

As we can see, it is a pretty standard Spring Boot application, configured with `@EnableEurekaClient` to work with the Eureka server as a service discovery implementation.

Configuring the Spring Cloud Gateway project

The primary project structure is ready. We will create the project configurations in this section. To achieve this, we need to carry out the following steps:

- Add a `gateway.yaml` file to GitHub
- Create the `bootstrap.yaml` in the Gateway project

We are using the Spring Cloud Config Server, so it is necessary to create the new file in GitHub because the Config Server will try to find the file on the repository. In our case, we are using GitHub as a repository.

The second task is necessary because the `bootstrap.yaml` file is processed before the application is fully ready to run. Then, during this phase, the application needs to look up the configuration file and to achieve this, the application needs to know the repository, in our case, the Config Server. Remember the address of the Config Server always needs to be placed on the `bootstrap.yaml`.

Let's create our `gateway.yaml` file – the file should look like this:

```
server:  
  port: 8888  
eureka:  
  client:  
    serviceUrl:  
      defaultZone: http://localhost:8761/eureka/  
logging:  
  level: debug
```

The `eureka.client` node in the YAML file is responsible for configuring the Eureka Client configurations. We need to configure our Eureka server address instance. It should be pointed to the correct address.



There are more options for the Eureka Configuration Client properties. The full documentation can be found in <https://github.com/Netflix/eureka/wiki/Configuring-Eureka>; the Netflix team maintains Eureka.

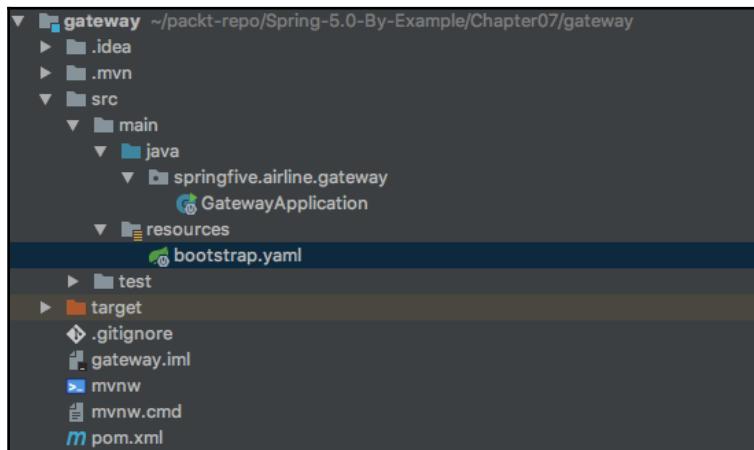
Then, we need to create our `bootstrap.yaml` file on the Gateway project. This file will instruct the Spring Framework to look up the configuration file on the Config Server and then download the required file to finish the application bootstrap. Our file should look like this:

```
spring:  
  application:  
    name: gateway  
  cloud:  
    config:  
      uri: http://localhost:5000  
      label: master
```

Pretty simple. The `application.name` is required to instruct the framework to look up the correct file. Usually, there are many configuration files for different applications and environments as well.

On the `cloud.config` node, we need to put in the Spring Cloud Config Server address, which we configured in the previous sections.

The project final structure should look like this:



Look at the screenshot. There is no `application.yaml` in the classpath. This gives us several advantages; there is no configuration file in classpath projects, which helps us a great deal in managing the microservices configurations.

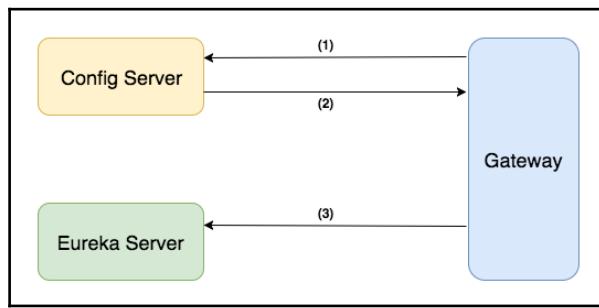
In the next section, we will run it and explain the whole application bootstrap process. Let's do it.

Running the Spring Cloud Gateway

The project is well-configured, so now it is time to run it. We can use the Java command line or IDE. There is no difference either way.

The Config Server and Eureka server need to stay up; it is mandatory that the Gateway project works correctly. Then, we can run the project.

Run the project and look at the logs. We can see some interesting stuff, such as the project connecting to the Config Server and download the configuration and after this, it connects to the Eureka server and self-registers. The following diagram explains the application bootstrap flow:



Let's look at what the different flows are and understand them:

1. The Gateway application requests the configuration file
2. The Config Server serves the config file
3. The Gateway application registers to the Eureka server

Awesome, our Gateway application is connected to our infrastructure services.

Checking the Eureka server

Our Gateway is running. Now, we can check the Eureka server page to confirm this information.

Go to `http://localhost:8761/`, and check the **Instances currently registered with Eureka** section. We should see the Gateway application, as shown in the following screenshot:

The screenshot shows the Spring Eureka dashboard. At the top, it displays "spring Eureka" with a logo, "HOME", and "LAST 1000 SINCE STARTUP". Below this, the "System Status" section shows environment details: Environment is "test" and Data center is "default". To the right, there's a table with system metrics: Current time is "2018-01-07T10:33:09 -0200", Uptime is "00:09", Lease expiration enabled is "true", Renews threshold is "3", and Renews (last min) is "4". The "DS Replicas" section shows a single entry for "localhost". The final section, "Instances currently registered with Eureka", lists a single instance: Application is "GATEWAY", AMIs is "n/a (1)", Availability Zones is "(1)", and Status is "UP (1) - 192.168.100.106:gateway:8888".

Excellent. It worked well. The Gateway application is successfully registered, and it can be looked up via the service discovery. Our Gateway will connect to the Eureka server to get the service available and distribute the requested calls to the correct services.

Well done. Now, we can create our routes in the Gateway. We will do this in the next chapter when we create our airline microservices.

Creating our first route with Spring Cloud Gateway

Our Gateway is running. Before we start the real routes for our Airline application, let's try to use some fake routes to test the Spring Cloud Gateway behaviors. We will use the `https://httpbin.org/` site, which helps us to test some routes.

Let's create a class with the `@Configuration` annotation to provide the routes for the Spring Container. Let's create a package called `springfive.airline.gateway.infra.route`, then create the following class:

```
package springfive.airline.gateway.infra.route;

import java.util.function.Function;
import org.springframework.cloud.gateway.route.RouteLocator;
import org.springframework.cloud.gateway.route.builder.PredicateSpec;
import org.springframework.cloud.gateway.route.builder.RouteLocatorBuilder;
import org.springframework.cloud.gateway.route.builder.RouteLocatorBuilder.Bu
ilder;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class SampleRoute {

    private Function<PredicateSpec, Builder> addCustomHeader =
predicateSpec -> predicateSpec
        .path("/headers")
        .addRequestHeader("Book", "Spring 5.0 By Example")
        .uri("http://httpbin.org:80");

    @Bean
    public RouteLocator sample(RouteLocatorBuilder builder) {
        return builder.routes()
            .route("custom-request-header", addCustomHeader)
            .route("add-query-param", r ->
r.path("/get").addRequestParameter("book", "spring5.0")
                .uri("http://httpbin.org:80"))
            .route("response-headers", (r) -> r.path("/response-headers")
                .addResponseHeader("book", "spring5.0")
                .uri("http://httpbin.org:80"))
            .route("combine-and-change", (r) ->
r.path("/anything").and().header("access-key", "AAA")
                .addResponseHeader("access-key", "BBB")
                .uri("http://httpbin.org:80"))
            .build();
    }
}
```

There are some different types to configure routes; the first one we extracted is the function to a private attribute called `addCustomHeader`, which will be used in the `custom-request-header` route. We will use `curl` to test some routes created previously.

The first one we will test is the `custom-request-header`, the route was configured to route to: `http://httpbin.org:80` and the path will be `/headers`. This service will return the Request Headers sent to the server. Take a look at `addCustomHeader`, we have configured it to add a custom header to the Request. It will be `Book` as the key and `Spring 5.0 By Example`, as the value. Let's call the gateway URL, using curl:

```
curl http://localhost:8888/headers
```

The output should look like this:

```
{  
  "headers": {  
    "Accept": "*/*",  
    "Book": "Spring 5.0 By Example",  
    "Connection": "close",  
    "Host": "httpbin.org",  
    "User-Agent": "curl/7.54.0"  
  }  
}
```

Let's analyze the output. The first thing to look at is we have called the localhost address. The `Host` key in the Request shows `httpbin.org`, it means the Spring Cloud Gateway has changed the address. Awesome, but we expected it. The second one is where we have added the `Book` key, and bingo, there it is in the Request Headers. The Gateway worked as expected, and with a few lines of code, we did some interesting stuff.

Let's do one more test. We will test the `combine-and-change`, this route is configured to answer the `/anything` with the Request Header `access-key: AAA`, so the command line should be:

```
curl -v -H "access-key: AAA" http://localhost:8888/anything
```

As we can see, the `-v` argument makes the call in verbose mode, it is useful for debugging purposes and the `-H` indicates the Request Headers. Let's look at the output:

```
* Trying ::1...
* TCP_NODELAY set
* Connected to localhost (::1) port 8888 (#0)
> GET /anything HTTP/1.1
> Host: localhost:8888
> User-Agent: curl/7.54.0
> Accept: /*
> access-key: AAA
>
< HTTP/1.1 200 OK
< access-key: BBB
< Connection: keep-alive
< Server: meinheld/0.6.1
< Date: Wed, 10 Jan 2018 00:49:29 GMT
< Content-Type: application/json
< Access-Control-Allow-Origin: *
< Access-Control-Allow-Credentials: true
< X-Powered-By: Flask
< X-Processed-Time: 0.00110197067261
< Content-Length: 329
< Via: 1.1 vegur
<
{
  "args": {},
  "data": "",
  "files": {},
  "form": {},
  "headers": {
    "Accept": "/*/*",
    "Access-Key": "AAA",
    "Connection": "close",
    "Host": "httpbin.org",
    "User-Agent": "curl/7.54.0"
  },
}
```

Awesome. If you look at the `access-key` value, the Gateway changed to a requested value `BBB`. Good job guys. There are some endpoints to test, feel free to test as you want.



You can find the httpbin documentation at: <https://httpbin.org/>. There are some interesting other methods to test HTTP.

Putting the infrastructure on Docker

Our infrastructure is ready and it enables us to develop the application. We can create a Docker compose file to spin up the infrastructure services; during the development life cycle, components such as Eureka, Config Server, Trace Server, and API Gateway do not suffer changes because they interact as an infrastructure.

Then, it enables us to create component images and use them in the `docker-compose.yaml` file. Let's list our components:

- Config Server
- Eureka
- Zipkin
- RabbitMQ
- Redis

We know how to create Docker images using the Fabric8 Maven plugin, we have done this several times in the previous chapters – let's do it.

Let's configure one as an example, keep in mind we need to do the same configuration for all projects, Eureka, Gateway, Config Server, and Gateway. The following snippet configures the `docker-maven-plugin` to generate a Docker image:

```
<plugin>
  <groupId>io.fabric8</groupId>
  <artifactId>docker-maven-plugin</artifactId>
  <version>0.21.0</version>
  <configuration>
    <images>
      <image>
        <name>springfivebyexample/${project.build.finalName}</name>
        <build>
          <from>openjdk:latest</from>
          <entryPoint>java -Dspring.profiles.active=docker -jar
/application/${project.build.finalName}.jar</entryPoint>
          <assembly>
            <basedir>/application</basedir>
            <descriptorRef>artifact</descriptorRef>
            <inline>
              <id>assembly</id>
              <files>
                <file>
<source>target/${project.build.finalName}.jar</source>
                </file>
              </files>
            </assembly>
          </build>
        </image>
      </images>
    </configuration>
  </plugin>
```

```
        </inline>
    </assembly>
    <tags>
        <tag>latest</tag>
    </tags>
    <ports>
        <port>8761</port>
    </ports>
</build>
<run>
    <namingStrategy>alias</namingStrategy>
</run>
<alias>${project.build.finalName}</alias>
</image>
</images>
</configuration>
</plugin>
```

It is a pretty simple configuration. A simple Maven plugin with a couple of configurations. Then, after the plugin configuration, we are able to generate the Docker image. The command to generate Docker images is:

```
mvn clean install docker:build
```

It will generate a Docker image for us.

The projects configured can be found on GitHub; there are so many configurations to do as in the previous chapters. We need to configure the `docker-maven-plugin` and generate the Docker images.



Fully configured projects can be found in the chapter seven folder. The GitHub repository is: <https://github.com/PacktPublishing/Spring-5.0-By-Example/tree/master/Chapter07>.

After the images have been created, we are able to create a Docker compose file defining the whole thing. The `docker-compose-infra-full.yaml` file should look like this:

```
version: '3'
services:

  config:
    hostname: config
    image: springfivebyexample/config
    ports:
      - "5000:5000"
```

```
networks:
  - airline
rabitmq:
  hostname: rabbitmq
  image: rabbitmq:3.7.0-management-alpine
  ports:
    - "5672:5672"
    - "15672:15672"
  networks:
    - airline
mysql:
  hostname: mysql
  image: mysql:5.7.21
  ports:
    - "3306:3306"
  environment:
    - MYSQL_ROOT_PASSWORD=root
    - MYSQL_DATABASE=zipkin
  networks:
    - airline
redis:
  hostname: redis
  image: redis:3.2-alpine
  ports:
    - "6379:6379"
  networks:
    - airline

zipkin:
  hostname: zipkin
  image: springfivebyexample/zipkin
  ports:
    - "9999:9999"
  networks:
    - airline
networks:
  airline:
    driver: bridge
```

There are some interesting things to pay attention to here. It is very important that all container instances are attached to the same Docker network called `airline`. Pay attention to the ports exposed by the containers, it is important to enable service discovery features in Docker.

Then, we can execute the instruction to spin up the whole infrastructure; it can be done using the following command:

```
docker-compose -f docker-compose-infra-full.yaml up -d
```

The following output should appear:

```
Creating network "docker_airline" with driver "bridge"
Creating docker_zipkin_1    ... done
Creating docker_rabbitmq_1  ... done
Creating docker_mysql_1     ... done
Creating docker_config_1    ... done
Creating docker_gateway_1   ... done
Creating docker_discovery_1 ... done
```

Also, we can execute the following instruction to check the container's execution:

```
docker-compose -f docker-compose-infra-full.yaml ps
```

It will list the running containers, as shown in the following screenshot:

Name	Command	State	Ports
docker_config_1	/bin/sh -c java -Dspring.p ...	Up	0.0.0.0:5000->5000/tcp
docker_discovery_1	/bin/sh -c java -Dspring.p ...	Up	0.0.0.0:8761->8761/tcp
docker_mysql_1	docker-entrypoint.sh mysqld	Up	0.0.0.0:3306->3306/tcp
docker_rabbitmq_1	docker-entrypoint.sh rabbit ...	Up	15671/tcp, 0.0.0.0:15672->15672/tcp, 25672/tcp, 4369/tcp, 5671/tcp, 0.0.0.0:5672->5672/tcp
docker_redis_1	docker-entrypoint.sh redis ...	Up	0.0.0.0:6379-> 6379/tcp
docker_zipkin_1	/bin/sh -c java -Dspring.p ...	Up	0.0.0.0:9999->9999/tcp

All applications are up and running. Well done.

To remove the containers, we can use:

```
docker-compose -f docker-compose-infra-full.yaml down
```

It will remove the containers from the stack.

Excellent job, our infrastructure is fully operational in Docker containers. It is a base for starting to create our microservices.

Summary

In this chapter, we have built the essential infrastructures services adopting the microservices architectural style.

We have learned how Spring Framework eliminates the infrastructure code from our microservices and enables us to create these services, using a couple of annotations.

We understand how it works under the hood; it is extremely important to debug and troubleshoot when the application gets some errors in the production stage.

Now, we are ready to create scalable, fault tolerant, and responsive systems. We have built the foundations of our system.

In the next chapter, we will start to build our Airline Ticket System, understand how to connect the new microservices with the whole infrastructure, and enable service discovery and other amazing features.

See you there.

8

Putting It All Together

There are some challenges to face when we adopt the microservices architectural style. The first one handles operational complexity; services such as service discovery and load balancer help us to tackle these points. We solved these challenges in the previous chapters and got to know some important tools while doing so.

There are some other important key points to handle in microservices adoption. The effective way to monitor what happens in our microservices environments is to monitor how many times microservices consume other microservices resources, such as HTTP APIs, and how many times they fail. If we have near real-time statistics, it can save the developer days of troubleshooting and error investigations.

In this chapter, we will create some services which help us monitor the Hystrix commands and aggregate the command's statistics in a distributed environment.

Security is an important characteristic in microservices architecture, especially because of the distributed characteristic adopted by the microservices architecture. There are a lot of microservices in our architecture; we cannot share state between services, so the stateless security fits well for our environment.

The OAuth 2.0 protocol specification has this important characteristic: the stateless implementation. Spring Cloud Security provides support for OAuth 2.0.

Finally, we will Dockerize our microservices to use the images in Docker compose files.

In this chapter, we will learn about:

- Implementing the Turbine server to aggregate Hystrix streams
- Configuring the Hystrix Dashboard to use Turbine and input data
- Creating a mail service that will integrate an email API

- Understanding Spring Cloud Security
- Dockerizing our microservices

The airline Bookings microservice

The airline Bookings microservice is a standard Spring Boot Application. There are some interactions with other services, such as the flights microservice.

These interactions were created using Hystrix to bring some desired behaviors, such as fault-tolerance and resilience, to the airline Bookings microservice.

There are some business rules on this service, they are is not important to the learning context now, so we will skip the project creation and execution sections.



The full source code can be found at GitHub (<https://github.com/PacktPublishing/Spring-5.0-By-Example/tree/master/Chapter09/airline-booking>); let's check it out and take a look at some code.

The airline Payments microservice

The Airline Payments is a microservice that gives payments confirmation for our Airline Ticket System. For learning purposes, we will jump this project because there are some business rules, nothing important in the Spring Framework context.

We can find the full source code on GitHub (<https://github.com/PacktPublishing/Spring-5.0-By-Example/tree/master/Chapter09/airline-payments>).

Learning about the Turbine server

There are some integrations in our microservices group; the Bookings microservice calls the Fares microservice and the Passengers microservice, these integrations are done using Hystrix to make it more resilient and fault tolerant.

However, in the microservices world, there are several instances of service. This will require us to aggregate the Hystrix command metrics by instance. Managing the instances panel by panel is not a good idea. The Turbine server helps developers in this context.

By default, Turbine pulls metrics from servers run by Hystrix, but it is not recommended for cloud environments because it can consume high values of network bandwidth and it will increase the traffic costs. We will use Spring Cloud Stream RabbitMQ to push metrics to Turbine via the **Advanced Message Queuing Protocol (AMQP)**. Due to this, we will need to configure the RabbitMQ connections and put two more dependencies in our microservices, the dependencies are:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-netflix-hystrix-stream</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
```

These dependencies will enable the metrics to be sent to the Turbine server via the AMQP protocol.

The Turbine stream, by default, uses the port 8989 . We will configure it to run at 8010, and we can use the `turbine.stream.port` property in the `application.yaml` to customize it.

The Turbine stream will be a Hystrix Dashboard data input to show the commands metrics.



The full source code can be found on GitHub (<https://github.com/PacktPublishing/Spring-5.0-By-Example/tree/master/Chapter09/turbine>).

There are many configurations to customize the Turbine server. They make the server extremely adaptable for different use cases.



We can find the Turbine documentation in the *Spring Cloud Turbine* section (https://cloud.spring.io/spring-cloud-netflix/single/spring-cloud-netflix.html#_turbine). There is a great deal of information, especially if you need to customize some configurations.

Creating the Turbine server microservice

Let's create our Turbine server. We will create a standard Spring Boot Application with a couple of annotations to enable Turbine stream and discovery client, as well.

The main class should be:

```
package springfive.airline.turbine;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.EnableEurekaClient;
import org.springframework.cloud.netflix.turbine.stream.EnableTurbineStream;

@EnableEurekaClient
@EnableTurbineStream
@SpringBootApplication
public class AirlineTurbineApplication {

    public static void main(String[] args) {
        SpringApplication.run(AirlineTurbineApplication.class, args);
    }

}
```

As we can see, `@EnableTurbineStream` will enable us to push Hystrix commands metrics via the RabbitMQ message broker, which is enough for us.

The Turbine server application.yaml file can be found on GitHub (<https://github.com/PacktPublishing/Spring-5.0-By-Example/blob/master/config-files/turbine.yaml>). There are a couple of configurations, such as discovery client and Turbine server configuration.

We can run the application, via the command line or IDE. Run it!

Make some calls to the flights microservice. The Create Flight API will call the planes microservice, which uses the Hystrix command, and will trigger some Hystrix command calls.



We can use the Postman Collection located at GitHub (https://github.com/PacktPublishing/Spring-5.0-By-Example/blob/master/postman/flights.postman_collection). This collection has a Create Flight request, which will call the planes microservices to get plane details. It is enough to collect metrics.

Now, we can test whether our Turbine server is running correctly. Go to the Turbine stream endpoint and then the JSON data with metrics should be displayed like this:

```

event: message
data:
{"rollingCountFallbackFailure":0,"rollingCountFallbackSuccess":0,"propertyValue_circuitBreakerRequestVolumeThreshold":20,"propertyValue_circuitBreakerForceOpen":false,"propertyValue_metricsRollingStatisticalWindowInMilliseconds":10000,"latencyTotal_mean":0,"type":"HystrixCommand","rollingCountResponsesFromCache":0,"typeAndName":{"typeAndName=>HystrixCommand.edge.flights","rollingCountTimeout":0,"propertyValue_executionIsolationStrategy":"SEMAPHORE","instanceId":"edge:8888","rollingCountFailure":0,"rollingCountExceptionsThrown":0,"latencyExecute_mean":0,"isCircuitBreakerOpen":false,"errorCount":0,"group":"RibbonCommand","rollingCountSemaphoreRejected":0,"latencyTotal":{ "0":0,"25":0,"50":0,"75":0,"90":0,"95":0,"99":0,"99.5":0,"100":0}, "requestCount":0,"rollingCountCollapsedRequests":0,"rollingCountShortCircuited":0,"latencyExecute":{ "0":0,"25":0,"50":0,"75":0,"90":0,"95":0,"99":0,"99.5":0,"100":0}, "propertyValue_circuitBreakerSleepWindowInMilliseconds":5000,"currentConcurrentExecutionCount":0,"propertyValue_executionIsolationSemaphoreMaxConcurrentRequests":100,"errorPercentage":0,"rollingCountThreadPoolRejected":0,"propertyValue_circuitBreakerEnabled":true,"propertyValue_executionIsolationThreadInterruptOnTimeout":true,"propertyValue_requestCacheEnabled":true,"rollingCountFallbackRejection":0,"propertyValue_requestLogEnabled":true,"rollingCountSuccess":0,"propertyValue_fallbackIsolationSemaphoreMaxConcurrentRequests":10,"InstanceKey": "edge:8888","propertyValue_circuitBreakerErrorThresholdPercentage":50,"propertyValue_circuitBreakerForceClosed":false,"name": "edge.flights","reportingHosts":1,"propertyValue_executionIsolationThreadPoolKeyOverride": "null","propertyValue_executionIsolationThreadTimeoutInMilliseconds":2000}

```

There are some Hystrix commands information, but as we can see, this information needs to be organized to make it useful for us. Turbine uses the **Server-Sent Events (SSE)** technology, which was introduced in Chapter 6, *Playing with Server-Sent Events*.

In the next section, we will introduce the Hystrix Dashboard. It will help us to organize and make this information useful for us.

Let's jump to the next section.

Hystrix Dashboard

The Hystrix Dashboard will help us to organize the Turbine stream information. As we saw in the previous section, the Turbine server sends information via SSE. It is done using JSON objects.

The Hystrix stream provides a dashboard for us. Let's create our Hystrix Dashboard microservice. The application is a standard Spring Boot Application annotated with `@EnableHystrixDashboard`. Let's add the dependency to enable it:

```

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-hystrix-

```

```
dashboard</artifactId>
</dependency>
```

Good, now we can create the main class for our application. The main class should look like this:

```
package springfive.airline.hystrix.ui;

import org.springframework.boot.SpringApplication;
import
org.springframework.boot.autoconfigure.SpringBootApplication;
import
org.springframework.cloud.netflix.eureka.EnableEurekaClient;
import
org.springframework.cloud.netflix.hystrix.dashboard.EnableHystrixDashboard;

@EnableEurekaClient
@SpringBootApplication
@EnableHystrixDashboard
public class HystrixApplication {

    public static void main(String[] args) {
        SpringApplication.run(HystrixApplication.class, args);
    }
}
```



The full source code can be found at GitHub: <https://github.com/PacktPublishing/Spring-5.0-By-Example/tree/master/Chapter09/hystrix-ui>.

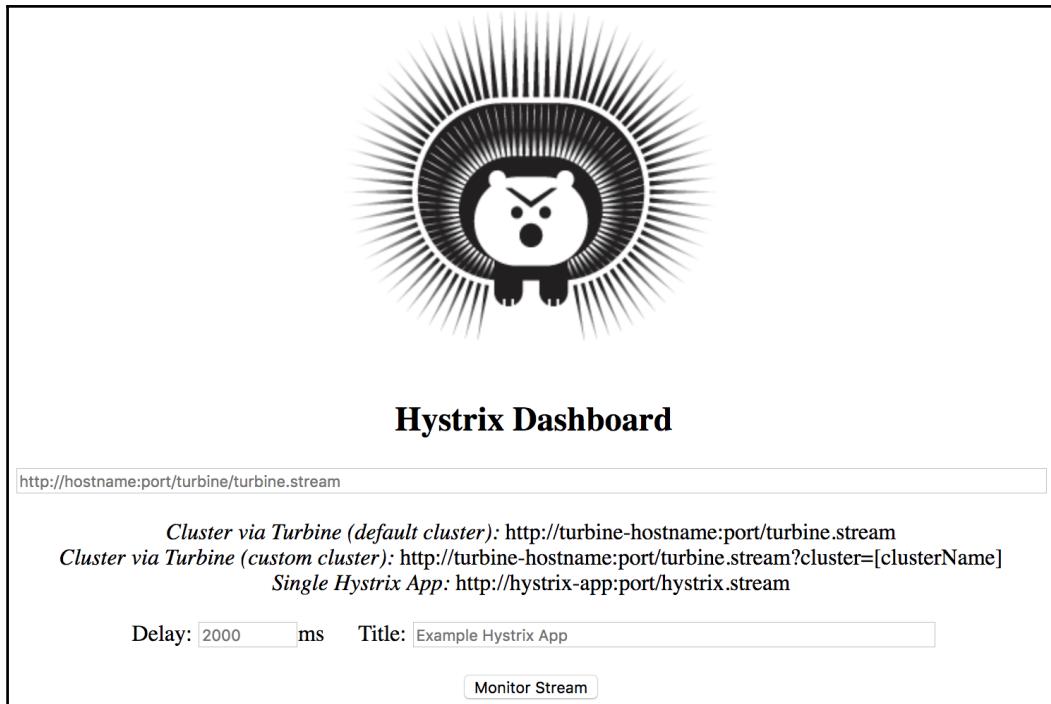
As we can see, this is a pretty standard Spring Boot Application annotated with `@EnableHystrixDashboard`. It will provide the Hystrix Dashboard for us.

Now, we can run the application via IDE or the Java command line. Run it!



The Hystrix Dashboard can be accessed using the following URL :
<http://localhost:50010/hystrix>.

Then, go to the **Hystrix Dashboard** main page. The following page should be displayed:



Awesome – our **Hystrix Dashboard** is up and running. On this page, we can point to `hystrix.stream` or `turbine.stream` to consume and show the commands' metrics.

Keep this application running, we will use it later in this chapter.

Awesome job, guys, let's move to the next section.

Creating the Mail microservice

Now, we will create our `Mail` microservice. The name is self-explanatory, this component will be responsible for sending emails. We will not configure an **SMTP (Simple Mail Transfer Protocol)** server, we will use SendGrid.

SendGrid is an **SaaS (Software as a Service)** service for emails, we will use this service to send emails to our Airline Ticket System. There are some triggers to send email, for example, when the user creates a booking and when the payment is accepted.

Our Mail microservice will listen to a queue. Then the integration will be done using the message broker. We choose this strategy because we do not need the feature that enables us to answer synchronously. Another essential characteristic is the retry policy when the communication is broken. This behavior can be done easily using the message strategy.

We are using RabbitMQ as a message broker. For this project, we will use RabbitMQ Reactor, which is a reactive implementation of RabbitMQ Java client.

Creating the SendGrid account

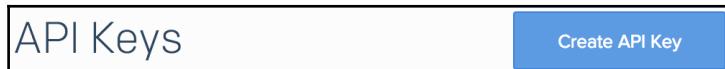
Before we start to code, we need to create a SendGrid account. We will use the trial account which is enough for our tests. Go to the SendGrid portal (<https://sendgrid.com/>) and click on the **Try for Free** button.

Fill in the required information and click on the **Create Account** button.

In the main page, on the left side, click on **Settings**, then go to the **API Key** section, follow the image shown here:



Then, we can click on the **Create API Key** button at the top-right corner. The page should look like this:



Fill in the **API Key** information and choose **Full Access**. After that the **API Key** will appear on your screen. Take a note of it in a safe place, as we will use it as an environment variable soon.

Good job, our SendGrid account is ready to use, now we can code our Mail microservice.

Let's do it in the next section.

Creating the Mail microservice project

As we did in Chapter 8, *Circuit Breakers and Security*, we will take a look at essential project parts. We will be using Spring Initializr, as we have several times in the previous chapters.



The full source code can be found at GitHub (<https://github.com/PacktPublishing/Spring-5.0-By-Example/tree/master/Chapter09/mail-service>).

Adding RabbitMQ dependencies

Let's add the RabbitMQ required dependencies. The following dependencies should be added:

```
<dependency>
    <groupId>io.projectreactor.rabbitmq</groupId>
    <artifactId>reactor-rabbitmq</artifactId>
    <version>1.0.0.M1</version>
</dependency>

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
```

The first one is about the reactive implementation for RabbitMQ and the second one is the starter AMQP, which will set up some configurations automatically.

Configuring some RabbitMQ stuff

We want to configure some RabbitMQ exchanges, queues, and bindings. It can be done using the RabbitMQ client library. We will configure our required infrastructure for the Mail microservice.

Our configuration class should look like this:

```
package springfive.airline.mailservice.infra.rabbitmq;

// imports are omitted

@Configuration
public class RabbitMQConfiguration {

    private final String pass;

    private final String user;

    private final String host;

    private final Integer port;

    private final String mailQueue;

    public RabbitMQConfiguration(@Value("${spring.rabbitmq.password}")
String pass,
        @Value("${spring.rabbitmq.username}") String user,
        @Value("${spring.rabbitmq.host}") String host,
        @Value("${spring.rabbitmq.port}") Integer port,
        @Value("${mail.queue}") String mailQueue) {
        this.pass = pass;
        this.user = user;
        this.host = host;
        this.port = port;
        this.mailQueue = mailQueue;
    }

    @Bean("springConnectionFactory")
    public ConnectionFactory connectionFactory() {
        CachingConnectionFactory factory = new CachingConnectionFactory();
        factory.setUsername(this.user);
        factory.setPassword(this.pass);
    }
}
```

```
        factory.setHost(this.host);
        factory.setPort(this.port);
        return factory;
    }

    @Bean
    public AmqpAdmin amqpAdmin(@Qualifier("springConnectionFactory")
ConnectionFactory connectionFactory) {
    return new RabbitAdmin(connectionFactory);
}

    @Bean
    public TopicExchange emailExchange() {
        return new TopicExchange("email", true, false);
    }

    @Bean
    public Queue mailQueue() {
        return new Queue(this.mailQueue, true, false, false);
    }

    @Bean
    public Binding mailExchangeBinding(Queue mailQueue) {
        return
BindingBuilder.bind(mailQueue).to(emailExchange()).with("**");
    }

    @Bean
    public Receiver receiver() {
        val options = new ReceiverOptions();
        com.rabbitmq.client.ConnectionFactory connectionFactory = new
com.rabbitmq.client.ConnectionFactory();
        connectionFactory.setUsername(this.user);
        connectionFactory.setPassword(this.pass);
        connectionFactory.setPort(this.port);
        connectionFactory.setHost(this.host);
        options.connectionFactory(connectionFactory);
        return ReactorRabbitMq.createReceiver(options);
    }

}
```

There is interesting stuff here, but all of it is about infrastructure in RabbitMQ. It is important because when our application is in bootstrapping time, it means our application is preparing to run. This code will be executed and create the necessary queues, exchanges, and bindings. Some configurations are provided by the application.yaml file, look at the constructor.

Modeling a Mail message

Our Mail service is abstract and can be used for different purposes, so we will create a simple class to represent a mail message in our system. Our Mail class should look like this:

```
package springfive.airline.mailservice.domain;

import lombok.Data;

@Data
public class Mail {

    String from;

    String to;

    String subject;

    String message;

}
```

Easy, this class represents an abstract message on our system.

The MailSender class

As we can expect, we will integrate with the SendGrid services through the REST APIs. In our case, we will use the reactive WebClient provided by Spring WebFlux.

Now, we will use the SendGrid API Key created in the previous section. Our MailSender class should look like this:

```
package springfive.airline.mailservice.domain.service;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.http.HttpStatus;
import org.springframework.http.ReactiveHttpOutputMessage;
import org.springframework.stereotype.Service;
import org.springframework.web.reactive.function.BodyInserter;
import org.springframework.web.reactive.function.BodyInserters;
import org.springframework.web.reactive.function.client.WebClient;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;
import springfive.airline.mailservice.domain.Mail;
import
```

```
springfive.airline.mailservice.domain.service.data.SendgridMail;

@Service
public class MailSender {

    private final String apiKey;

    private final String url;

    private final WebClient webClient;

    public MailSender(@Value("${sendgrid.apikey}") String apiKey,
                      @Value("${sendgrid.url}") String url,
                      WebClient webClient) {
        this.apiKey = apiKey;
        this.webClient = webClient;
        this.url = url;
    }

    public Flux<Void> send(Mail mail) {
        final BodyInserter<SendgridMail, ReactiveHttpOutputMessage> body =
        BodyInserter.of
            .fromObject(SendgridMail.builder().content(mail.getMessage()).from(mail
                .getFrom()).to(mail.getTo()).subject(mail.getSubject()).build())
            .return this.webClient.mutate().baseUrl(this.url).build().post()
            .uri("/v3/mail/send")
            .body(body)
            .header("Authorization", "Bearer " + this.apiKey)
            .header("Content-Type", "application/json")
            .retrieve()
            .onStatus(HttpStatus::is4xxClientError, clientResponse ->
                Mono.error(new RuntimeException("Error on send email")))
            .bodyToFlux(Void.class);
    }

}
```

We received the configurations in the constructor, that is, the `sendgrid.apikey` and `sendgrid.url`. They will be configured soon. In the `send()` method, there are some interesting constructions. Look at `BodyInserter.of.fromObject()`: it allows us to send a JSON object in the HTTP body. In our case, we will create a SendGrid mail object.

In the `onStatus()` function, we can pass a predicate to handle the HTTP errors family. In our case, we are interested in the 4xx error family.

This class will process sending the mail messages, but it is necessary to listen to the RabbitMQ queue, which we will do in the next section.

Creating the RabbitMQ queue listener

Let's create our `MailQueueConsumer` class, which will listen to the RabbitMQ queue. The class should look like this:

```
package springfive.airline.mailservice.domain.service;

import com.fasterxml.jackson.databind.ObjectMapper;
import java.io.IOException;
import javax.annotation.PostConstruct;
import lombok.extern.slf4j.Slf4j;
import lombok.val;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Service;
import reactor.rabbitmq.Receiver;
import springfive.airline.mailservice.domain.Mail;

@Service
@Slf4j
public class MailQueueConsumer {

    private final MailSender mailSender;

    private final String mailQueue;

    private final Receiver receiver;

    private final ObjectMapper mapper;

    public MailQueueConsumer(MailSender mailSender,
        @Value("${mail.queue}") String mailQueue,
        Receiver receiver, ObjectMapper mapper) {
        this.mailSender = mailSender;
        this.mailQueue = mailQueue;
        this.receiver = receiver;
        this.mapper = mapper;
    }

    @PostConstruct
    public void startConsume() {
        this.receiver.consumeAutoAck(this.mailQueue).subscribe(message ->
    {
        try {

```

```
        val mail = this.mapper.readValue(new
String(message.getBody()), Mail.class);
        this.mailSender.send(mail).subscribe(data ->{
            log.info("Mail sent successfully");
        });
    } catch (IOException e) {
        throw new RuntimeException("error on deserialize object");
    }
});
}

}
```

The method annotated with `@PostConstruct` will be invoked after `MailQueueConsumer` is ready, which will mean that the injections are processed. Then `Receiver` will start to process the messages.

Running the Mail microservice

Now, we will run our Mail microservice. Find the `MailServiceApplication` class, the main class of our project. The main class should look like this:

```
package springfive.airline.mailservice;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.EnableEurekaClient;
import org.springframework.cloud.netflix.hystrix.EnableHystrix;
import org.springframework.cloud.netflix.zuul.EnableZuulProxy;

@EnableHystrix
@EnableZuulProxy
@EnableEurekaClient
@SpringBootApplication
public class MailServiceApplication {

    public static void main(String[] args) {
        SpringApplication.run(MailServiceApplication.class, args);
    }
}
```

It is a standard Spring Boot Application.

We can run the application in IDE or via the Java command line.

Run it!



We need to pass `SENDGRID_APIKEY` and `SENDGRID_URL` as environment variables. If you are running the application with the Java command line, the `-D` option allows us to pass environment variables. If you are using the IDE, you can configure in the [Run/Debug Configurations](#).

Creating the Authentication microservice

We want to secure our microservices. Security is essential for microservices applications, especially because of the distributed characteristics.

On the microservices architectural style, usually, there is a service that will act as an authentication service. It means this service will authenticate the requests in our microservices group.

Spring Cloud Security provides a declarative model to help developers enable security on applications. There is support for commons patterns such as OAuth 2.0. Also, Spring Boot Security enables **Single Sign-On (SSO)**.

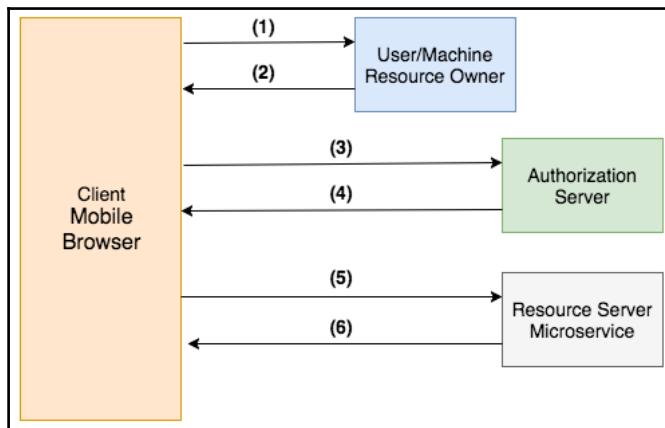
Spring Boot Security also supports relay SSO tokens integrating with Zuul proxy. It means the tokens will be passed to downstream microservices.

For our architecture, we will use the OAuth 2.0 and JWT patterns, both integrate with Zuul proxy.

Before we do so, let's understand the main entities in OAuth 2.0 flow:

- **Protected resource:** This service will apply security rules; the microservices applications, in our case
- **OAuth authorization server:** The authentication server is a service between the application, which can be a frontend or a mobile, and a service that applications want to call
- **Application:** The application that will call the service, the client.
- **Resource Owner:** The user or machine that will authorize the client application to access their account

Let's draw the basic OAuth flow:



We can observe the following in this diagram:

1. The **Client** requests the authorization
2. The **Resource Owner** sends the authorization grant
3. The application client requests the access token from the **Authorization Server**
4. If the authorization grant is valid, the **Authorization Server** will provide the access token
5. The application calls the protected resource and sends the access token
6. If the **Resource Server** recognizes the token, the resource will serve for the application

These are the basics of the OAuth 2.0 authorization flow. We will implement this flow using Spring Cloud Security. Let's do it.

Creating the Auth microservice

As we have been doing in this chapter, we will take a look at the important parts. Let's start with our dependencies. We need to put in the following dependencies:

```
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-oauth2</artifactId>
</dependency>
```

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-core</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-config</artifactId>
</dependency>
```

These dependencies will enable us to use the Spring Cloud Security features. Let's start to code our Authentication microservice.

Configuring the security

Let's start coding our Auth microservice. We will start with the authorization and authentication, as we want to protect all resources in our microservices, then we will configure `WebSecurityConfigurerAdapter`. The class should look like this:

```
package springfive.airline.authservice.infra.security;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.authentication.builders
.AuthenticationManagerBuilder;
import org.springframework.security.config.annotation.method.configuration.EnableGlobalMethodSecurity;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;
import org.springframework.security.crypto.password.PasswordEncoder;
import springfive.airline.authservice.service.CredentialsDetailsService;

@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class SecurityConfig extends WebSecurityConfigurerAdapter {
```

```
private final PasswordEncoder passwordEncoder;

private final CredentialsDetailsService credentialUserDetails;

public SecurityConfig(PasswordEncoder passwordEncoder,
    CredentialsDetailsService credentialUserDetails) {
    this.passwordEncoder = passwordEncoder;
    this.credentialUserDetails = credentialUserDetails;
}

@Override
@.Autowired
protected void configure(AuthenticationManagerBuilder auth) throws
Exception {
    auth.userDetailsService(this.credentialUserDetails).passwordEncoder(th
is.passwordEncoder);
}

@Override
protected void configure(HttpSecurity http) throws Exception {
    http.csrf().disable()
        .authorizeRequests()
        .antMatchers("/login", "/**/register/**").permitAll()
        .anyRequest().authenticated()
        .and()
        .formLogin().permitAll();
}

}
```

There is a lot of stuff here. Let's start with the `@EnableWebSecurity`, this annotation enables Spring Security integrations with Spring MVC.

`@EnableGlobalMethodSecurity` provides AOP interceptors to enable methods security using the annotations. We can use this feature by annotating the methods on a controller, for instance. The basic idea is to wrap the methods call in AOP interceptors and apply security on the methods.

`WebSecurityConfigurerAdapter` enables us to configure the secure endpoints and some stuff about how to authenticate users, which can be done using the `configure(AuthenticationManagerBuilder auth)` method. We have configured our `CredentialsDetailsService` and our `PasswordEncoder` to avoid plain password between application layers. In this case, `CredentialsDetailsService` is the source of our user's data.

In our method, `configure(HttpSecurity http)`, we have configured some HTTP security rules. As we can see, all users can access `/login` and `/**/register/**`. It's about *Sign In* and *Sign Up* features. All other requests need to be authenticated by the Authorization server.

The `CredentialsDetailsService` should look like this:

```
package springfive.airline.authservice.service;

import
org.springframework.security.core.userdetails.UserDetailsService;
import
org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Component;
import springfive.airline.authservice.domain.Credential;
import springfive.airline.authservice.domain.data.CredentialData;
import springfive.airline.authservice.repository.CredentialRepository;

@Component
public class CredentialsDetailsService implements UserDetailsService {

    private final CredentialRepository credentialRepository;

    public CredentialsDetailsService(CredentialRepository
credentialRepository) {
        this.credentialRepository = credentialRepository;
    }

    @Override
    public CredentialData loadUserByUsername(String email) throws
UsernameNotFoundException {
        final Credential credential =
this.credentialRepository.findByEmail(email);
        return
CredentialData.builder().email(credential.getEmail()).password(credential.getPassword()).scopes(credential.getScopes()).build();
    }

}
```

There is nothing special here. We need to override the `loadUserByUsername(String email)` method to provide the user data to Spring Security.

Let's configure our token signer and our token store. We will provide these beans using the `@Configuration` class, as we did in the previous chapters:

```
package springfive.airline.authservice.infra.oauth;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import
org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import
org.springframework.security.oauth2.provider.token.store.JwtAccessTokenConverter;
import
org.springframework.security.oauth2.provider.token.store.JwtTokenStore
;

@Configuration
public class OAuthTokenProducer {

    @Value("${config.oauth2.privateKey}")
    private String privateKey;

    @Value("${config.oauth2.publicKey}")
    private String publicKey;

    @Bean
    public JwtTokenStore tokenStore(JwtAccessTokenConverter
tokenEnhancer) {
        return new JwtTokenStore(tokenEnhancer);
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    @Bean
    public JwtAccessTokenConverter tokenEnhancer() {
        JwtAccessTokenConverter converter = new
JwtAccessTokenConverter();
        converter.setSigningKey(privateKey);
        converter.setVerifierKey(publicKey);
        return converter;
    }

}
```

We have configured our private and public keys in the `application.yaml` file. Optionally, we can read the `jks` files from the classpath as well. Then, we provided our token signer or token enhancer using the `JwtAccessTokenConverter` class, where we have used the private and public key.

In our token store, Spring Security Framework will use this object to read data from tokens, then set up the `JwtAccessTokenConverter` on the `JwtTokenStore` instance.

Finally, we have provided the password encoder class using the `BCryptPasswordEncoder` class.

Our last class is the Authorization server configuration. The configuration can be done using the following class:

Look at the `OAuth2AuthServer` class located on GitHub (<https://github.com/PacktPublishing/Spring-5.0-By-Example/blob/master/Chapter09/auth-service/src/main/java/springfive/airline/authservice/infra/oauth/OAuth2AuthServer.java>).

We have used `@EnableAuthorizationServer` to configure the Authorization server mechanism in our Auth microservice. This class works together with `AuthorizationServerConfigurerAdapter` to provide some customizations.

On `configure(AuthorizationServerSecurityConfigurer oauthServer)`, we have configured the security for token endpoints.

At `configure(AuthorizationServerEndpointsConfigurer endpoints)`, we have configured the endpoints of the token service such as, `/oauth/token` and `/oauth/authorize`.

Finally, on `configure(ClientDetailsServiceConfigurer clients)`, we have configured the client's ID and secrets. We used in-memory data, but we can use JDBC implementations as well.

The Auth microservice main class should be:

```
package springfive.airline.authservice;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.EnableEurekaClient;
import org.springframework.cloud.netflix.zuul.EnableZuulProxy;

@EnableZuulProxy
@EnableEurekaClient
@SpringBootApplication
public class AuthServiceApplication {

    public static void main(String[] args) {
        SpringApplication.run(AuthServiceApplication.class, args);
    }

}
```

Here, we have created a standard Spring Boot Application with service discovery and Zuul proxy enabled.

Testing the Auth microservice

As we can see, the Auth microservice is ready for testing. Our microservice is listening to port 7777, which we configured using the `application.yaml` file on GitHub.

Client credentials flow

Let's start with the client credentials flow.

Our application needs to be up on port 7777, then we can use the following command line to get the token:

```
curl -s
442cf4015509eda9c03e5ca3aceef752:4f7ec648a48b9d3fa239b497f7b6b4d801969
7bd@localhost:7777/oauth/token -d grant_type=client_credentials -d
scope=trust | jq .
```

As we can see, this *client ID* and *client secret* are from the planes microservice. We did this configuration at the OAuth2AuthServer class. Let's remember the exact point:

```
....  
@Override  
public void configure(ClientDetailsServiceConfigurer clients) throws  
Exception {  
    clients  
        .inMemory()  
        .withClient("ecommerce") // ecommerce microservice  
        .secret("9ecc8459ea5f39f9da55cb4d71a70b5d1e0f0b80")  
        .authorizedGrantTypes("authorization_code", "refresh_token",  
"implicit",  
            "client_credentials")  
        .authorities("maintainer", "owner", "user")  
        .scopes("read", "write")  
        .accessTokenValiditySeconds(THREE_HOURS)  
        .and()  
        .withClient("442cf4015509eda9c03e5ca3aceef752") // planes  
microservice  
        .secret("4f7ec648a48b9d3fa239b497f7b6b4d8019697bd")  
        .authorizedGrantTypes("authorization_code", "refresh_token",  
"implicit",  
            "client_credentials")  
        .authorities("operator")  
        .scopes("trust")  
        .accessTokenValiditySeconds(ONE_DAY)  
    ....
```

After you call the preceding command, the result should be:

```
{  
    "access_token": "eyJhbGciOiJSUzI1NiTsInRSccI6IkpXVCJ9.eyJzY29wZSI6IyJ0cnVzdC1dLC1eHA10jE1MtzcNTkdMjIsImF1dGvcm10dWzIpbIm9wZXJhdGyI1QsImp0aS16IjM0N2YwYTg5LTooyODmtNDc3ZC05YnYvLTZHMtNY1ThhYW02Yy1stNsoWudP9zC16IjQ9MmNmNDANtUwOWkYt1JMN1NWmM2FjZWVmNzilyIn0.UsmLH1ETs3Urxx_IMoif7foLqjkSX14htzausdi9D2GsVnxJn21YfrwT7341QG50rHQeSGGZA8c9SM0yITYvoYSytBqSiFl0dHJQglLnNBDOF_I_eSbpsStqay7T_EP-SkoyGA7NkdlqAz1n1TmqN1Je-Mb3Iy8-P1ATsn9xSE",  
    "token_type": "bearer",  
    "expires_in": 86399,  
    "scope": "trust",  
    "jti": "347f0089-8283-477d-9bf2-6a13ba8aaedfc"  
}
```

As we can see, the token was obtained with success. Well done, our client credentials flow was configured successfully. Let's move to the implicit flow, which will be covered in the next section.

Implicit grant flow

In this section, we will take a look at how to authenticate in our Auth microservice using the implicit flow.

Before we test our flow, let's create a user to enable authentication in the Auth microservice. The following command will create a user in the Auth service:

```
curl -H "Content-Type: application/json" -X POST -d '{"name": "John Doe", "email": "john@doe.com", "password": "john"}'  
http://localhost:7777/register
```

As we can see, the email is `john@doe.com` and the password is `john`.

We will use the browser to do this task. Let's go to the following URL:

```
http://localhost:7777/oauth/authorize?client_id=ecommerce&response_type=token&scope=write&state=8777&redirect_uri=https://httpbin.org/anything
```

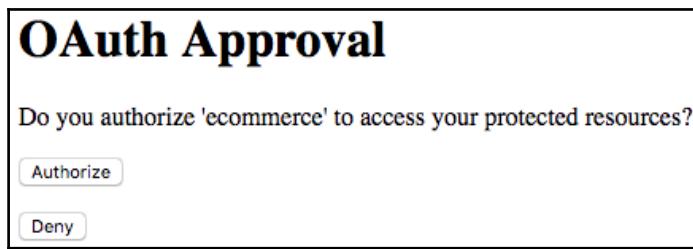
Let's understand the parameters:

The first part is the service address. To use the implicit grant flow, we need the path `/oauth/authorize`. Also we will use `ecommerce` as a client ID because we have configured it previously. `response_type=token` informs the implicit flow, `scope` is the scope as what we want in our case is `write`, `state` is a random variable, and `redirect_uri` is the URI to go after the `oauth` login process.

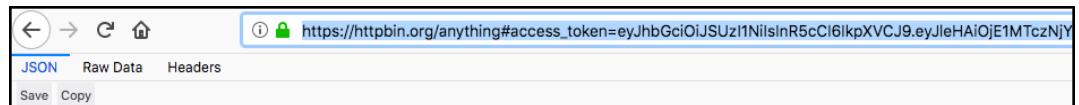
Put the URL in a web browser, and the following page should be displayed:

The screenshot shows a standard web browser interface with a title bar displaying 'localhost:7777/login'. Below the title bar is a header with navigation icons (back, forward, search, etc.). The main content area contains a form titled 'Login with Username and Password'. The form includes two input fields: one for 'User' and one for 'Password', both with placeholder text. Below the password field is a 'Login' button.

After typing the **User** and **Password**, the following page will be displayed to authorize our protected resources:



Click on the **Authorize** button. Then we will see the token in the browser URL like this:



The full token can be viewed if we copy the browser URL.

Awesome job, guys, our Auth microservice is fully operational.

In the next sections, we will configure the Auth microservice to protect Zuul proxy downstream microservices, such as the planes microservices. Let's jump to the next section.

Protecting the microservices with OAuth 2.0

Now we will configure OAuth 2.0 to protect our microservices; in our case, our microservices are the resource servers. Let's start with the planes microservices. We will add the new dependency and configure the private and public keys. Also, we will configure our `JwtTokenStore`.

Let's do it.

Adding the security dependency

To add the newly required dependency, we will change the `pom.xml` of the planes microservice. We will add the following dependency:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-oauth2</artifactId>
</dependency>
```

A piece of cake – our required dependency is configured properly.

In the next section, we will configure the `application.yaml` file.

Configuring the application.yaml file

To configure our private and public keys, we will use the `application.yaml` file. We did this configuration in the Auth microservice. The configuration is pretty easy. We need to add the following snippet:

```
config:
  oauth2:
    privateKey: |
      -----BEGIN RSA PRIVATE KEY-----
MIICXQIBAAKBgQDNQZKqt1O/+2b4ZdhqGJzGBD1tb5PZmBz1ALN2YLvt341pH6i5
m01V9cx5Ty1LM70fKfnIoYUP4KCE33dPnC7LkUwE/myh1zM6m8cbL5cYFPyP09t
hbVxzJkjHWqyvQih/qOOjliomKbM9pxG8Z1dB26hL9dSAzuA8xEejlPmQIDAQAB
AoGAImnYGU3ApPOVtBf/TOqlfne+2SZX96eVU06myDY3zA4rO3DfbR7CzCLE6qPn
yDAIiW0UQB0oBDdWOnOqz5YaePZu/yrLyj6KM6Q2e9ywRdtDh3ywrSfGpjdsVvvo
aeL1WesBWsgWv1vFKkves7ILFLUxKwyCRC2Lgh7aI9GGZfECQQD84m98Yrehhin3
fZuRaBNiu348Ci7ZFZmrvyxAIxRv4jbjpACW0RM2BvF5oYM2gOJqIfBOVjmPwUro
bYEfcHRvAkEAz8jsfmxsZVwh3Y/Y47BzhKIC5FLaads541jnJVWfrPirljyCy1n4
sg3WQH2IEyap3WTP84+csCtsfNfyK7fQdwJBAJNRyobY74cupJYkW5OK4OkXKQQL
Hp2iosJV/Y5jpQeC3JO/gARcSmfIBbbI66q9zKjtmpPYUXI4tc3PtUEY8QscQQCc
xySyC0sKe6bNzyC+Q8AVvkxiTKwiI5idEr8duhJd589H72Zc2wkMB+a2CEGo+Y5H
```

```
jy5cvuph/pG/7Qw7sljnAkAy/feClt1mUEiAcWrHRwcQ71AoA0+21yC9VkqPNrn3  
w70Eg8gBqPjR1XBNb00QieNegGSkXOoU6gFschR22Dzy  
-----END RSA PRIVATE KEY-----  
publicKey: |  
-----BEGIN PUBLIC KEY-----  
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDNQZKqTlO/+2b4ZdhqGJzGBD1t  
b5PZmBz1ALN2YLvt341pH6i5mO1V9cX5Ty1LM70fKfnIoYUP4KCE33dPnC7LkUwE  
/myh1zM6m8cbL5cYFPyP099thbVxzJkjHWqywvQih/qOojliomKbM9pxG8Z1dB26  
hL9dSAZuA8xExjlPmQIDAQAB  
-----END PUBLIC KEY-----
```

Moreover, the user info URI will be done using the following configuration in YAML:

```
oauth2:  
  resource:  
    userInfoUri: http://localhost:7777/credential
```

Awesome – our application is fully configured. Now, we will do the last part: configuring to get the information token.

Let's do that.

Creating the JwtTokenStore Bean

We will create the `JwtTokenStore`, which will be used to get token information. The class should look like this:

```
package springfive.airline.airlineplanes.infra.oauth;  
  
import org.springframework.beans.factory.annotation.Value;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
import  
org.springframework.security.oauth2.provider.token.store.JwtAccessTokenToke  
nConverter;  
import  
org.springframework.security.oauth2.provider.token.store.JwtTokenStore  
;  
  
@Configuration  
public class OAuthTokenConfiguration {  
  
  @Value("${config.oauth2.privateKey}")  
  private String privateKey;  
  
  @Value("${config.oauth2.publicKey}")  
  private String publicKey;
```

```
@Bean  
public JwtTokenStore tokenStore() throws Exception {  
    JwtAccessTokenConverter enhancer = new JwtAccessTokenConverter();  
    enhancer.setSigningKey(privateKey);  
    enhancer.setVerifierKey(publicKey);  
    enhancer.afterPropertiesSet();  
    return new JwtTokenStore(enhancer);  
}  
  
}
```

Awesome – our token signer is configured.

Finally, we will add the following annotation to the main class, which should look like this:

```
package springfive.airline.airlineplanes;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.cloud.netflix.eureka.EnableEurekaClient;  
import org.springframework.cloud.netflix.zuul.EnableZuulProxy;  
import  
org.springframework.security.oauth2.config.annotation.web.configuration.  
n.EnableResourceServer;  
  
{@EnableZuulProxy  
@EnableEurekaClient  
@EnableResourceServer  
@SpringBootApplication  
public class AirlinePlanesApplication {  
  
    public static void main(String[] args) {  
        SpringApplication.run(AirlinePlanesApplication.class, args);  
    }  
  
}
```

It will protect our application, and it will require the access token to access the application endpoints.

Remember, we need to do the same task for all microservices that we want to protect.

Monitoring the microservices

In the microservice architectural style, monitoring is a crucial part. There are a lot of benefits when we adopt this architecture, such as time to market, source maintenance, and an increase of business performance. This is because we can divide the business goals for different teams, and each team will be responsible for some microservices. Another important characteristic is optimization of computational resources, such as cloud computing costs.

As we know, there is no such thing as a free lunch, and this style brings some drawbacks, such as operational complexity. There are a lot of *small services* to monitor. There are potentially hundreds of different service instances.

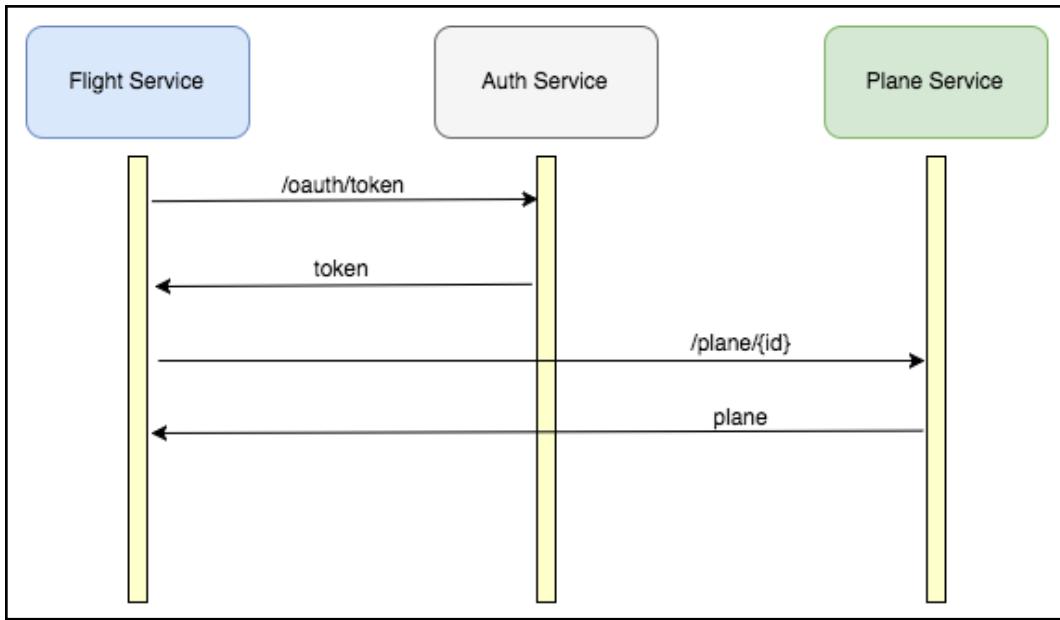
We have implemented some of these services in our infrastructure but until now, we did not have the data to analyze our system health. In this section, we will explore our configured services.

Let's analyze right now!

Collecting metrics with Zipkin

We have configured our Zipkin server in the previous chapter. Now we will use this server to analyze our microservices data. Let's do it.

Make some calls to create a flight. The Create Flight API will call the **Auth Service** and the **Flight Service**. Look at the following diagram:



We will take a look at the `flights` microservice and the `planes` microservice communications. Let's analyze it:

Go to the Zipkin main page, `http://localhost:9999/`, select **flights**, and then click on **Find a trace**. The page should look like this:

The screenshot shows the Zipkin web interface. At the top, there are three tabs: "flights" (selected), "Investigate system behavior", and "Dependencies". Below the tabs are search fields for "Service" (set to "flights"), "Annotations Query" (empty), and "Duration (μs) >= 0". There are also dropdowns for "Start time" (01-27-2018, 13:44), "End time" (02-03-2018, 13:44), "Limit" (10), and a "Sort" dropdown set to "Longest First". A "Find Traces" button and a "JSON" button are also present. The main area displays a list of four traces, each with a timestamp of "26 minutes ago". Each trace entry includes the duration, number of spans, service name, and specific span details (e.g., auth x1 367ms, edge x2 2569ms). The first trace is labeled "flights 68%", the second "flights 100%", the third "flights 99%", and the fourth "flights 100%".

As we can see, there is some data on our Zipkin server. Click on **Span**, which has the **flights** and **planes** tags, then we will take a look at this specific trace, and we will be redirected to another page with specific span data, like this:

The screenshot shows the Zipkin trace details page. At the top, it displays the search filters: Duration: 1.090s, Services: 2, Depth: 1, Total Spans: 1, and a "JSON" button. Below these are buttons for "Expand All", "Collapse All", and "Find...". The main table lists one span with the following data:

Services	Duration	Start Time	End Time	Timestamp
flights x1	218.000ms	436.000ms	654.000ms	872.000ms
planes x1	1.090s : http://5a73a5f1eec7b00a9047b42e			1.090s

On this page, we can see important information, such as the total request time. Then click on the **planes** row, where we will be able to see detailed information, as in the following image:

planes.http://5a73a5f1eec7b00a9047b42e: 1.090s			
AKA: flights,planes			
Date Time	Relative Time	Annotation	Address
2/3/2018, 1:08:28 PM		Client Send	192.168.100.100:50005 (flights)
2/3/2018, 1:08:28 PM	36.000ms	Server Receive	192.168.100.100:50001 (planes)
2/3/2018, 1:08:28 PM	93.000ms	Server Send	192.168.100.100:50001 (planes)
2/3/2018, 1:08:29 PM	1.090s	Client Receive	192.168.100.100:50005 (flights)

Key	Value
http.host	192.168.100.100
http.method	GET
http.path	/5a73a5f1eec7b00a9047b42e
http.url	http://192.168.100.100:50001/5a73a5f1eec7b00a9047b42e
mvc.controller.class	PlaneResource
mvc.controller.method	plane
spring.instance_id	192.168.100.100:flights:50005
spring.instance_id	192.168.100.100:planes:50001

[More Info](#)

Look at the request information. There are some interesting things, such as `mvc.controller.class` and `mvc.controller.method`. These help developers to troubleshoot errors. Also in the first panel, we have the times of the service's interactions. It is very helpful to find microservices network latencies; for example, it makes environment management easier because we have visual tools to understand data better.

Also, the Zipkin server provides others interesting features to find microservices statistics, such as finding requests that have delayed for more than a specific time. It is very helpful for the operations guys.



We can find more information about Spring Cloud Sleuth on the documentation page (<http://cloud.spring.io/spring-cloud-static/spring-cloud-sleuth/2.0.0.M5/single/spring-cloud-sleuth.html>) or in the GitHub (<https://github.com/spring-cloud/spring-cloud-sleuth>) project page.

Collection commands statistics with Hystrix

Now, we want to monitor our Hystrix commands. There are several commands in our microservices and probably the most used will be the OAuth token requester, because we always need to have a token to call any microservice in our system. Our Turbine server and Hystrix UI were configured at the beginning of this chapter and we will use these services right now.

Remember, we are using `spring-cloud-netflix-hystrix-stream` as an implementation to send Hystrix data to the Turbine server, as it performs better than HTTP and also brings some asynchronous characteristics.



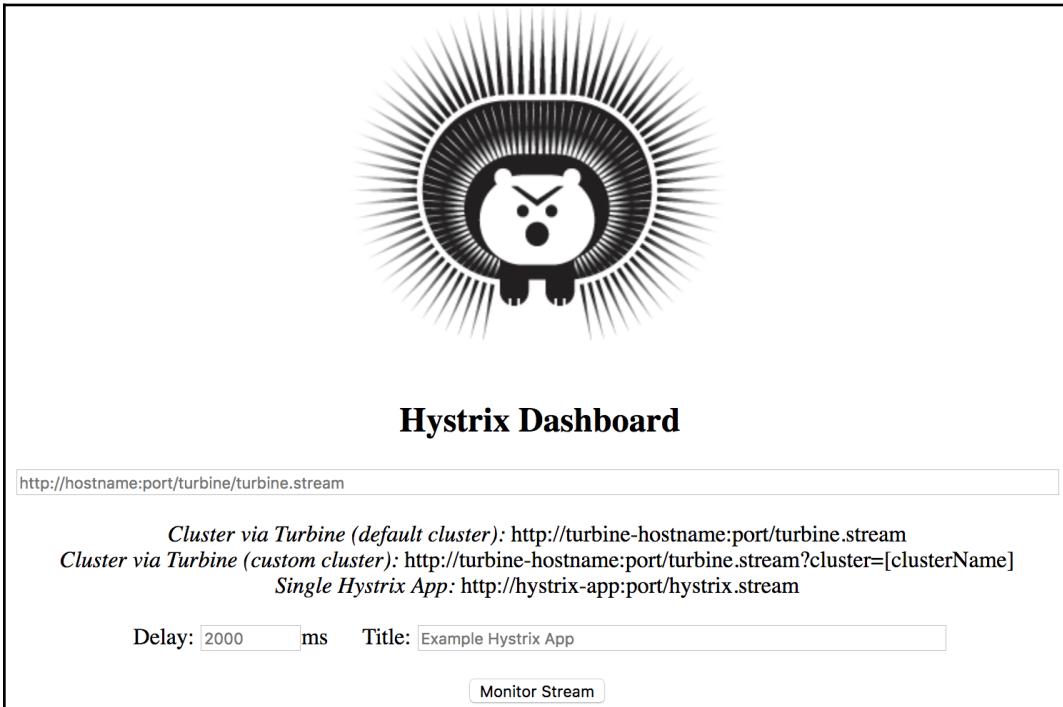
Asynchronous calls can make the microservice more resilient. In this case, we will not use HTTP calls (synchronous calls) to register Hystrix Commands statistics. We will use the RabbitMQ queue to register it. In this case, we will put the message in the queue. Also, asynchronous calls make our application more optimized to use computational resources.

Run the Turbine server application and Hystrix UI application. Turbine will aggregate the metrics from the servers. Optionally, you can run several instances of the same service, such as `flights`. Turbine will aggregate the statistics properly.

Let's call the Create Flights API; we can use the Postman to do that.

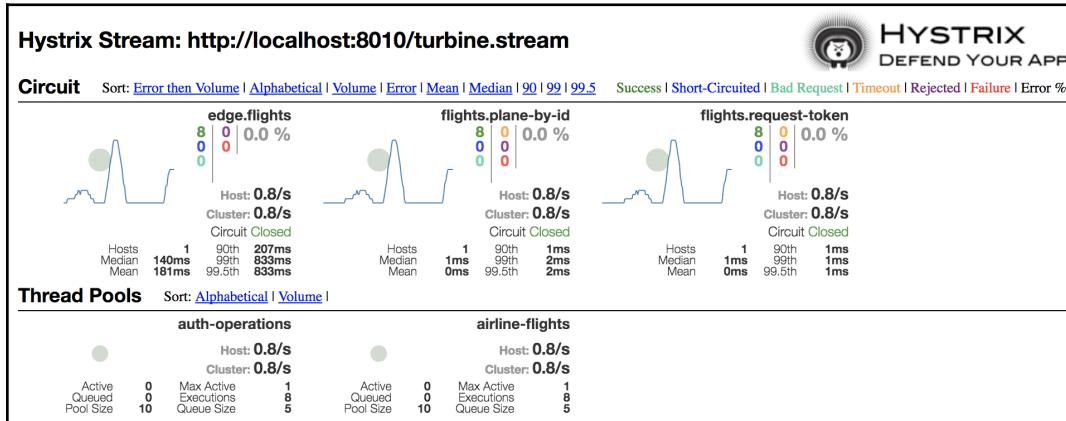
Then we can see the real-time commands statistics. Before that, we will configure `turbine.stream` in our Hystrix Dashboard.

Go to the Hystrix Dashboard page: <http://localhost:50010/hystrix/>. The following page will be displayed:



Then we have some work to do. Let's configure our Turbine server stream. Our Turbine stream is running at <http://localhost:8010/turbine.stream>. Put this information below the **Hystrix Dashboard** information, and then we can click on the **Monitor Stream** button.

We will redirect to the Hystrix Commands Dashboard; we called the Create Flights API a few times ago. The commands metrics will be displayed, like the following image:



As we can see, we called the Create Flights API eight times. This API uses some commands, such as `flights.plane-by-id`, it calls the planes microservice, and the `flights.request-token` calls the Auth service.

Look how easy it is to monitor the commands. Operation guys like the Zipkin server can use this page.

Awesome job, guys, our services integrations are adequately monitored, which makes our microservices adoption more comfortable because we have useful applications to monitor our services instances.

Dockerizing the microservices

In the previous chapters, we have used the Fabric8 Maven Docker plugin to enable us to create Docker images, using the Maven goals.

Now, we need to configure our microservices to use this plugin to easily create images for us. It can be helpful to integrate with some Continuous Integration and Delivery tools, such as Jenkins, because we can call the `docker: build` goal easily.

Each project has the custom configurations, such as port and image name. We can find the configuration at the GitHub repository. Remember, the configuration is done using the pom.xml.

The following list has the GitHub repository addresses for all projects; the pom.xml has the Maven Docker plugin configuration:

- **Flights:** <https://github.com/PacktPublishing/Spring-5.0-By-Example/blob/master/Chapter09/airline-flights/pom.xml>
- **Planes:** <https://github.com/PacktPublishing/Spring-5.0-By-Example/blob/master/Chapter09/airline-planes/pom.xml>
- **Fares:** <https://github.com/PacktPublishing/Spring-5.0-By-Example/blob/master/Chapter09/airline-fare/pom.xml>
- **Bookings:** <https://github.com/PacktPublishing/Spring-5.0-By-Example/blob/master/Chapter09/airline-booking/pom.xml>
- **Admin:** <https://github.com/PacktPublishing/Spring-5.0-By-Example/blob/master/Chapter09/admin/pom.xml>
- **EDGE:** <https://github.com/PacktPublishing/Spring-5.0-By-Example/blob/master/Chapter09/api-edge/pom.xml>
- **Passengers:** <https://github.com/PacktPublishing/Spring-5.0-By-Example/blob/master/Chapter09/airline-passengers/pom.xml>
- **Auth:** <https://github.com/PacktPublishing/Spring-5.0-By-Example/blob/master/Chapter09/auth-service/pom.xml>
- **Mail:** <https://github.com/PacktPublishing/Spring-5.0-By-Example/blob/master/Chapter09/mail-service/pom.xml>
- **Turbine:** <https://github.com/PacktPublishing/Spring-5.0-By-Example/blob/master/Chapter09/turbine/pom.xml>
- **Zipkin:** <https://github.com/PacktPublishing/Spring-5.0-By-Example/blob/master/Chapter09/zipkin-server/pom.xml>
- **Payments:** <https://github.com/PacktPublishing/Spring-5.0-By-Example/blob/master/Chapter09/airline-payments/pom.xml>
- **Hystrix-dashboard:** <https://github.com/PacktPublishing/Spring-5.0-By-Example/blob/master/Chapter09/hystrix-ui/pom.xml>
- **Discovery:** <https://github.com/PacktPublishing/Spring-5.0-By-Example/blob/master/Chapter09/eureka/pom.xml>
- **Config Server:** <https://github.com/PacktPublishing/Spring-5.0-By-Example/blob/master/Chapter09/config-server/pom.xml>

Running the system

Now we can run our Docker containers using our images, which were created in the previous section.

We will split the services into two Docker compose files. The first one is about infrastructure services. The second one is about our microservices.

The stacks must be run on the same Docker network, because the service should be connected by the container hostname.

The Docker compose file for infrastructure can be found at GitHub: <https://github.com/PacktPublishing/Spring-5.0-By-Example/blob/master/stacks/docker-compose-infra.yaml>.

The Docker compose file for microservices can be found at GitHub: <https://github.com/PacktPublishing/Spring-5.0-By-Example/blob/master/stacks/docker-compose-micro.yaml>.

Now, we can run these files using the docker-compose commands. Type the following commands:

```
docker-compose -f docker-compose-infra.yaml up -d  
docker-compose -f docker-compose-micro.yaml up -d
```

Then the full application will be up and running.

Well done, guys.

Summary

In this chapter, we have learned some important points on microservices architecture.

We were introduced to some important tools for monitoring the microservices environment. We have learned how the Turbine server can help us to monitor our Hystrix commands in distributed environments.

We were also introduced to the Hystrix Dashboard feature, which helps the developers and operations guys provide a rich dashboard with the commands statistics in near real time.

We learned how Spring Cloud Security enables security features for our microservices, and we implemented the OAuth 2 server, using JWT to enable resilience for our security layer.

9

Overview of GOF Design Patterns - Core Design Patterns

In this chapter, you'll be given an overview of GOF Design Patterns, including some best practices for making an application design. You'll also get an overview of common problem--solving with design patterns.

I will explain the design patterns that are commonly used by the Spring Framework for better design and architecture. We are all in a global world, which means that if we have services in the market, they can be accessed across the Globe. Simply put, now is the age of the distributed computing system. So first, what is a distributed system? It's an application that is divided into smaller parts that run simultaneously on different computers and the smaller parts communicate over the network, generally using protocols. These smaller parts are called **tiers**. So if we want to create a distributed application, *n*-tier architecture is a better choice for that type of application. But developing an *n*-tier distributed application is a complex and challenging job. Distributing the processing into separate tiers leads to better resource utilization. It also support the allocation of tasks to experts who are best suited to work and develop a particular tier. Many challenges exist in developing distributed applications, some of which are detailed here:

- Integration between the tiers
- Transaction management
- Concurrency handling of enterprise data
- Security of the application and so on

So my focus in this book is on simplifying Java EE application design and development by applying patterns and best practices with the Spring Framework. In this book, I will cover some common GOF Design Patterns, and how Spring adopted these for providing the best solutions to the aforementioned listed problems of enterprise application because the design of distributed objects is an immensely complicated task, even for experienced professionals. You need to consider critical issues, such as scalability, performance, transactions, and so on, before drafting a final solution. That solution is described as a pattern.

At the end of this chapter, you will understand how design patterns provide the best solution to address any design-related and development-related issues, and how to start development with the best practices. Here, you will get more ideas about GOF Design Patterns, with real-life examples. You will get information about how the Spring Framework implements these design patterns internally to provide the best enterprise solution.

This chapter will cover the following points:

- Introducing the power of design patterns
- Common GOF Design Patterns overview
 - Core design patterns
 - Creational design patterns
 - Structural design patterns
 - Behavioral design patterns
 - J2EE design patterns
 - Design patterns at presentation layer
 - Design patterns at business layer
 - Design patterns at integration layer
- Some best practices for Spring application development

Introducing the power of design patterns

So what is a design pattern? Actually, the phrase design pattern is not associated with any programming language, and also it doesn't provide language-specific solutions to problems. A design pattern is associated with the solution to repetitive problems. For example, if any problem occurs frequently, a solution to that problem has been used effectively. Any non-reusable solution to a problem can't be considered a pattern, but the problem must occur frequently in order to have a reusable solution, and to be considered as a pattern. So a design pattern is a software engineering concept describing recurring solutions to common problems in software design. Design patterns also represent the best practices used by experienced object-oriented software developers.

When you make a design for an application, you should consider all the solutions to common problems, and these solutions are called **design patterns**. The understanding of design patterns must be good across the developer team so that the staff can communicate with each other effectively. In fact, you may be familiar with some design patterns; however, you may not have used well-known names to describe them. This book will take you through a step-by-step approach and show you examples that use Java while you learn design pattern concepts.

A design pattern has three main characteristics:

- A Design pattern is *specific to a particular scenario* rather than a specific platform. So its context is the surrounding condition under which the problem exists. The context must be documented within the pattern.
- Design patterns have been *evolved to provide the best solutions* to certain problems faced during software development. So this should be limited by the context in which it is being considered.
- Design patterns are *the remedy for the problems under consideration*.

For example, if a developer is referring to the GOF Singleton design pattern and signifies the use of a single object, then all developers involved should understand that you need to design an object that will only have a single instance in the application. So the Singleton design pattern will be composed of a single object and the developers can tell each other that the program is following a Singleton pattern.

Common GoF Design Pattern overview

The authors Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides are often referred to as the GoF, or Gang of Four. They published a book titled *Design Patterns: Elements of Reusable Object-Oriented Software*, which initiated the concept of design patterns in software development.

In this chapter, you will learn what GOF patterns are and how they help solve common problems encountered in object-oriented design.

The **Gang of Four (GoF)** patterns are 23 classic software design patterns providing recurring solutions to common problems in software design. The patterns are defined in the book *Design Patterns: Elements of Reusable Object-Oriented Software*. These patterns are categorized into two main categories:

- Core Design Patterns
- J2EE Design Patterns

Furthermore, **Core Design Patterns** are also subdivided into three main categories of design pattern, as follows:

- **Creational Design Pattern:** Patterns under this category provide a way to construct objects when constructors will not serve your purpose. The creation logic of objects is hidden. The programs based on these patterns are more flexible in deciding object creation according to your demands and your use cases for the application.
- **Structural Design Pattern:** Patterns under this category deal with the composition of classes or objects. In the enterprise application, there are two commonly used techniques for reusing functionality in object-oriented systems: one is class Inheritance and the other is the Object Composition Concept of inheritance. The Object Composition Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities.
- **Behavioral Design Pattern:** Patterns under this category, characterize the ways in which classes or objects interact and distribute responsibility. These design patterns are specifically concerned with communication between objects. The behavioral design pattern is used to control and reduce complicated application flow in the enterprise application.

Now, let's look at the other category, the **JEE Design patterns**. This is the other main category of design patterns. Application design can be immensely simplified by applying Java EE design patterns. Java EE design patterns have been documented in Sun's Java Blueprints. These Java EE Design patterns provide time-tested solution guidelines and best practices for object interaction in the different layers of a Java EE application. These design patterns are specifically concerned with the following listed layers:

- Design pattern at the presentation layer
- Design pattern at the business layer
- Design pattern at the integration layer

Let's explore creational design patterns in the upcoming section.

Creational design patterns

Let's look at the underlying design patterns of this category and how Spring Framework adopts them to provide loose coupling between components and create and manage the lifecycle of Spring components. Creational design patterns are associated with the method of object creation. The creation logic of the object is hidden to the caller of this object.

We are all aware of how to create an object using the `new` keyword in Java, as follows:

```
Account account = new Account();
```

But this way is not suitable for some cases, because it is a hardcoded way of creating an object. It is also not a best practice to create an object because the object might be changed according to the nature of the program. Here, the creational design pattern provides the flexibility to create an object according to the nature of the program.

Now let's look at the different design patterns under this category.

Factory design pattern

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

- GOF Design Pattern

The Factory design pattern is a creational design pattern. The Factory design pattern is also known as the Factory method design pattern. According to this design pattern, you get an object of a class without exposing the underlying logic to the client. It assigns a new object to the caller by using a common interface or abstract class. This means that the design pattern hides the actual logic of the implementation of an object, how to create it, and which class to instantiate it in. So the client shouldn't worry about creating, managing, and destroying an object—the Factory pattern takes responsibility for these tasks. The Factory pattern is one of the most-used design patterns in Java.

Let's look at the benefits of the Factory pattern:

- The Factory pattern promotes loose coupling between collaborating components or classes by using interfaces rather than binding application-specific classes into the application code
- Using this pattern, you can get an implementation of an object of classes that implement an interface, at runtime
- The object life cycle is managed by the factory implemented by this pattern

Now let's discuss some common problems where you should apply the Factory design pattern:

- This pattern removes the burden on the developer to create and manage the objects
- This pattern removes the tight coupling between collaboration components because a component doesn't know what subclasses it will be required to create
- Avoid hard code to create an object of the class

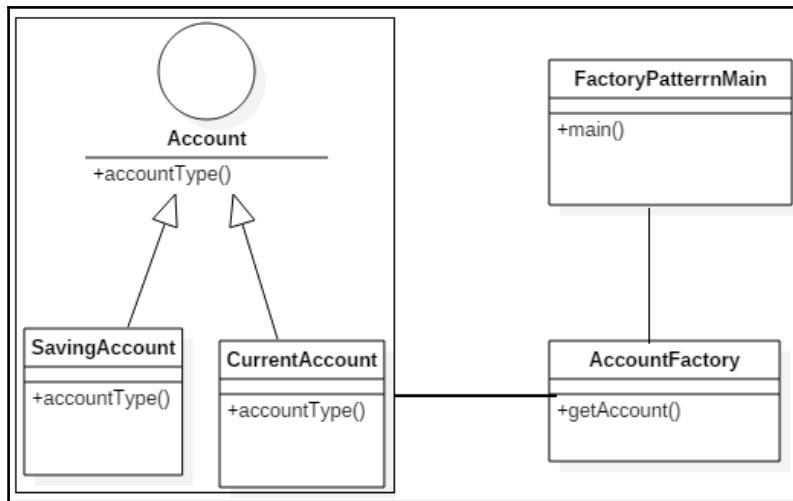
Implementing the Factory design pattern in Spring Framework

Spring Framework transparently uses this Factory design pattern to implement Spring containers using `BeanFactory` and `ApplicationContext` interfaces. Spring's container works based on the Factory pattern to create spring beans for the Spring application and also manages the life cycle of every Spring bean. `BeanFactory` and `ApplicationContext` are factory interfaces, and Spring has lots of implementing classes. The `getBean()` method is the factory method that gives you Spring beans accordingly.

Let's see a sample implementation of the Factory design pattern.

Sample implementation of the Factory design pattern

There are two classes `SavingAccount` and `CurrentAccount` implementing an interface `Account`. So, you can create a `Factory` class with a method that takes one or more arguments and its return type is `Account`. This method is known as the `Factory` method because it creates the instances of either `CurrentAccount` or `SavingAccount`. The `Account` interface is used for loose coupling. So, according to the passed arguments in the factory method, it chooses which subclass to instantiate. This factory method will have the superclass as its return type:



UML Diagram for the Factory design pattern

Let's look at this design pattern in the following example. Here, I am going to create an `Account` interface and some concrete classes that implement the `Account` interface:

```

package com.packt.patterninspring.chapter2.factory;
public interface Account {
    void accountType();
}
  
```

Now let's create `SavingAccount.java`, which will implement the `Account` interface:

```
package com.packt.patterninspring.chapter2.factory;
public class SavingAccount implements Account{
    @Override
    public void accountType() {
        System.out.println("SAVING ACCOUNT");
    }
}
```

Same with `CurrentAccount.java`, it will also implement the `Account` interface:

```
package com.packt.patterninspring.chapter2.factory;
public class CurrentAccount implements Account {
    @Override
    public void accountType() {
        System.out.println("CURRENT ACCOUNT");
    }
}
```

A Factory class `AccountFactory` is now going to be defined. `AccountFactory` generates an object of the concrete class, either `SavingAccount` or `CurrentAccount`, based on the account type given as an argument to the Factory method:

`AccountFactory.java` is a Factory to produce the `Account` type object:

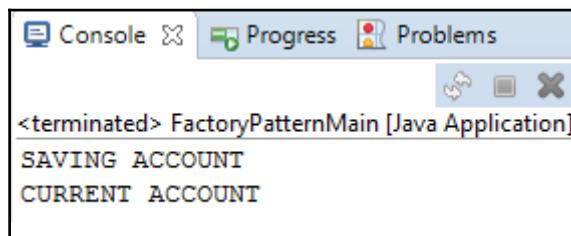
```
package com.packt.patterninspring.chapter2.factory.pattern;
import com.packt.patterninspring.chapter2.factory.Account;
import com.packt.patterninspring.chapter2.factory.CurrentAccount;
import com.packt.patterninspring.chapter2.factory.SavingAccount;
public class AccountFactory {
    final String CURRENT_ACCOUNT = "CURRENT";
    final String SAVING_ACCOUNT = "SAVING";
    //use getAccount method to get object of type Account
    //It is factory method for object of type Account
    public Account getAccount(String accountType){
        if(CURRENT_ACCOUNT.equals(accountType)) {
            return new CurrentAccount();
        }
        else if(SAVING_ACCOUNT.equals(accountType)){
            return new SavingAccount();
        }
        return null;
    }
}
```

`FactoryPatternMain` is the main calling class of `AccountFactory` to get an `Account` object. It will pass an argument to the factory method that contains information of the account type, such as `SAVING` and `CURRENT`. `AccountFactory` returns the object of the type that you passed to the factory method.

Let's create a demo class `FactoryPatterMain.java` to test the factory method design pattern:

```
package com.packt.patterninspring.chapter2.factory.pattern;
import com.packt.patterninspring.chapter2.factory.Account;
public class FactoryPatterMain {
    public static void main(String[] args) {
        AccountFactory accountFactory = new AccountFactory();
        //get an object of SavingAccount and call its accountType()
        //method.
        Account savingAccount = accountFactory.getAccount("SAVING");
        //call accountType method of SavingAccount
        savingAccount.accountType();
        //get an object of CurrentAccount and call its accountType()
        //method.
        Account currentAccount =
        accountFactory.getAccount("CURRENT");
        //call accountType method of CurrentAccount
        currentAccount.accountType();
    }
}
```

You can test this file and see the output on the console, which should look like this:



Now that we've seen the Factory design pattern, let's turn to a different variant of it—the Abstract factory design pattern.

Abstract factory design pattern

Provide an interface for creating families of related or dependent objects without specifying their concrete classes. - GOF Design Patterns

The Abstract Factory pattern comes under the creational design pattern. It is a high-level design pattern compared to the factory method design pattern. According to this design pattern, you just define an interface or abstract class to create a related dependent object without specifying its concrete subclass. So here, the abstract factory returns a factory of classes. Let me simplify it for you. You have a set of factory method design patterns, and you just put these factories under a factory using the factory design pattern, which means that it is simply a factory of factories. And there is no need to take the knowledge about all of the factories into the factory--you can make your program using a top-level factory.

In the Abstract Factory pattern, an interface is responsible for creating a factory of related objects without explicitly specifying their classes. Each generated factory can give the objects as per the Factory pattern.

The benefits of the Abstract Factory pattern are as follows:

- The Abstract Factory Design provides loose coupling between the component families. It also isolates the client code from concrete classes.
- This design pattern is a higher-level design than the Factory pattern.
- This pattern provides better consistency at construction time of objects across the application.
- This pattern easily swaps component families.

Common problems where you should apply the Abstract factory design pattern

When you design a Factory pattern for object creation in your application, there are times when you want a particular set of related objects to be created with certain constraints and apply the desired logic across the related objects in your application. You can achieve this design by creating another factory inside the factory for a set of related objects and apply the required constraints. You can also program the logic to a set of related objects.

When you want to customize the instantiation logic of related objects, then you could use this design pattern.

Implementing the Abstract factory design pattern in the Spring Framework

In the Spring Framework, the `FactoryBean` interface is based on the Abstract Factory design pattern. Spring provides a lot of implementation of this interface, such as `ProxyFactoryBean`, `JndiFactoryBean`, `LocalSessionFactoryBean`, `LocalContainerEntityManagerFactoryBean`, and so on. A `FactoryBean` is also useful to help Spring construct objects that it couldn't easily construct itself. Often this is used to construct complex objects that have many dependencies. It might also be used when the construction logic itself is highly volatile and depends on the configuration.

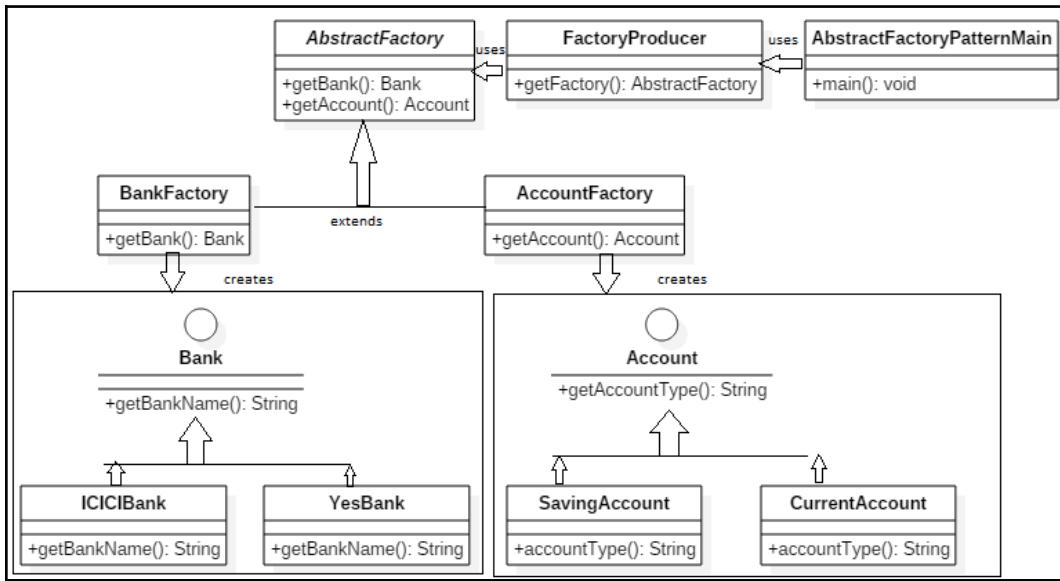
For example, in Spring Framework, one of the `FactoryBean` implementations is `LocalSessionFactoryBean`, which is used to get a reference of a bean that was associated with the hibernate configuration. It is a specific configuration concerning the data source. It should be applied before you get an object of `SessionFactory`. You can use the `LocalSessionFactoryBean` to apply the specific data source configuration in a consistent way. You may inject the result of a `FactoryBean`'s `getObject()` method into any other property.

Let's create a sample implementation of the Abstract Factory design pattern.

Sample implementation of the Abstract Factory design pattern

I am going to create a `Bank` and `Account` interface and some concrete classes implementing these interfaces. Here, I also create an abstract factory class, `AbstractFactory`. I have some factory classes, `BankFactory` and `AccountFactory`; these classes extend the `AbstractFactory` class. I will also create a `FactoryProducer` class to create the factories.

Let's see this design pattern in the following image:



UML diagram for the Abstract Factory design pattern

Create a demo class, `AbstractFactoryPatternMain`; it uses `FactoryProducer` to get an `AbstractFactory` object. Here, I pass information such as ICICI, YES to `AbstractFactory` to get an object of `Bank`, and I also pass information such as SAVING, CURRENT to `AbstractFactory` to get an `Account` type.

Here is the code for `Bank.java`, which is an interface:

```

package com.packt.patterninspring.chapter2.model;
public interface Bank {
    void bankName();
}
  
```

Now let's create `ICICIBank.java`, which implements the `Bank` interface:

```
package com.packt.patterninspring.chapter2.model;
public class ICICIBank implements Bank {
    @Override
    public void bankName() {
        System.out.println("ICICI Bank Ltd.");
    }
}
```

Let's create another `YesBank.java`, an implementing `Bank` interface:

```
package com.packt.patterninspring.chapter2.model;
public class YesBank implements Bank{
    @Override
    public void bankName() {
        System.out.println("Yes Bank Pvt. Ltd.");
    }
}
```

In this example, I am using the same interface and implementing classes of `Account` as I used in the Factory pattern example in this book.

`AbstractFactory.java` is an abstract class that is used to get factories for `Bank` and `Account` objects:

```
package
com.packt.patterninspring.chapter2.abstractfactory.pattern;
import com.packt.patterninspring.chapter2.model.Account;
import com.packt.patterninspring.chapter2.model.Bank;
public abstract class AbstractFactory {
    abstract Bank getBank(String bankName);
    abstract Account getAccount(String accountType);
}
```

`BankFactory.java` is a factory class extending `AbstractFactory` to generate an object of the concrete class based on the given information:

```
package
com.packt.patterninspring.chapter2.abstractfactory.pattern;
import com.packt.patterninspring.chapter2.model.Account;
import com.packt.patterninspring.chapter2.model.Bank;
import com.packt.patterninspring.chapter2.model.ICICIBank;
import com.packt.patterninspring.chapter2.model.YesBank;
public class BankFactory extends AbstractFactory {
    final String ICICI_BANK = "ICICI";
    final String YES_BANK   = "YES";
    //use getBank method to get object of name bank
```

```

//It is factory method for object of name bank
@Override
Bank getBank(String bankName) {
    if(ICICI_BANK.equalsIgnoreCase(bankName)) {
        return new ICICIBank();
    }
    else if(YES_BANK.equalsIgnoreCase(bankName)) {
        return new YesBank();
    }
    return null;
}
@Override
Account getAccount(String accountType) {
    return null;
}
}

```

`AccountFactory.java` is a factory class that extends `AbstractFactory.java` to generate an object of the concrete class based on the given information:

```

package
com.packt.patterninspring.chapter2.abstractfactory.pattern;
import com.packt.patterninspring.chapter2.model.Account;
import com.packt.patterninspring.chapter2.model.Bank;
import com.packt.patterninspring.chapter2.model.CurrentAccount;
import com.packt.patterninspring.chapter2.model.SavingAccount;
public class AccountFactory extends AbstractFactory {
    final String CURRENT_ACCOUNT = "CURRENT";
    final String SAVING_ACCOUNT = "SAVING";
    @Override
    Bank getBank(String bankName) {
        return null;
    }
    //use getAccount method to get object of type Account
    //It is factory method for object of type Account
    @Override
    public Account getAccount(String accountType){
        if(CURRENT_ACCOUNT.equals(accountType)) {
            return new CurrentAccount();
        }
        else if(SAVING_ACCOUNT.equals(accountType)){
            return new SavingAccount();
        }
        return null;
    }
}

```

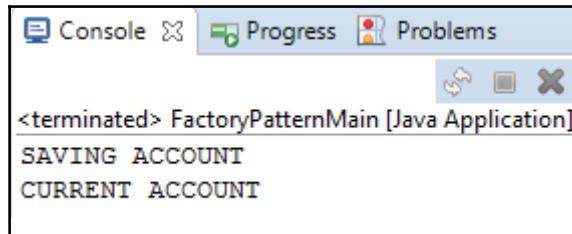
`FactoryProducer.java` is a class that creates a Factory generator class to get factories by passing a piece of information, such as Bank or Account:

```
package com.packt.patterninspring.chapter2.abstractfactory.pattern;
public class FactoryProducer {
    final static String BANK      = "BANK";
    final static String ACCOUNT   = "ACCOUNT";
    public static AbstractFactory getFactory(String factory) {
        if(BANK.equalsIgnoreCase(factory)){
            return new BankFactory();
        }
        else if(ACCOUNT.equalsIgnoreCase(factory)){
            return new AccountFactory();
        }
        return null;
    }
}
```

`FactoryPatterMain.java` is a demo class for the Abstract Factory design pattern. `FactoryProducer` is a class to get `AbstractFactory` in order to get the factories of concrete classes by passing a piece of information, such as the type:

```
package com.packt.patterninspring.chapter2.factory.pattern;
import com.packt.patterninspring.chapter2.model.Account;
public class FactoryPatterMain {
    public static void main(String[] args) {
        AccountFactory accountFactory = new AccountFactory();
        //get an object of SavingAccount and call its accountType()
        //method.
        Account savingAccount = accountFactory.getAccount("SAVING");
        //call accountType method of SavingAccount
        savingAccount.accountType();
        //get an object of CurrentAccount and call its accountType()
        //method.
        Account currentAccount =
accountFactory.getAccount("CURRENT");
        //call accountType method of CurrentAccount
        currentAccount.accountType();
    }
}
```

You can test this file and see the output on the console:



```
<terminated> FactoryPatternMain [Java Application]
SAVING ACCOUNT
CURRENT ACCOUNT
```

Now that we've seen the abstract Factory design pattern, let's turn to a different variant of it—the singleton design pattern.

Singleton design pattern

Ensure a class has only one instance and provide a global point of access to it - GOF Design Patterns

The Singleton pattern is a creational design pattern, it is one of the simplest design patterns in Java. According to the singleton design pattern, the class provides the same single object for each call—that is, it is restricting the instantiation of a class to one object and provides a global point of access to that class. So the class is responsible for creating an object and also ensures that only a single object should be created for each client call for this object. This class doesn't allow a direct instantiation of an object of this class. It allows you to get an object instance only by an exposed static method.

This is useful when exactly one object is needed to coordinate actions across the system. You can create a single pattern using two forms, as listed here:

- **Early instantiation:** Creation of instance at load time
- **Lazy instantiation:** Creation of instance when required

Benefits of the Singleton pattern:

- It provides controller access to crucial (usually heavy object) classes, such as the connection class for DB and the SessionFactory class in hibernate
- It saves heaps of memory
- It is a very efficient design for multithreaded environments
- It is more flexible because the class controls the instantiation process, and the class has the flexibility to change the instantiation process
- It has low latency

Common problems where you should apply Singleton pattern

The Singleton pattern solves only one problem--if you have a resource that can only have a single instance, and you need to manage that single instance, then you need a singleton. Normally, if you want to create a database connection with the given configuration in the distributed and multithread environment, it might be the case that every thread can create a new database connection with a different configuration object, if you don't follow the singleton design. With the Singleton pattern, each thread gets the same database connection object with the same configuration object across the system. It is mostly used in multithreaded and database applications. It is used in logging, caching, thread pools, configuration settings, and so on.

Singleton design pattern implementation in the Spring Framework

The Spring Framework provides a Singleton scoped bean as a singleton pattern. It is similar to the singleton pattern, but it's not exactly the same as the Singleton pattern in Java. According to the Singleton pattern, a scoped bean in the Spring Framework means a single bean instance per container and per bean. If you define one bean for a particular class in a single Spring container, then the Spring container creates one and only one instance of the class defined by that bean definition.

Let's create a sample application of the singleton design pattern.

Sample implementation of the Singleton design pattern

In the following code example, I will be creating a class with a method to create an instance of this class if one does not exist. If the instance is already present, then it will simply return the reference of that object. I have also taken thread safety into consideration, and so I have used a synchronized block here before creating the object of that class.

Let's check out the UML diagram for the Singleton design pattern:

```
package com.packt.patterninspring.chapter2.singleton.pattern;
public class SingletonClass {
    private static SingletonClass instance = null;
    private SingletonClass() {
    }
    public static SingletonClass getInstance() {
        if (instance == null) {
            synchronized(SingletonClass.class){
                if (instance == null) {
                    instance = new SingletonClass();
                }
            }
        }
        return instance;
    }
}
```

One thing to be noted in the preceding code is that I have written a private constructor of the `SingletonClass` class to make sure that there is no way to create the object of that class. This example is based on lazy initialization, which means that the program creates an instance on demand the first time. So you could also eagerly instantiate the object to improve the runtime performance of your application. Let's see the same `SingletonClass` with eager initialization:

```
package com.packt.patterninspring.chapter2.singleton.pattern;
public class SingletonClass {
    private static final SingletonClass INSTANCE =
        new SingletonClass();
    private SingletonClass() {}
    public static SingletonClass getInstance() {
        return INSTANCE;
    }
}
```

Now that we've seen the singleton design pattern, let's turn to a different variant of it—the Prototype design pattern.

Prototype design pattern

Specify the kind of objects to create using a prototypical instance, and create new objects by copying this prototype. - GOF Design Patterns

The Prototype pattern comes under the creational design pattern family of GOF patterns in software development. This pattern is used to create the objects by using a clone method of objects. It is determined by a prototypical instance. In the enterprise application, object creation is costly in terms of creating and initializing the initial properties of objects. If such a type of object is already in your hand, then you go for the prototype pattern; you just copy an existing similar object instead of creating it, which is time-consuming.

This pattern involves implementing a prototype interface, it creates a clone of the current object. This pattern is used when the direct creation of the object is costly. For example, say that an object is to be created after a costly database operation. We can cache the object, returns its clone on the next request, and update the database as and when it is needed, thus reducing database calls.

Benefits of the Prototype design pattern

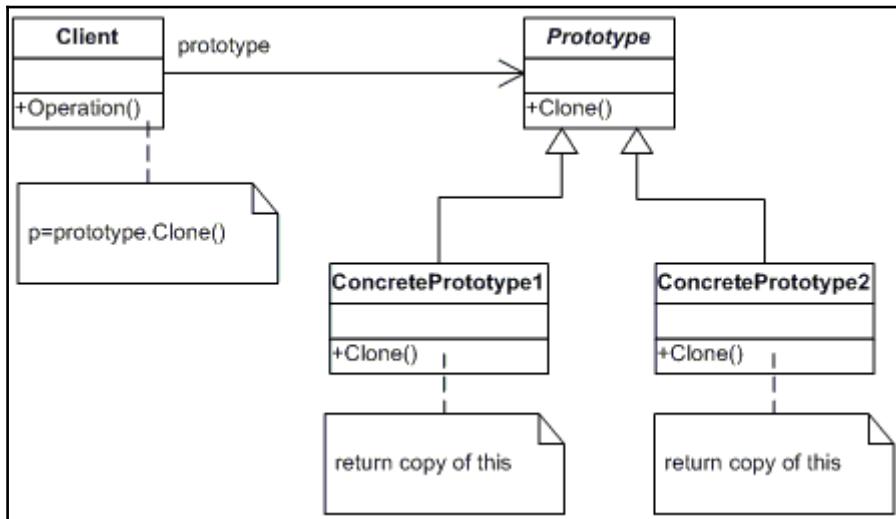
The following list shows the benefits of using the Prototype pattern:

- Reduces the time to create the time-consuming objects by using the prototype pattern
- This pattern reduces subclassing
- This pattern adds and removes objects at runtime
- This pattern configures the application with classes dynamically

Let's see the UML class structure of the Prototype design pattern.

UML class structure

The following UML diagram shows all the components of the Prototype design pattern:



UML diagram for Prototype design pattern

Let's see these components as listed in following points:

- **Prototype**: The Prototype is an interface. It uses the clone method to create instances of this interface type.
- **ConcretePrototype**: This is a concrete class of the Prototype interface to implement an operation to clone itself.
- **Client**: This is a caller class to create a new object of a Prototype interface by calling a `clone` method of the prototype interface.

Let's see a sample implementation of the prototype design pattern.

Sample implementation of the Prototype design pattern

I am going to create an abstract Account class and concrete classes extending the Account class. An AccountCache class is defined as a next step, which stores account objects in a HashMap and returns their clone when requested. Create an abstract class implementing the Clonable interface.

```
package com.packt.patterninspring.chapter2.prototype.pattern;
public abstract class Account implements Cloneable{
    abstract public void accountType();
    public Object clone() {
        Object clone = null;
        try {
            clone = super.clone();
        }
        catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
        return clone;
    }
}
```

Now let's create concrete classes extending the preceding class:

Here's the CurrentAccount.java file:

```
package com.packt.patterninspring.chapter2.prototype.pattern;
public class CurrentAccount extends Account {
    @Override
    public void accountType() {
        System.out.println("CURRENT ACCOUNT");
    }
}
```

Here's how SavingAccount.java should look:

```
package com.packt.patterninspring.chapter2.prototype.pattern;
public class SavingAccount extends Account{
    @Override
    public void accountType() {
        System.out.println("SAVING ACCOUNT");
    }
}
```

Let's create a class to get concrete classes in the `AccountCache.java` file:

```
package com.packt.patterninspring.chapter2.prototype.pattern;
import java.util.HashMap;
import java.util.Map;
public class AccountCache {
    public static Map<String, Account> accountCacheMap =
        new HashMap<>();
    static{
        Account currentAccount = new CurrentAccount();
        Account savingAccount = new SavingAccount();
        accountCacheMap.put("SAVING", savingAccount);
        accountCacheMap.put("CURRENT", currentAccount);
    }
}
```

`PrototypePatternMain.java` is a demo class that we will use to test the design pattern `AccountCache` to get the `Account` object by passing a piece of information, such as the type, and then call the `clone()` method:

```
package com.packt.patterninspring.chapter2.prototype
    .pattern;
public class PrototypePatternMain {
    public static void main(String[] args) {
        Account currentAccount = (Account)
            AccountCache.accountCacheMap.get("CURRENT").clone();
        currentAccount.accountType();
        Account savingAccount = (Account)
            AccountCache.accountCacheMap.get("SAVING") .clone();
        savingAccount.accountType();
    }
}
```

We've covered this so far and it's good. Now let's look at the next design pattern.

Builder design pattern

Separate the construction of a complex object from its representation so that the same construction process can create different representations. - GOF Design Patterns

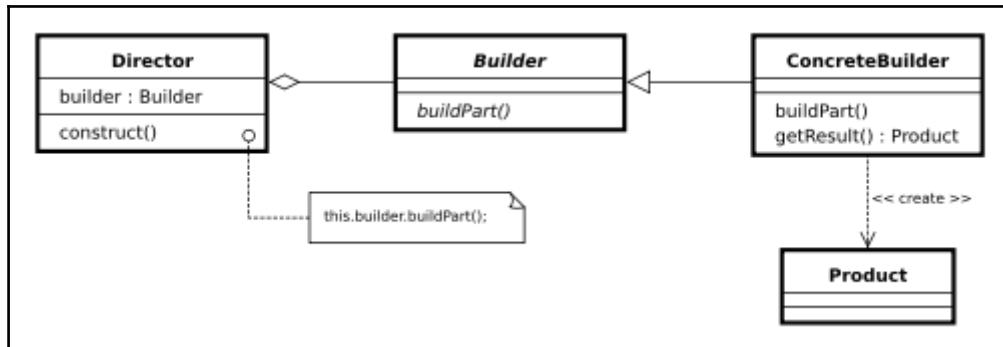
The Builder design pattern is used to construct a complex object step by step, and finally it will return the complete object. The logic and process of object creation should be generic so that you can use it to create different concrete implementations of the same object type. This pattern simplifies the construction of complex objects and it hides the details of the object's construction from the client caller code. When you are using this pattern, remember you have to build it one step at a time, which means you have to break the object construction login into multiple phases, unlike other patterns, such as the abstract factory and the factory method pattern, which the object in a single step.

Benefits of the Builder pattern:

- This pattern provides you with complete isolation between the construction and representation of an object
- This pattern allows you to construct the object in multiple phases, so you have greater control over the construction process
- This pattern provides the flexibility to vary an object's internal representation

UML class structure

The following UML diagram shows all the components of the Builder design pattern:



UML diagram for the Builder design pattern:

- **Builder** (AccountBuilder): This is an abstract class or interface for creating the details of an Account object.
- **ConcreteBuilder**: This is an implementation to construct and assemble details of the account by implementing the Builder interface.
- **Director**: This constructs an object using the Builder interface.
- **Product** (Account): This represents the complex object under construction. AccountBuilder builds the account's internal representation and defines the process by which it's assembled.

Implementing the Builder pattern in the Spring Framework

The Spring Framework implements the Builder design pattern transparently in some functionalities. The following classes are based on the Builder design pattern in the Spring Framework:

- EmbeddedDatabaseBuilder
- AuthenticationManagerBuilder
- UriComponentsBuilder
- BeanDefinitionBuilder
- MockMvcWebClientBuilder

Common problems where you should apply Builder pattern

In an enterprise application, you can apply the Builder pattern where the object creation has been done by using multiple steps. In each step, you do a portion of the process. In this process, you set some required parameters and some optional parameters, and after the final step, you will get a complex object.

The Builder pattern is an object creation software design pattern. The intention is to abstract the steps of construction so that different implementations of these steps can construct different representations of objects. Often, the Builder pattern is used to build products in accordance with the composite pattern.

Sample implementation of the Builder design pattern

In the following code example, I am going to create an `Account` class that has `AccountBuilder` as an inner class. The `AccountBuilder` class has a method to create an instance of this class:

```
package com.packt.patterninspring.chapter2.builder.pattern;
public class Account {
    private String accountName;
    private Long accountNumber;
    private String accountHolder;
    private double balance;
    private String type;
    private double interest;
    private Account(AccountBuilder accountBuilder) {
        super();
        this.accountName = accountBuilder.accountName;
        this.accountNumber = accountBuilder.accountNumber;
        this.accountHolder = accountBuilder.accountHolder;
        this.balance = accountBuilder.balance;
        this.type = accountBuilder.type;
        this.interest = accountBuilder.interest;
    }
    //setters and getters
    public static class AccountBuilder {
        private final String accountName;
        private final Long accountNumber;
        private final String accountHolder;
        private double balance;
        private String type;
        private double interest;
        public AccountBuilder(String accountName,
            String accountHolder, Long accountNumber) {
            this.accountName = accountName;
            this.accountHolder = accountHolder;
            this.accountNumber = accountNumber;
        }
        public AccountBuilder balance(double balance) {
            this.balance = balance;
            return this;
        }
        public AccountBuilder type(String type) {
            this.type = type;
            return this;
        }
        public AccountBuilder interest(double interest) {
```

```

        this.interest = interest;
        return this;
    }
    public Account build() {
        Account user = new Account(this);
        return user;
    }
}
public String toString() {
    return "Account [accountName=" + accountName + ", "
           + accountNumber + ", accountHolder="
           + accountHolder + ", balance=" + balance + ", type="
           + type + ", interest=" + interest + "]";
}
}

```

`AccountBuilderTest.java` is a demo class that we will use to test the design pattern. Let's look at how to build an `Account` object by passing the initial information to the object:

```

package com.packt.patterninspring.chapter2.builder.pattern;
public class AccountBuilderTest {
    public static void main(String[] args) {
        Account account = new Account.AccountBuilder("Saving
            Account", "Dinesh Rajput", 11111)
            .balance(38458.32)
            .interest(4.5)
            .type("SAVING")
            .build();
        System.out.println(account);
    }
}

```

You can test this file and see the output on the console:

```

<terminated> AccountBuilderTest [Java Application] C:\Program Files\Java\jre1.8.0_131\bin\javaw.exe (27-Jun-2017, 2:10:21 AM)
Account [accountName=Saving Account, accountNumber=11111, accountHolder=Dinesh Rajput, balance=38458.32, type=SAVING, interest=4.5]

```

Now, we've seen the Builder design pattern. In the upcoming Chapter 3, *Consideration of Structural and Behavioural Patterns*, I will explore another part of the GOF Design Patterns family.

Summary

After reading this chapter, the reader should now have a good idea about the overview of GOF creational design patterns and its best practices. I highlighted the problems that come from not using design patterns in enterprise application development, and how Spring solves these problems by using the creational design patterns and good practices in the application. In this chapter, I have mentioned only one of the Creational Design pattern categories out of the three main categories of the GOF Design Patterns. The Creational design pattern is used for the creation of object instances, and also applies constraints at the creation time in the enterprise application in a specific manner using the Factory, Abstract Factory, Builder, Prototype, and Singleton patterns. In the next chapter, we will look at the other categories of the GOF Design Patterns—the structural design pattern and the behavioral design pattern. The structural design pattern is used to design the structure of an enterprise application by dealing with the composition of classes or objects so that it reduces the application's complexity and improves the reusability and performance of the application. The Adapter Pattern, Bridge Pattern, Composite Pattern, Decorator Pattern, Facade Pattern, and Flyweight Pattern come under this category of the pattern. The Behavioral design pattern characterizes the ways in which classes or objects interact and distribute responsibility. The patterns that come under this category are specifically concerned with communication between objects. Let's move to complete the remaining GOF patterns in the next chapter.

10

Consideration of Structural and Behavioral Patterns

You have seen implementations and examples of the creational design pattern from the GOF pattern family in [Chapter 2, Overview of GOF Design Patterns - Core Design Patterns](#). Now, in this chapter, you'll be given an overview of other parts of GOF Design Patterns, they are the structural and behavioral design patterns, including some best practices for application design. You'll also get an overview of common problem solving with these design patterns.

At the end of this chapter, you will understand how these design patterns provide the best solution to address the design and development related issues in the object composition and delegating responsibilities between the working objects in the application. You will get information about how the Spring Framework implements the structural and behavioral designs pattern internally to provide best enterprise solutions.

This chapter will cover the following points:

- Implementing the structural design patterns
- Implementing the behavioral design patterns
- J2EE design patterns

Examining the core design patterns

Let's continue our journey into the core design patterns:

- **Structural design pattern:** Patterns under this category deal with the composition of classes or objects. In the enterprise application, there are two common techniques for reusing functionality in object-oriented systems as follows:
 - **Inheritance:** It is used to inherit commonly used states and behaviors from other classes.
 - **Composition:** It is used to compose the other objects as instance variables of classes. It defines ways to compose objects to obtain new functionalities.
- **Behavioral design pattern:** Patterns under this category characterize the ways in which classes or objects interact with and distribute responsibility. These patterns define the methods of communication between the objects in the enterprise application. So here, you will learn how to use behavioral patterns to reduce complicated flow control. Furthermore, you will use behavioral patterns to encapsulate algorithms and dynamically select them at runtime.

Structural design patterns

In the previous section, we discussed creational design patterns and how they provide the best solutions for object creation according to business demands. Creational design patterns only provide a solution for creating objects in the application with how these objects merge with each other in the application for a specific business goal, the structural design pattern comes into the picture. In this chapter, we will be exploring structural patterns, and how these patterns are useful to define the relationship between the objects either using inheritance or composition for larger structures of an application. Structural patterns allow you to solve many problems related to structuring the relationship between the objects. They show you how to glue different parts of a system together in a flexible and extensible fashion. Structural patterns help you guarantee that when one of the parts changes, the entire structure does not need to change; in a car you could replace the tyres with different vendors without impacting the other parts of that car. They also show you how to recast parts of the system that do not fit (but that you need to use) into parts that do fit.

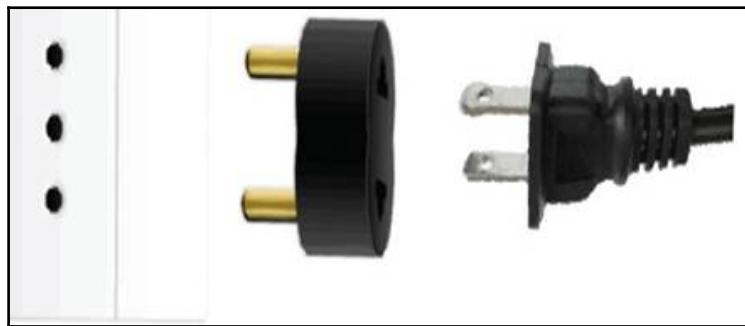
The adapter design pattern

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

-GoF Design Patterns: Elements of Reusable Object-Oriented Software

Adapter design patterns come under the structural design pattern, according to this design pattern two incompatible classes work together that couldn't otherwise because of incompatible interfaces. This pattern works as a bridge between two incompatible interfaces. This pattern is used when two inferences of the application are incompatible in their functionalities, but these functionalities need to be integrated as a business requirement.

There are many real-life examples where we can use the adapter pattern. Suppose you have different types of electric plugs such as cylindrical and rectangular plugs, as shown in the following figure. You can use an adapter in between to fit a rectangular plug in a cylindrical socket assuming voltage requirements are met:



Benefits of the adapter pattern

Let's look at the following benefits of using the adapter design pattern in the application.

- The adapter pattern allows you to communicate and interact with two or more incompatible objects
- This pattern promotes the reusability of older existing functionalities in your application

Common requirements for the adapter pattern

The following are the common requirements for this design pattern to addresses the design problems:

- If you are to use this pattern in your application, there is a need to use an existing class with an incompatible interface.
- Another use of this pattern in your application is when you want to create a reusable class that collaborates with classes that have incompatible interfaces.
- There are several existing subclasses to be used, but it's impractical to adapt their interface by sub classing each one. An object adapter can adapt the interface of its parent class.

Let's see how Spring implements the adapter design pattern internally.

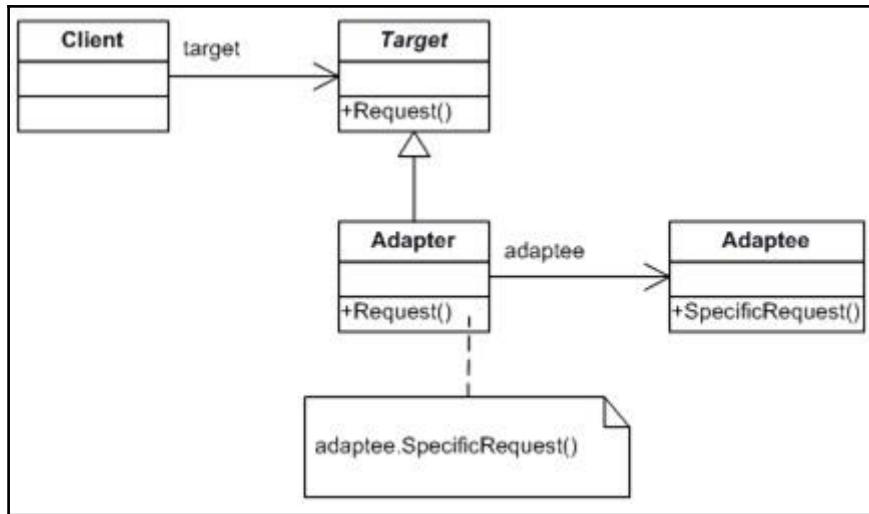
Implementation of the adapter design pattern in the Spring Framework

Spring Framework uses the adapter design pattern to implement a lot of functionality across the framework transparently. The following are some listed classes based on the adapter design pattern in the Spring Framework:

- JpaVendorAdapter
- HibernateJpaVendorAdapter
- HandlerInterceptorAdapter
- MessageListenerAdapter
- SpringContextResourceAdapter
- ClassPreProcessorAgentAdapter
- RequestMappingHandlerAdapter
- AnnotationMethodHandlerAdapter
- WebMvcConfigurerAdapter

The UML diagram for the adapter pattern

Let's understand the preceding UML diagram that illustrates the components of the adapter design pattern:



- **The Target Interface:** This is the desired interface class that will be used by the clients
- **The Adapter class:** This class is a wrapper class that implements the desired target interface and modifies the specific request available from the Adaptee class
- **The Adaptee class:** This is the class that is used by the Adapter class to reuse the existing functionalities and modify them for desired use
- **Client:** This class will interact with the Adapter class

Let's look at the following sample implementation of the adapter design pattern.

Sample implementation of the adapter design pattern

I am going to create an example that shows the actual demonstration of the adapter design pattern, so let's discuss this example, I am creating this example based on making payment through a payment gateway. Suppose I have one old payment gateway and also have the latest advanced payment gateway, and both gateways are unrelated to each other, so my requirement is, I want to migrate from the old payment gateway to an advanced payment gateway while changing my existing source code. I am creating an adapter class to solve this problem. This adapter class is working as a bridge between two different payment gateways, let's look at the following code:

Let's now create an interface for the old payment gateway:

```
package com.packt.patterninspring.chapter3.adapter.pattern;
import com.packt.patterninspring.chapter3.model.Account;
public interface PaymentGateway {
    void doPayment(Account account1, Account account2);
}
```

Let's now create an implementation class for the old payment gateway

PaymentGateway.java:

```
package com.packt.patterninspring.chapter3.adapter.pattern;
import com.packt.patterninspring.chapter3.model.Account;
public class PaymentGatewayImpl implements PaymentGateway{
    @Override
    public void doPayment(Account account1, Account account2) {
        System.out.println("Do payment using Payment Gateway");
    }
}
```

The following interface and its implementation have new and advanced functionalities for the payment gateway:

```
package com.packt.patterninspring.chapter3.adapter.pattern;
public interface AdvancedPayGateway {
    void makePayment(String mobile1, String mobile2);
}
```

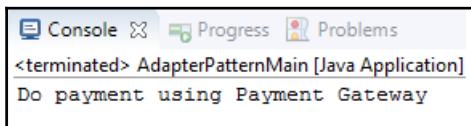
Let's now create an implementation class for the advance payment gateway interface:

```
package com.packt.patterninspring.chapter3.adapter.pattern;
import com.packt.patterninspring.chapter3.model.Account;
public class AdvancedPaymentGatewayAdapter implements
    AdvancedPayGateway{
    private PaymentGateway paymentGateway;
    public AdvancedPaymentGatewayAdapter(PaymentGateway
        paymentGateway) {
        this.paymentGateway = paymentGateway;
    }
    public void makePayment(String mobile1, String mobile2) {
        Account account1 = null;//get account number by
        mobile number mobile
        Account account2 = null;//get account number by
        mobile number mobile
        paymentGateway.doPayment(account1, account2);
    }
}
```

Let's see a demo class for this pattern as follows:

```
package com.packt.patterninspring.chapter3.adapter.pattern;
public class AdapterPatternMain {
    public static void main(String[] args) {
        PaymentGateway paymentGateway = new PaymentGatewayImpl();
        AdvancedPayGateway advancedPayGateway = new
            AdvancedPaymentGatewayAdapter(paymentGateway);
        String mobile1 = null;
        String mobile2 = null;
        advancedPayGateway.makePayment(mobile1, mobile2);
    }
}
```

In the preceding class, we have the old payment gateway object as the `PaymentGateway` interface, but we convert this old payment gateway implementation to the advanced form of the payment gateway by using the `AdvancedPaymentGatewayAdapter` adapter class. Let's run this demo class and see the output as follows:



Now that we've seen the adapter design pattern, let's turn to a different variant of it--the Bridge design pattern.

The Bridge design pattern

Decouple an abstraction from its implementation so that the two can vary independently

- GoF Design Patterns: Elements of Reusable Object-Oriented Software

In software engineering, one of the most popular notions is preferred composition over inheritance. Bridge design pattern promotes this popular notion. Similar to the adapter pattern, this pattern also comes under the structural design pattern family of the GoF Design Pattern. The approach of the Bridge pattern is to decouple an abstraction used by the client code from its implementation; that means it separates the abstraction and its implementation into separate class hierarchies. And also, Bridge pattern prefers composition over inheritance because inheritance isn't always flexible and it breaks the encapsulation, so any change made in the implementer affects the abstraction used by the client code.

The bridge provides a way to communicate between two different independent components in software development, and a bridge structure provides you with a way to decouple the abstract class and the implementer class. So any change made in either the implementation class or the implementer (that is, the interface) doesn't affect the abstract class or its refined abstraction class. It makes this possible by using composition between the interface and the abstraction. Bridge pattern uses an interface as a bridge between the concrete classes of an abstract class and implementing classes of that interface. You can make changes in both types of class without any impact on the client code.

Benefits of the Bridge pattern

Following are the benefits of the Bridge design pattern:

- The Bridge design pattern allows you to separate the implementation and the abstraction
- This design pattern provides the flexibility to change both types of classes without side effects in the client code
- This design pattern allows the hiding of actual implementation details from the client by using abstraction between them

Common problems solved by the Bridge design pattern

Following are the common problems addressed by the Bridge design pattern:

- Removes a permanent binding between the functional abstraction and its implementation
- You can make changes to the implementing classes without affecting the abstraction and client code
- You can extend the abstraction and its implementation using subclasses

Implementing the Bridge design pattern in the Spring Framework

The following Spring modules are based on the Bridge design pattern:

- `ViewRendererServlet`: It is a bridge servlet, mainly for Portlet MVC support
- **The Bridge design pattern**: The Bridge design pattern is used in the Spring logging process

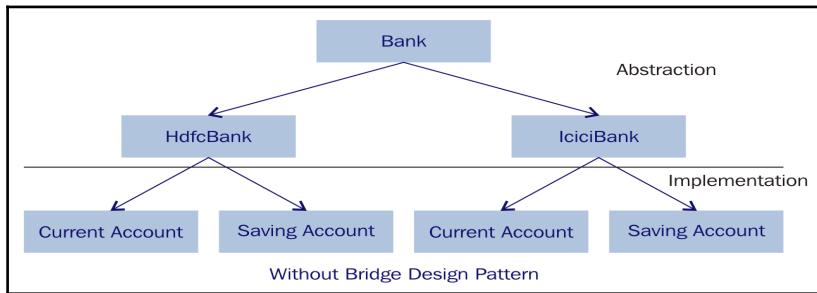
Let's see a sample implementation of the Bridge design pattern.

Sample implementation of the Bridge design pattern

Let's look at the following example, where we will demonstrate the use of the Bridge design pattern. Suppose you want to open two types of accounts, one is a Savings Account and the other is a Current Account in the banking system.

System without using the Bridge design pattern

Let's look at an example without using the Bridge design pattern. In the following figure, you can see the relationship between the Bank and Account interfaces:



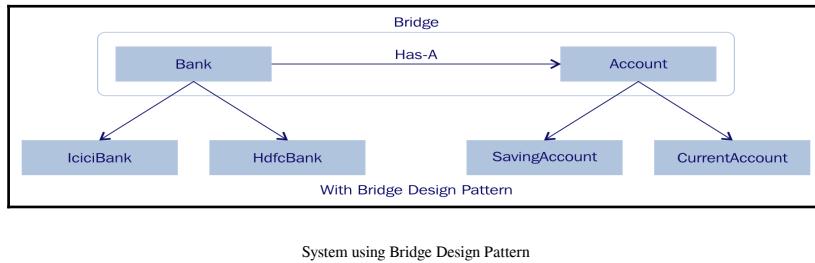
System without using the Bridge design pattern

Let's create a design without using the Bridge design pattern. First create an interface or an abstract class, **Bank**. And then create its derived classes: **IciciBank** and **HdfcBank**. To open an account in the bank, first decide on the types of account classes--**Saving Account** and **Current Account**, these classes extend the specific banks classes (**HdfcBank** and **IciciBank**). There is a simple deep inheritance hierarchy in this application. So what is wrong with this design as compared to the preceding figure? You will notice that in this design, there are two parts, one is the abstraction part and the other is the implementation part. Client code interacts with the abstraction part. Client code can only access new changes or new functionalities of the implementation part when you will update the abstraction part, meaning the parts, the abstraction, and the implementation, are tightly coupled with each other.

Now let's see how to improve this example using the Bridge design pattern:

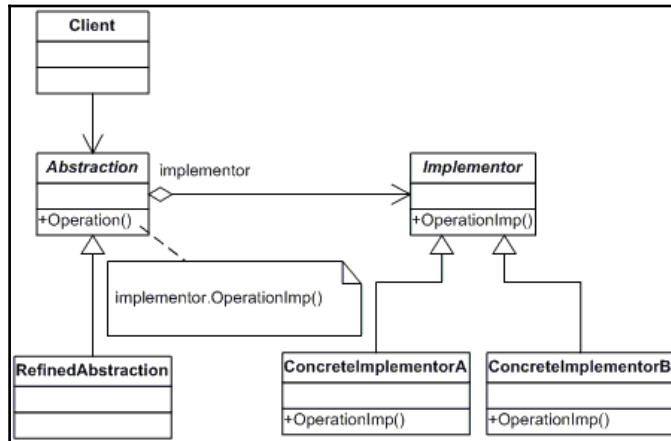
System with the Bridge design pattern

In the following figure, we create a relation between the Bank and Account interface by using the Bridge design pattern:



UML structure for the Bridge design pattern

Let's look at the following figure of how the Bridge design pattern solves these design issues, as seen in the example where we did not use the Bridge design pattern. Bridge pattern separates the abstraction and implementation into two class hierarchies:



We have an Account interface that is acting as a bridge implementer and the concrete classes SavingAccount, and CurrentAccount implementing the Account interface. The Bank is an abstract class and it will use object of Account.

Let's create a bridge implementer interface.

Following is the `Account.java` file:

```
package com.packt.patterninspring.chapter3.bridge.pattern;
public interface Account {
    Account openAccount();
    void accountType();
}
```

Create concrete bridge implementer classes to implement the `implementer` interface. Let's create a `SavingAccount` class as an implementation of `Account`.

Following is the `SavingAccount.java` file:

```
package com.packt.patterninspring.chapter3.bridge.pattern;
public class SavingAccount implements Account {
    @Override
    public Account openAccount() {
        System.out.println("OPENED: SAVING ACCOUNT ");
        return new SavingAccount();
    }
    @Override
    public void accountType() {
        System.out.println("##It is a SAVING Account##");
    }
}
```

Create a `CurrentAccount` class that implements the `Account` interface.

Following is the `CurrentAccount.java` file:

```
package com.packt.patterninspring.chapter3.bridge.pattern;
public class CurrentAccount implements Account {
    @Override
    public Account openAccount() {
        System.out.println("OPENED: CURRENT ACCOUNT ");
        return new CurrentAccount();
    }
    @Override
    public void accountType() {
        System.out.println("##It is a CURRENT Account##");
    }
}
```

Create abstraction in the Bridge design pattern, but first, create the interface **Bank**.

Following is the `Bank.java` file:

```
package com.packt.patterninspring.chapter3.bridge.pattern;
public abstract class Bank {
    //Composition with implementor
    protected Account account;
    public Bank(Account account) {
        this.account = account;
    }
    abstract Account openAccount();
}
```

Let's implement the first abstraction for the `Bank` interface and see the following implementation class for the `Bank` interface.

Following is the `IciciBank.java` file:

```
package com.packt.patterninspring.chapter3.bridge.pattern;
public class IciciBank extends Bank {
    public IciciBank(Account account) {
        super(account);
    }
    @Override
    Account openAccount() {
        System.out.print("Open your account with ICICI Bank");
        return account;
    }
}
```

Let's implement the second abstraction for the `Bank` interface and look at the following implementation class for the `Bank` interface.

Following is the `HdfcBank.java` file:

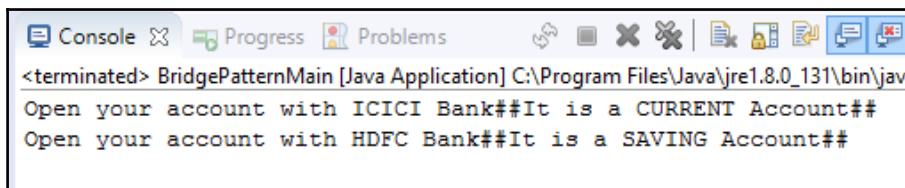
```
package com.packt.patterninspring.chapter3.bridge.pattern;
public class HdfcBank extends Bank {
    public HdfcBank(Account account) {
        super(account);
    }
    @Override
    Account openAccount() {
        System.out.print("Open your account with HDFC Bank");
        return account;
    }
}
```

Create a demonstration class of the Bridge design pattern.

Following is the `BridgePatternMain.java` file:

```
package com.packt.patterninspring.chapter3.bridge.pattern;
public class BridgePatternMain {
    public static void main(String[] args) {
        Bank icici = new IciciBank(new CurrentAccount());
        Account current = icici.openAccount();
        current.accountType();
        Bank hdfc = new HdfcBank(new SavingAccount());
        Account saving = hdfc.openAccount();
        saving.accountType();
    }
}
```

Let's run this demo class and see the following output in the console:



The screenshot shows a Java application window with tabs for 'Console', 'Progress', and 'Problems'. The 'Console' tab is active, displaying the output of a Java application named 'BridgePatternMain'. The output text is:
<terminated> BridgePatternMain [Java Application] C:\Program Files\Java\jre1.8.0_131\bin\jav
Open your account with ICICI Bank##It is a CURRENT Account##
Open your account with HDFC Bank##It is a SAVING Account##

Now that we've seen the Bridge design pattern, let's turn to a different variant of it--the composite design pattern.

Composite design pattern

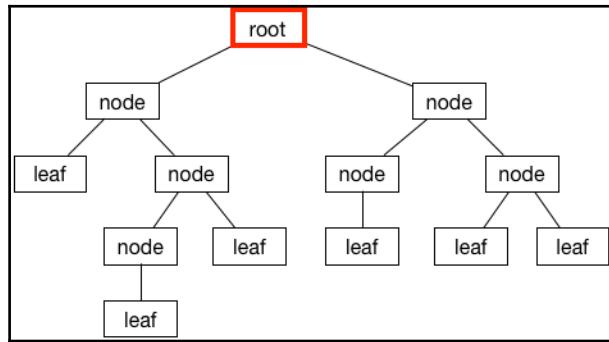
Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

-GoF Design Patterns

In software engineering, the composite pattern comes under the structural design pattern. According to this pattern, a group of objects of the same type are treated as a single object by the client. The idea behind the Composite design pattern is to compose a set of objects into a tree structure to represent a module of a larger structural application. And this structure for clients is a single unit or instance uniformly.

The motivation behind the Composite design pattern is that objects in the system are grouped into the tree structure, and a tree structure is a combination of the node-leaf and branches. In the tree structure, nodes have a number of leaves and other nodes. Leaf doesn't have anything, which means there is no child of leaf in the tree. Leaf is treated as the end point of tree-structured data.

Let's look at the following figure, which represents data in the tree structure in the form of node and leaf:



Tree structured data using nodes and leaves

Common problems solved by the composite pattern

As a developer, it is more difficult to design an application so that the client can access your objects uniformly across the application, even if that object was a composition of objects or an individual object. This design pattern resolves difficulties and allows you to design objects in such a way that you can use that object as a composition of objects and a single individual object.

This pattern solves the challenges faced when creating hierarchical tree structures to provide clients with a uniform way to access and manipulate objects in the tree. The composite pattern is a good choice; it is less complex in this situation to treat primitives and composites as homogeneous.

UML structure of the Composite design pattern

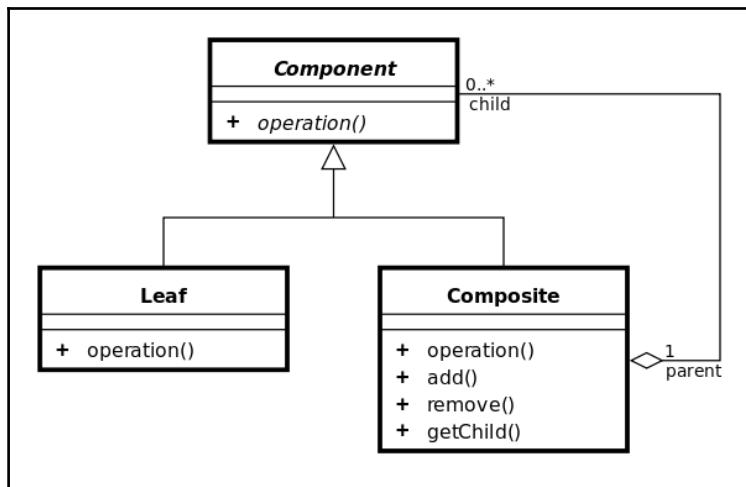
Composite design pattern is based on the composition of similar types of objects into the tree structure, as you know that each tree has three main parts branch, node, and leaf. So let's have a look at the following terms used in this design pattern.

Component: It is basically a branch of the tree and the branch has other branches, nodes, and leaves. Component provides the abstraction for all components, including composite objects. In the composition pattern, component is basically declared as an interface for objects.

Leaf: It is an object that implements all component methods.

Composite: It is represented as a node in the tree structure, it has other nodes and leaves, and it represents a composite component. It has methods to add the children, that is, it represents a collection of the same type of objects. It has other component methods for its children.

Let's look at the following UML diagram for this design pattern:



UML diagram for the Composite Design Pattern

Benefits of the Composite design pattern

- This pattern provides the flexibility to add new component to process dynamically, with change in the existing components
- This pattern allows you to create a class hierarchy that contains individual and composite objects

Sample implementation of the Composite design pattern

In the following example, I am implementing an `Account` interface, which can be either a `SavingAccount` and `CurrentAccount` or a composition of several accounts. I have a `CompositeBankAccount` class, which acts as a composite pattern actor class. Let's look at the following code for this example.

Create an `Account` interface that will be treated as a component:

```
public interface Account {  
    void accountType();  
}
```

Create a `SavingAccount` class and `CurrentAccount` class as an implementation of the component and that will also be treated as a leaf:

Following is the `SavingAccount.java` file:

```
public class SavingAccount implements Account {  
    @Override  
    public void accountType() {  
        System.out.println("SAVING ACCOUNT");  
    }  
}
```

Following is the `CurrentAccount.java` file:

```
public class CurrentAccount implements Account {  
    @Override  
    public void accountType() {  
        System.out.println("CURRENT ACCOUNT");  
    }  
}
```

Create a `CompositeBankAccount` class that will be treated as a Composite and implements the `Account` interface:

Following is the `CompositeBankAccount.java` file:

```
package com.packt.patterninspring.chapter3.composite.pattern;  
import java.util.ArrayList;  
import java.util.List;  
import com.packt.patterninspring.chapter3.model.Account;  
public class CompositeBankAccount implements Account {  
    //Collection of child accounts.  
    private List<Account> childAccounts = new ArrayList<Account>();  
    @Override
```

```
public void accountType() {
    for (Account account : childAccounts) {
        account.accountType();
    }
}
//Adds the account to the composition.
public void add(Account account) {
    childAccounts.add(account);
}
//Removes the account from the composition.
public void remove(Account account) {
    childAccounts.remove(account);
}
}
```

Create a `CompositePatternMain` class that will also be treated as a Client:

Following is the `CompositePatternMain.java` file:

```
package com.packt.patterninspring.chapter3.composite.pattern;
import com.packt.patterninspring.chapter3.model.CurrentAccount;
import com.packt.patterninspring.chapter3.model.SavingAccount;
public class CompositePatternMain {
    public static void main(String[] args) {
        //Saving Accounts
        SavingAccount savingAccount1 = new SavingAccount();
        SavingAccount savingAccount2 = new SavingAccount();
        //Current Account
        CurrentAccount currentAccount1 = new CurrentAccount();
        CurrentAccount currentAccount2 = new CurrentAccount();
        //Composite Bank Account
        CompositeBankAccount compositeBankAccount1 = new
        CompositeBankAccount();
        CompositeBankAccount compositeBankAccount2 = new
        CompositeBankAccount();
        CompositeBankAccount compositeBankAccount = new
        CompositeBankAccount();
        //Composing the bank accounts
        compositeBankAccount1.add(savingAccount1);
        compositeBankAccount1.add(currentAccount1);
        compositeBankAccount2.add(currentAccount2);
        compositeBankAccount2.add(savingAccount2);
        compositeBankAccount.add(compositeBankAccount2);
        compositeBankAccount.add(compositeBankAccount1);
        compositeBankAccount.accountType();
    }
}
```

Let's run this demo class and see the following output at the console:

```
Console Progress Problems
<terminated> CompositePatternMain [Java]
CURRENT ACCOUNT
SAVING ACCOUNT
SAVING ACCOUNT
CURRENT ACCOUNT
```

Now that we have discussed the composite design pattern, let's turn to the decorator design pattern.

Decorator design pattern

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub classing for extending functionality.
- GOF Design Pattern

In software engineering, the common intent of all GOF structural patterns is to simplify the complex relationship between objects and classes in a flexible enterprise application. The decorator pattern is a special type of design pattern among these that comes under the structural design pattern, which allows you to add and remove behaviors for an individual object at runtime dynamically or statically, without changing the existing behavior of other associated objects from the same class. This design pattern does this without violating the Single Responsibility Principle or the SOLID principle of object-oriented programming.

This design pattern uses the compositions over the inheritance for objects associations; it allows you to divide the functionality into different concrete classes with a unique area of concern.

Benefits of the Decorator design pattern

- This pattern allows you to extend functionality dynamically and statically without altering the structure of existing objects
- By using this pattern, you could add a new responsibility to an object dynamically
- This pattern is also known as **Wrapper**

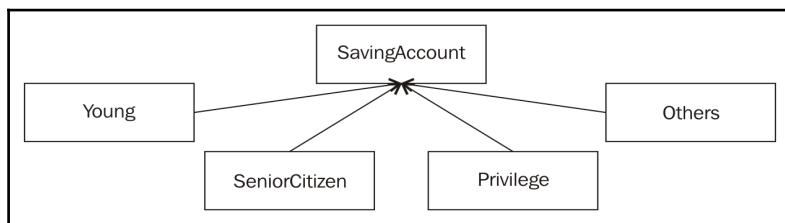
- This pattern uses the compositions for object relationships to maintain SOLID principles
- This pattern simplifies coding by writing new classes for every new specific functionality rather than changing the existing code of your application

Common problems solved by the Decorator pattern

In an enterprise application, there is a business requirement or there might be a future plan to extend the behavior of the product by adding new functionalities. To achieve this, you could use inheritance to extend the behavior of an object. But inheritance should be done at compile time and methods are also available for other instances of that class. Because of the code modification, there is a violation of the Open Closed Principle. To avoid this violation of the SOLID principle, you can attach new responsibility to an object dynamically. This is the situation where the decorator design pattern comes into the picture and addresses this issue in a very flexible way. Let's look at the following example of how to implement this design pattern into a real case study.

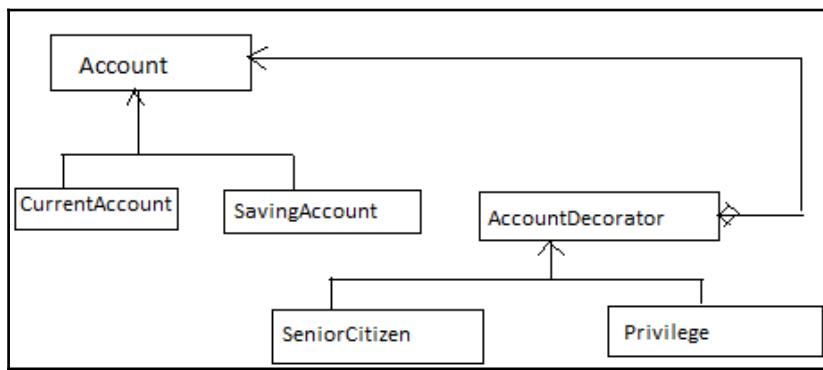
Consider that a bank offers multiple accounts with different benefits to customers. It divides the customers into three categories--senior citizens, privileged, and young. The bank launches a scheme on the savings account for senior citizens--if they open a savings account in this bank, they will be provided medical insurance of up to \$1,000. Similarly, the bank also provides a scheme for the privileged customers as an accident insurance of up to \$1,600 and an overdraft facility of \$84. There is no scheme for the young.

To address the new requirement, we can add new subclasses of `SavingAccount`; one each to represent a saving account with additional benefits as decoration, and this is what our design looks like now:



Application design with inheritance without using the Decorator Design Pattern

This design will be very complex as I will add more benefit schemes to the **SavingAccount**, but what would happen when the bank launches the same scheme for **CurrentAccount**? Clearly, this design is flawed, but this is an ideal use case for the decorator pattern. This pattern allows you to add runtime dynamic behavior. In this case, I will create an abstract **AccountDecorator** class to implement **Account**. And furthermore, I will create the **SeniorCitizen** class and **Privilege** class, which extends **AccountDecorator** because young does not have any extra benefits, so the **SavingAccount** class does not extend **AccountDecorator**. This is how the design will be:

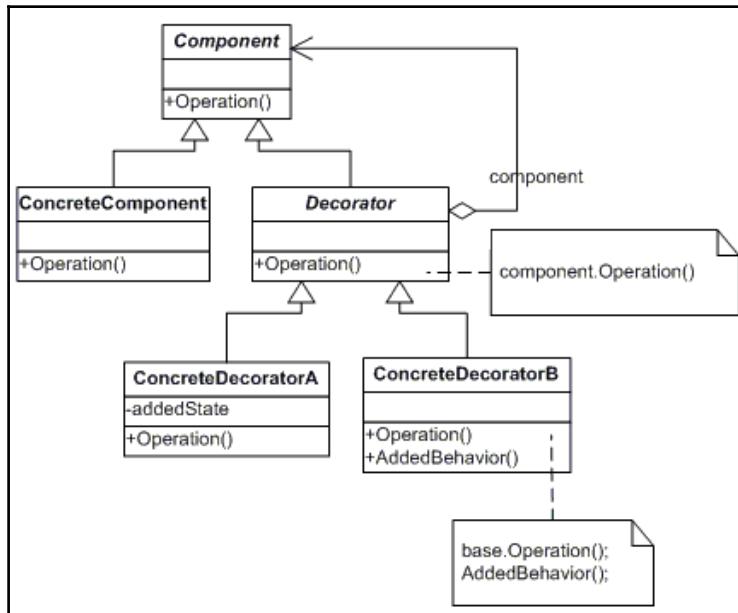


Application design with composition using the decorator design pattern

The preceding figure follows the Decorator design pattern by creating **AccountDecorator** as a **Decorator** in this pattern, and focuses on important things to observe the relationship between **Account** and **AccountDecorator**. This relationship is as follows:

- **Is-a** relationship between the **AccountDecorator** and **Account**, that is, inheritance for the correct type
- **Has-a** relationship between the **AccountDecorator** and **Account**, that is, composition in order to add new behavior without changing the existing code

Let's look at the UML structure:



UML for the Decorator design pattern

The classes and objects participating in this pattern are:

- **Component (Account)**: It is an interface for objects that can have responsibilities added to them dynamically
- **ConcreteComponent (SavingAccount)**: It is a concrete class of component interface and it defines an object to which additional responsibilities can be attached
- **Decorator (AccountDecorator)**: It has a reference to a **Component** object and defines an interface that conforms to the interface of the component
- **ConcreteDecorator (SeniorCitizen and Privilege)**: It is a concrete implementation of **Decorator** and it adds responsibilities to the component

Implementing the Decorator pattern

Let's look at the following code to demonstrate the Decorator design pattern.

Create a component class:

Following is the `Account.java` file:

```
package com.packt.patterninspring.chapter3.decorator.pattern;
public interface Account {
    String getTotalBenefits();
}
```

Create concrete components classes:

Following is the `SavingAccount.java` file:

```
package com.packt.patterninspring.chapter3.decorator.pattern;
public class SavingAccount implements Account {
    @Override
    public String getTotalBenefits() {
        return "This account has 4% interest rate with per day
               $5000 withdrawal limit";
    }
}
```

Let's create another concrete class for `Account` component:

Following is the `CurrentAccount.java` file:

```
package com.packt.patterninspring.chapter3.decorator.pattern;
public class CurrentAccount implements Account {
    @Override
    public String getTotalBenefits() {
        return "There is no withdrawal limit for current account";
    }
}
```

Let's create a `Decorator` class for `Account` component. This decorator class apply other run time behavior to the `Account` component classes.

Following is the `AccountDecorator.java` file:

```
package com.packt.patterninspring.chapter3.decorator.pattern;
public abstract class AccountDecorator implements Account {
    abstract String applyOtherBenefits();
}
```

Let's create a `ConcreteDecorator` class to implement the `AccountDecorator` class. Following class `SeniorCitizen` is extended `AccountDecorator` class to access other run time behavior such as `applyOtherBenefits()`.

Following is the `SeniorCitizen.java` file:

```
package com.packt.patterninspring.chapter3.decorator.pattern;
public class SeniorCitizen extends AccountDecorator {
    Account account;
    public SeniorCitizen(Account account) {
        super();
        this.account = account;
    }
    public String getTotalBenefits() {
        return account.getTotalBenefits() + " other benefits are
               "+applyOtherBenefits();
    }
    String applyOtherBenefits() {
        return " an medical insurance of up to $1,000 for Senior
               Citizen";
    }
}
```

Let's create another `ConcreteDecorator` class to implement the `AccountDecorator` class. Following class `Privilege` is extended `AccountDecorator` class to access other run time behavior such as `applyOtherBenefits()`.

Following is the `Privilege.java` file:

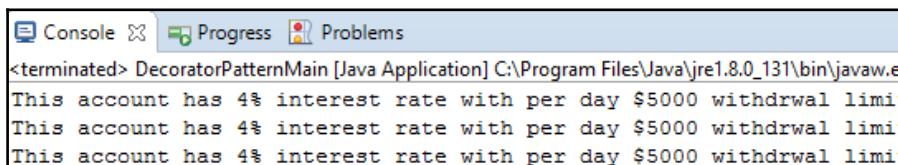
```
package com.packt.patterninspring.chapter3.decorator.pattern;
public class Privilege extends AccountDecorator {
    Account account;
    public Privilege(Account account) {
        this.account = account;
    }
    public String getTotalBenefits() {
        return account.getTotalBenefits() + " other benefits are
               "+applyOtherBenefits();
    }
    String applyOtherBenefits() {
        return " an accident insurance of up to $1,600 and
               an overdraft facility of $84";
    }
}
```

Let's now write some test code to see how the Decorator pattern works at runtime:

Following is the `DecoratorPatternMain.java` file:

```
package com.packt.patterninspring.chapter3.decorator.pattern;
public class DecoratorPatternMain {
    public static void main(String[] args) {
        /*Saving account with no decoration*/
        Account basicSavingAccount = new SavingAccount();
        System.out.println(basicSavingAccount.getTotalBenefits());
        /*Saving account with senior citizen benefits decoration*/
        Account seniorCitizenSavingAccount = new SavingAccount();
        seniorCitizenSavingAccount = new
            SeniorCitizen(seniorCitizenSavingAccount);
        System.out.println
        (seniorCitizenSavingAccount.getTotalBenefits());
        /*Saving account with privilege decoration*/
        Account privilegeCitizenSavingAccount = new SavingAccount();
        privilegeCitizenSavingAccount = new
            Privilege(privilegeCitizenSavingAccount);
        System.out.println
        (privilegeCitizenSavingAccount.getTotalBenefits());
    }
}
```

Let's run this demo class and see the following output at the console:



The screenshot shows a Java application running in an IDE. The title bar says "Console" and "DecoractorPatternMain [Java Application] C:\Program Files\Java\jre1.8.0_131\bin\javaw.exe". The output window displays three lines of text: "This account has 4% interest rate with per day \$5000 withdraw limit", "This account has 4% interest rate with per day \$5000 withdraw limit", and "This account has 4% interest rate with per day \$5000 withdraw limit".

Decorator design pattern in the Spring Framework

The Spring Framework uses the Decorator design pattern to build important functionalities such as transactions, cache synchronization, and security-related tasks. Let's look at some functionalities where Spring implements this pattern transparently:

- Weaving the advice into the Spring application. It uses the Decorator pattern via the CGLib proxy. It works by generating a subclass of the target class at runtime.

- `BeanDefinitionDecorator`: It is used to decorate the bean definition via applied custom attributes.
- `WebSocketHandlerDecorator`: It is used to decorate a `WebSocketHandler` with additional behaviors.

Now let's turn to another GOF Design Pattern - Facade design pattern.

Facade Design Pattern

Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- GOF Design Patterns

The Facade design pattern is nothing but an interface of interfaces to simplify interactions between the client code and subsystem classes. This design comes under the GOF structural design pattern.

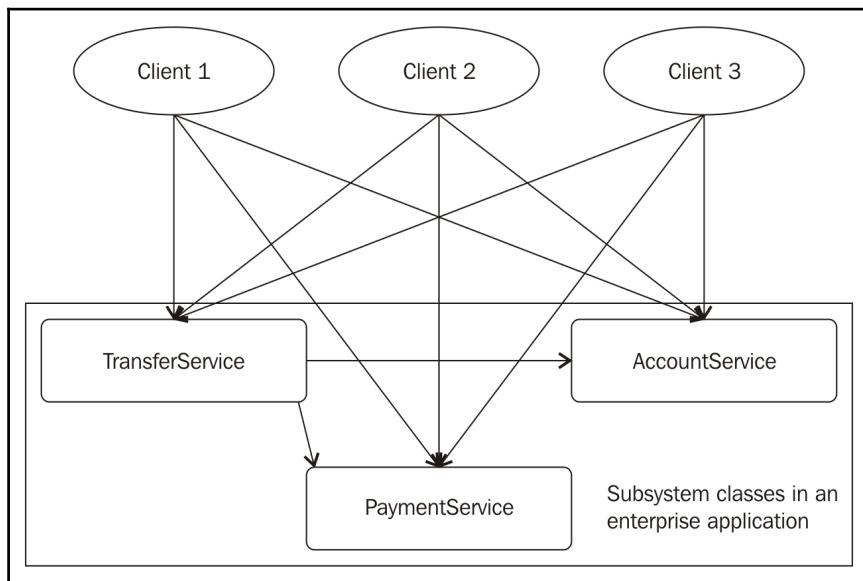
Benefits of Facade Pattern:

- This pattern reduces the complexities for clients to interact with subsystems
- This pattern consolidates all the business services as single interfaces to make them more understandable
- This pattern reduces dependencies of client code on the inner workings of a system

Knowing when to use the Facade Pattern

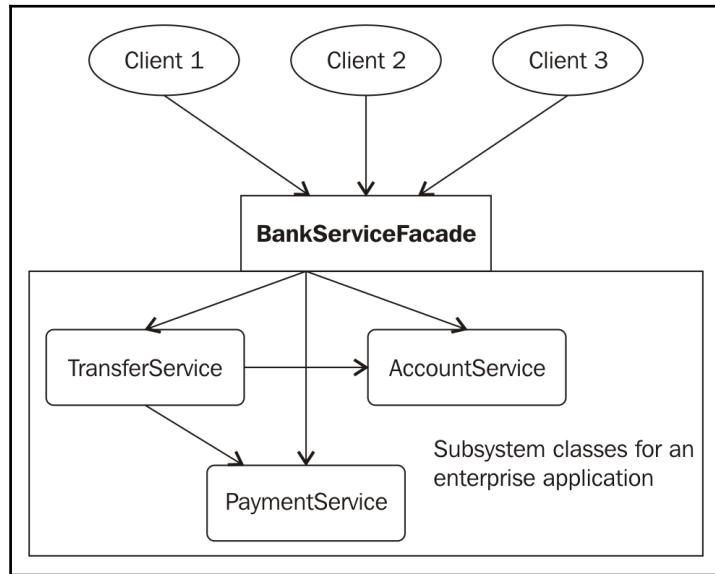
Suppose you are designing a system, and this system has a very large number of independent classes and also has a set of services to be implemented. This system is going to be very complex, so the Facade pattern comes into the picture and reduces the complexities of the larger system and simplifies interactions of the client code with a set of classes from a subsystem of the large complex system.

Suppose you want to develop a bank enterprise application with a large number of services to perform a task, for example, `AccountService` for getting the `Account` by `accountId`, `PaymentService` for payment gateway services, and `TransferService` for the amount transfer from one account to another account. A client code of the application interacts with all these services to transfer money from one account to another account. This is how different clients interact with the amount transfer process of the bank system. As shown in the following figure, here you can see client code that directly interacts with the subsystem classes and client also should aware about the internal working of subsystem classes, so it is simply a violation of the SOLID design principles because client code is tightly coupled with the classes of subsystem of your banking application:



Banking Application Subsystem without Facade Design Pattern

Rather than client code directly interacting with the classes of a subsystem, you could introduce one more interface, which makes the subsystems easier to use, as shown in the following figure. This interface is known as a Facade interface, it is based on the Facade pattern, and it is a simple way to interact with the subsystems:



Banking Application Subsystem with Facade design pattern

Implementing the Facade design pattern

Let's look into the following listings to demonstrate the Facade design pattern.

Create subsystem service classes for your Bank application: Let's see the following PaymentService class for the subsystem.

Following is the `PaymentService.java` file:

```
package com.packt.patterninspring.chapter3.facade.pattern;
public class PaymentService {
    public static boolean doPayment() {
        return true;
    }
}
```

Let's create another service class AccountService for the subsystem.

Following is the `AccountService.java` file:

```
package com.packt.patterninspring.chapter3.facade.pattern;
import com.packt.patterninspring.chapter3.model.Account;
import com.packt.patterninspring.chapter3.model.SavingAccount;
public class AccountService {
    public static Account getAccount(String accountId) {
        return new SavingAccount();
    }
}
```

Let's create another service class TransferService for the subsystem.

Following is the `TransferService.java` file:

```
package com.packt.patterninspring.chapter3.facade.pattern;
import com.packt.patterninspring.chapter3.model.Account;
public class TransferService {
    public static void transfer(int amount, Account fromAccount,
                               Account toAccount) {
        System.out.println("Transferring Money");
    }
}
```

Create a Facade Service class to interact with the subsystem: Let's see the following Facade interface for the subsystem and then implement this Facade interface as a global banking service in the application.

Following is the `BankingServiceFacade.java` file:

```
package com.packt.patterninspring.chapter3.facade.pattern;
public interface BankingServiceFacade {
    void moneyTransfer();
}
```

Following is the `BankingServiceFacadeImpl.java` file:

```
package com.packt.patterninspring.chapter3.facade.pattern;
import com.packt.patterninspring.chapter3.model.Account;
public class BankingServiceFacadeImpl implements
    BankingServiceFacade{
    @Override
    public void moneyTransfer() {
        if(PaymentService.doPayment()) {
            Account fromAccount = AccountService.getAccount("1");
            Account toAccount = AccountService.getAccount("2");
            TransferService.transfer(1000, fromAccount, toAccount);
        }
    }
}
```

Create the client of the Facade:

Following is the `FacadePatternClient.java` file:

```
package com.packt.patterninspring.chapter3.facade.pattern;
public class FacadePatternClient {
    public static void main(String[] args) {
        BankingServiceFacade serviceFacade = new
            BankingServiceFacadeImpl();
        serviceFacade.moneyTransfer();
    }
}
```

The UML structure for the Facade design pattern

The classes and objects participating in this pattern are:

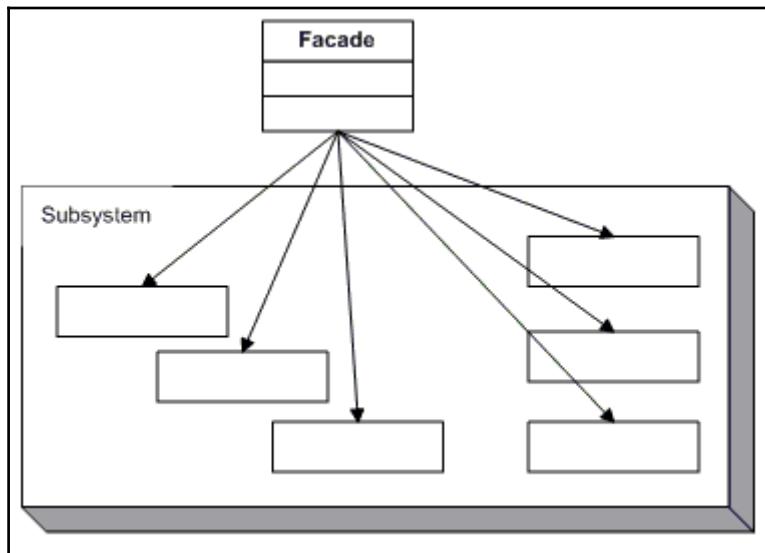
- Facade (`BankingServiceFacade`)

This is a Facade interface that knows which subsystem classes are responsible for a request. This interface is responsible for delegating client requests to appropriate subsystem objects.

- Subsystem classes (`AccountService`, `TransferService`, `PaymentService`)

These interfaces are actually subsystem functionalities of the banking process system application. These are responsible for handling processes assigned by the Facade object. No interfaces in this category have a reference to the Facade object; they don't have implementation details of Facade. These are totally independent of Facade objects.

Let's see the following UML diagram for this pattern:



UML diagram for Facade design pattern

Facade Pattern in the Spring Framework

In the enterprise application, if you are working in Spring applications, the facade pattern is used commonly in the business service layer of the application to consolidate all services. And you could also apply this pattern on DAOs on the persistent layer.

Now that we've seen the Facade design pattern, let's turn to a different variant of it--Proxy design pattern.

Proxy design pattern

Provide a surrogate or placeholder for another object to control access to it.
-GOF Design Patterns

Proxy design pattern provides an object of a class with the functionality of another class with having it. This pattern comes under the structural design pattern of GOF Design Patterns. The intent of this design pattern is to provide an alternate class for another class , along with its functionality, to the outside world.

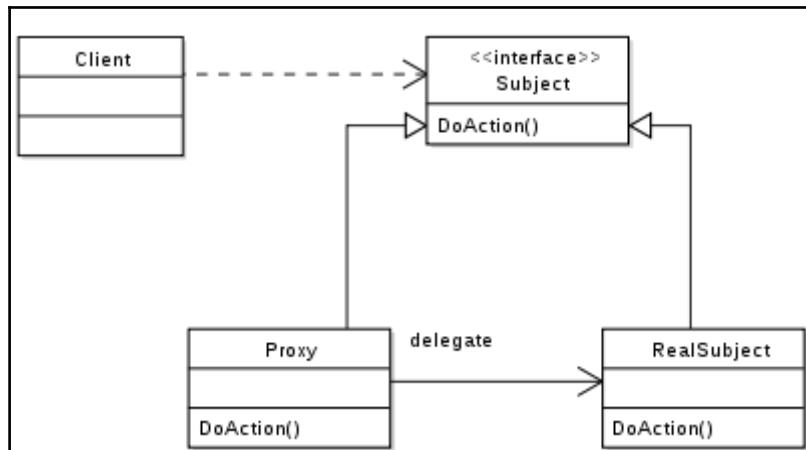
Purpose of the Proxy pattern

Let's look at the following points:

- This pattern hides the actual object from the outside world.
- This pattern can improve the performance because it is creating an object on demand.

UML structure for the Proxy design pattern

Let's see the following UML diagram for this pattern:



UML diagram for Proxy design pattern

Now let's look at the different components of this UML diagram:

- **Subject:** Actual interface to be implemented by Proxy and RealSubject.
- **RealSubject:** Real implementation of **Subject**. It is a real object that represented by the proxy.
- **Proxy:** It is a proxy object and it is also the implementation of the real object **Subject**. It maintains the references to the real object.

Implementing the Proxy design pattern

Let's look into following code to demonstrate the Proxy pattern.

Create a Subject.

Following is the `Account.java` file:

```
public interface Account {  
    void accountType();  
}
```

Create a RealSubject class that implements Subject, let's see the following class as RealSubject class for the Proxy design pattern.

Following is the `SavingAccount.java` file:

```
public class SavingAccount implements Account{  
    public void accountType() {  
        System.out.println("SAVING ACCOUNT");  
    }  
}
```

Create a Proxy class which implements Subject and having the Real Subject

Following is the `ProxySavingAccount.java` file:

```
package com.packt.patterninspring.chapter2.proxy.pattern;
import com.packt.patterninspring.chapter2.model.Account;
import com.packt.patterninspring.chapter2.model.SavingAccount;
public class ProxySavingAccount implements Account{
    private Account savingAccount;
    public void accountType() {
        if(savingAccount == null){
            savingAccount = new SavingAccount();
        }
        savingAccount.accountType();
    }
}
```

Proxy pattern in the Spring Framework

Spring Framework uses the Proxy design pattern in the Spring AOP module transparently. As I have discussed in *Chapter 1, Getting Started with Spring Framework 5.0 and Design Patterns*. In Spring AOP, you create proxies of the object to apply cross cutting concern across the point cut in the Spring application. In the Spring, other modules also implement the Proxy pattern, such as RMI, Spring's HTTP Invoker, Hessian, and Burlap.

Let's see the next section about Behavioral design pattern with its underlying patterns and example.

Behavioral design patterns

The intent of Behavioral design pattern is the interaction and cooperation between a set of objects to perform a task that no single object can carry out by itself. The interaction between the objects should be such that they should be loosely coupled. Patterns under this category, characterize the ways in which classes or objects interact and distribute responsibility. Let's see in the next sections, different variants of the Behavioral design patterns.

Chain of Responsibility design pattern

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

-GOF Design Patterns

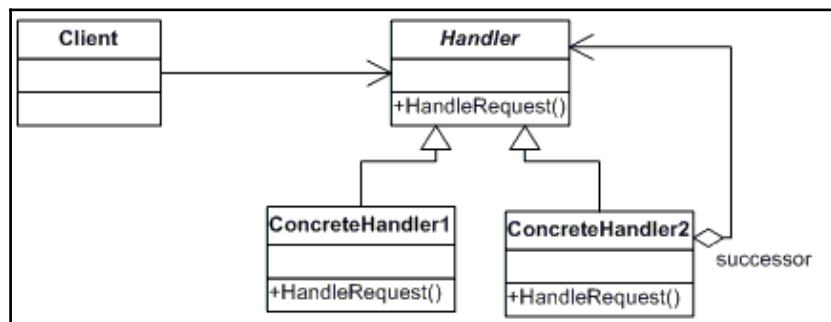
Chain of responsibility design pattern comes under the Behavioral design pattern of GOF patterns family. According to this pattern, sender and receiver of a request are decoupled. The sender sends a request to the chain of receivers and any one of receivers in the chain can handle the request. In this pattern, the receiver object has the reference of another receiver object so that if it does not handle the request then it passes the same request to the other receiver object.

For example, in a banking System, you could use any ATM to withdraw the money in any place, so it is one of the live examples of the Chain of Responsibility design pattern.

There are following benefits of this pattern:

- This pattern reduces the coupling between sender and receiver objects in the system to handle a request.
- This pattern is more flexible to assign the responsibility to another referenced object.
- This pattern makes a chain of objects using composition, and this set of objects work as a single unit.

Let's see the following UML diagram showing all components of a chain of responsibility design pattern:



UML Diagram for Chain of Responsibility design pattern

- **Handler:** This is an abstract class or interface in the system to handle request.
- **ConcreteHandler:** These are concrete classes which implement **Handler** to handle the request, or it passes same request to the next successor of the handler chain.
- **Client:** It is main application class to initiate the request to the handler objects on the chain.

Chain of Responsibility pattern in the Spring Framework

Spring Security project implemented the Chain of Responsibility pattern in the Spring Framework. Spring Security allows you to implement authentication and authorization functionality in your application by using chains of security filters. This is a highly configurable framework. You can add your custom filter with this chain of filters to customize the functionality because of Chain of Responsibility design pattern.

Now that we've seen the Chain of responsibility design pattern, let's turn to a different variant of it--Command design pattern.

Command design pattern

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations

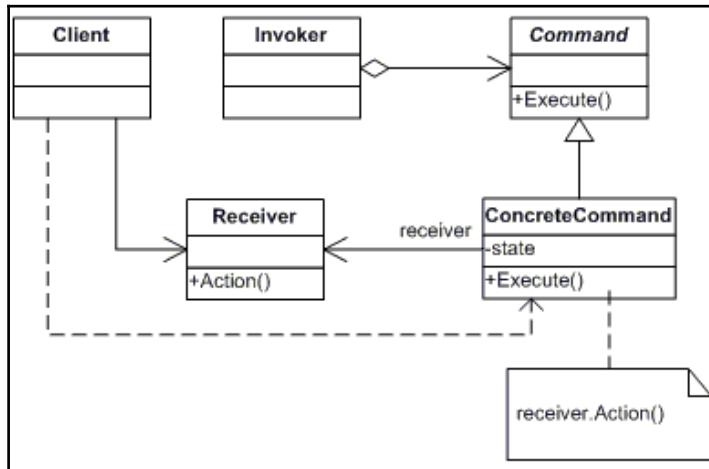
-GOF Design Patterns

The Command design pattern falls under the Behavioral pattern family of the GOF patterns, this pattern is a very simple data-driven pattern which allows you to encapsulate your request data into an object and pass that object as a command to the invoker method, and it return the command as another object to the caller.

The following lists the benefits of using the Command pattern:

- This pattern enables you to transfer data as an object between the system components sender and receiver.
- This pattern allows you to parameterize objects by an action to perform.
- You could easily add new commands in the system without changing existing classes.

Let's look at the following UML diagram showing all components of Command design pattern:



UML Diagram for Command Design Pattern

- **Command:** It is an interface or abstract class having an action to perform in the system.
- **ConcreteCommand:** It is a concrete implementation of the Command interface and defining an action will be performed.
- **Client:** This is a main class, it creates a ConcreteCommand object and sets its receiver.
- **Invoker:** It is a caller to invoke the request to carry the command object.
- **Receiver:** It is simple handler method which performs the actual operation by ConcreteCommand.

Command design pattern in the Spring Framework

Spring MVC has implemented the Command design pattern in the Spring Framework. In your enterprise applications using the Spring Framework, you often see the concepts of the Command pattern applied through the use of Command objects.

Now that we've seen Command design pattern, let's turn to a different variant of it--Interpreter design pattern.

Interpreter Design pattern

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

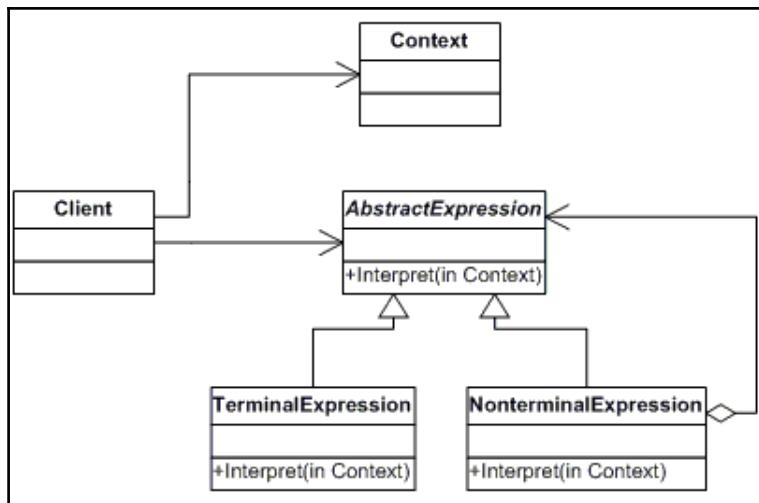
-GOF Design Pattern

Interpreter design pattern allows you to interpret an expression language in the programming to define a representation for its grammar. This type of a pattern comes under the Behavioral design pattern family of GOF patterns.

The following lists the benefits of using the Interpreter pattern:

- This pattern allows you to change and extend the grammar easily.
- Using the expression language is very easy

Let's see the following UML diagram is showing all components of Interpreter design pattern:



UML diagram for Interpreter design pattern

- **AbstractExpression**: It is an interface to execute a task by using `interpret()` operation.
- **TerminalExpression**: It is an implementation of above interface and it implements `interpret()` operation for terminal expressions.

- **NonterminalExpression:** It is also an implementation of above interface and it implements `interpret()` operation for non-terminal expressions.
- **Context:** It is a `String` expression and contains information that is global to the interpreter.
- **Client:** It is the main class to invoke the `Interpret` operation.

Interpreter design pattern in the Spring Framework

In the Spring Framework, Interpreter pattern is used with the **Spring Expression Language (SpEL)**. Spring added this new feature from Spring 3.0, you can use it in your enterprise application using the Spring Framework.

Now that we've seen Interpreter design pattern, let's turn to a different variant of it-- Iterator design pattern.

Iterator Design Pattern

Provide a way to access the elements of an aggregate object sequentially without Exposing its underlying representation.

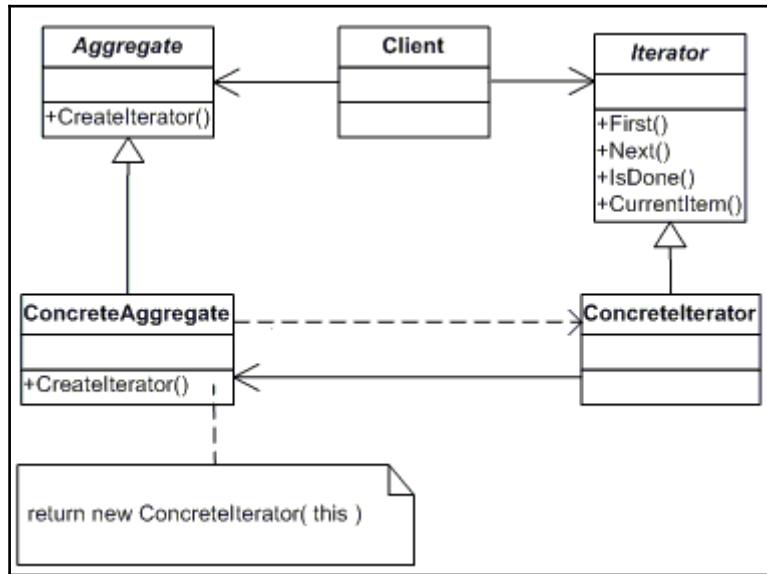
-GOF Design Pattern

This is a very commonly used design pattern in the programming language as like in Java. This pattern comes from the Behavioral Design Pattern family of GOF pattern. This pattern allows you to access the items from the collection object in sequence without information its internal representation.

These are following benefits of the Iterator pattern:

- Easily access the items of the collection.
- You can use multiple to access the item from the collection because it support lot of variations in the traversal.
- It provides a uniform interface for traversing different structures in a collection.

Let's see the following UML diagram is showing all components of Iterator design pattern:



UML Diagram for Iterator Design Pattern

- **Iterator:** It is an interface or abstract class for accessing and traversing items of the collections.
- **ConcreteIterator:** It is an implementation of the **Iterator** interface.
- **Aggregate:** It is an interface to create an Iterator object.
- **ConcreteAggregate:** It is the implementation of the **Aggregate** interface, it implements the **Iterator** creation interface to return an instance of the proper **ConcreteIterator**.

Iterator design pattern in the Spring Framework

The Spring Framework also extends the Iterator pattern through the **CompositeIterator** class. Mainly this pattern used in the Collection Framework of Java for iterating the elements in sequence.

Now that we've seen Iterator design pattern, let's turn to a different variant of it--Observer design pattern.

Observer Design Pattern

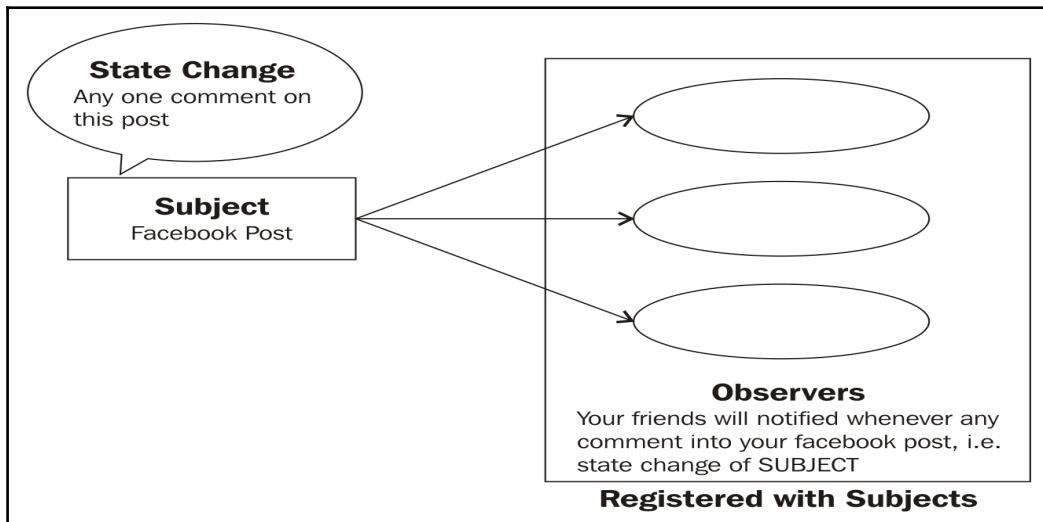
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically

-GOF Design Pattern

Observer pattern is one of very common design pattern. This pattern is a part of the Behavioral design pattern family of GOF pattern that addresses responsibilities of objects in an application and how they communicate between them at runtime. According to this pattern, sometimes objects make a one-to-many relationship between the objects in your application, such that if one object is modified, it's notified to other dependent objects automatically.

For example, Facebook post comments are one of the examples of the observer design pattern. If you comment on a post of your friend then you are always notified by this post whenever anyone else comments on the same post again.

The Observer pattern provides communication between decoupled objects. It makes a relationship between objects mostly a one-to-many relationship. In this pattern, there is an object which is known as the subject. Whenever there is any change in the state of this subject, it will be notified to its list of dependents accordingly. This list of dependents is known as observers. The following figure illustrates the Observer pattern:

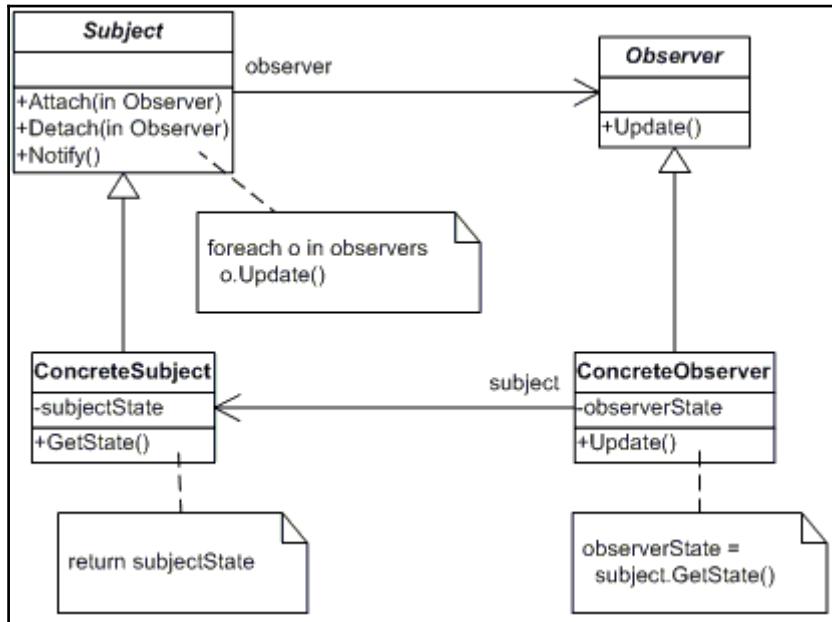


Use case of the Observer design pattern

There are following lists of the benefits of using the Observer pattern:

- This pattern provides decoupled relationship between the subject and observer
- It provides support for broadcasting

Let's see the following UML diagram is showing all components of Observer design pattern:



UML Diagram for Observer Design Pattern

- **Subject**: It is an interface. It has information about its observers.
- **ConcreteSubject**: It is a concrete implementation of **Subject**, it has information about all its observers to be notified when its state changes.
- **Observer**: It is an interface to be notified of changes in a subject.
- **ConcreteObserver**: It is a concrete implementation of **Observer**, it keeps its state consistent with the subject's state.

Observer pattern in the Spring Framework

In the Spring Framework, the Observer design pattern is used to implement event handling function of ApplicationContext. Spring provides us the ApplicationEvent class and ApplicationListener interface to enable event handling in Spring ApplicationContext. Any bean in your Spring application implements the ApplicationListener interface, it will receive an ApplicationEvent every time the ApplicationEvent is published by an event publisher. Here, the event publisher is the subject and the bean that implements ApplicationListener is the observer.

Now that we've seen Observer design pattern, let's turn to a different variant of it--Template design pattern.

Template Design Pattern

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

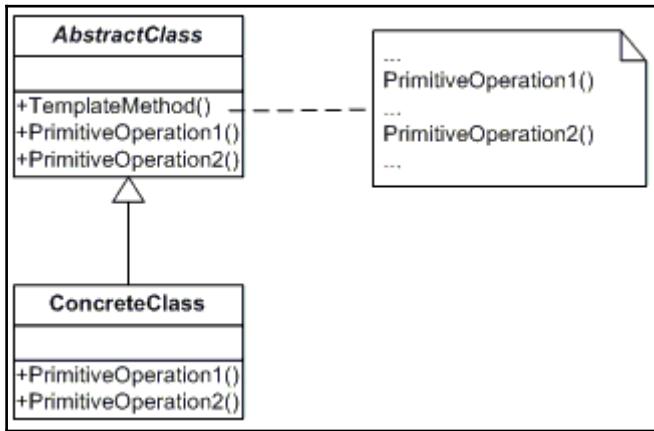
60; -GOF Design Patterns

In Template design pattern, an abstract class wraps some defined ways to its method. That method allows you to override parts of the method without rewriting it. You could use its concrete class to your application to perform similar type actions. This design pattern comes under the Behavior design pattern family of GOF pattern.

There are following lists the benefits of using the Template pattern:

- It reduces the boilerplate codes in the application by reusing code.
- This pattern creates a template or way to reuse multiple similar algorithms to perform some business requirements.

Let's see the following UML diagram is showing the components of Template design pattern:



UML Diagram for Template design pattern

- **AbstractClass:** This is an abstract class that contains a template method defining the skeleton of an algorithm.
- **ConcreteClass:** This is a concrete subclass of **AbstractClass** that implements the operations to carry out the algorithm-specific primitive steps.

Let's see the next section about J2EE design patterns in the enterprise distributed applications

JEE design patterns

It is other main category of design patterns. Application design can be immensely simplified by applying Java EE design patterns. Java EE design patterns have been documented in Sun's Java Blueprints. These Java EE design patterns provide time-tested solution guidelines and best practices for object interaction in the different layer of a Java EE application. These design patterns are specifically concerned with the following listed layers:

- Design pattern at presentation layer
- Design pattern at business layer
- Design pattern at integration layer

These design patterns are specifically concerned with the following listed layers.

- **Design pattern at presentation layer:**
 - **View Helper:** It separates views from the business logic of an enterprise J2EE application.
 - **Front Controller:** It provides a single point of action to handle all coming requests to the J2EE web application, it forwards the request to specific application controller to access model and view for presentation tier resources.
 - **Application Controller-**The request is actually handled by the Application Controller, it acts as a front controller helper. It is responsible for the coordination with the business models and view components.
 - **Dispatcher View-**It is related to the view only and it executes without business logic to prepare a response to the next view.
 - **Intercepting filters** -In the J2EE web application, you could configure multiple interceptors for pre and post processing an user's request such as tracking and auditing user's requests.
- **Design pattern at business layer:**
 - **Business Delegate-**It acts as a bridge between application controllers and business logic
 - **Application Service-**It provides business logics to implement the model as simple Java objects for presentation layer
- **Design pattern at integration layer:**
 - **Data Access Object-**It is implemented for accessing business data and it separates data access logic from business logic in the enterprise application.
 - **Web Service Broker-**It encapsulates the logic to access the external application's resources and it is exposed as web services.

Summary

After reading this chapter, the reader should now have a good idea about GOF Design Patterns and their best practices. I highlighted the problems that come if you don't implement design patterns in your enterprise application and how Spring solves these problems by using lots of design patterns and good practices to create an application. In the preceding chapter too, I have mentioned the three main categories of GOF Design Patterns such as Creational Design Pattern; it is useful for creation of object instances and also to apply some constraints at the creation time of the enterprise application by specific manner by Factory, Abstract Factory, Builder, Prototype and Singleton pattern. The second main category is the Structural design pattern, it is used for design structure of the enterprise application by dealing with the composition of classes or objects so that it reduces application complexity and improve the reusability and performance of the application. They are Adapter pattern, Bridge pattern, Composite pattern, Decorator pattern, and Facade pattern come under this category of patterns. Finally, one more main category of the pattern is Behavioral design pattern, it characterizes the ways in which classes or objects interact and distribute responsibility. Patterns come under this category are specifically concerned with communication between objects.

11

Wiring Beans using the Dependency Injection Pattern

In the previous chapter, you learned about the **Gang of Four (GOF)** design patterns with examples and use cases of each. Now, we will go into more detail about injecting beans and the configuration of dependencies in a Spring application, where you will see the various ways of configuring dependencies in a Spring application. This includes configuration with XML, Annotation, Java, and Mix.

Everyone loves movies, right? Well, if not movies, how about plays, or dramas, or theatre? Ever wondered what would happen if the different team members didn't speak to each other? By team I don't just mean the actors, but the sets team, make-up personnel, audio-visual guys, sound system guys, and so on. It is needless to say that every member has an important contribution towards the end product, and an immense amount of coordination is required between these teams.

A blockbuster movie is a product of hundreds of people working together toward a common goal. Similarly, great software is an application where several objects work together to meet some business target. As a team, every object must be aware of the other, and communicate with each other to get their jobs done.

In a banking system, the money transfer service must be aware of the account service, and the account service must be aware of the accounts repository, and so on. All these components work together to make the banking system workable. In Chapter 1, *Getting Started with Framework 5.0 and Design Patterns*, you saw the same banking example created with the traditional approach, that is, creating objects using construction and direct object initiation. This traditional approach leads to complicated code, is difficult to reuse and unit test, and is also highly coupled to one another.

But in Spring, objects have a responsibility to do their jobs without the need to find and create the other dependent objects that are required in their jobs. The Spring container takes the responsibility to find or create the other dependent objects, and to collaborate with their dependencies. In the previous example of the banking system, the transfer service depends on the account service, but it doesn't have to create the account service, so the dependency is created by the container, and is handed over to the dependent objects in the application.

In this chapter, we will discuss the behind-the-scenes story of the Spring-based application with reference to the **dependency injection (DI)** pattern, and how it works. By the end of this chapter, you will understand how the objects of your Spring-based application create associations between them, and how Spring wires these objects for a job done. You will also learn many ways to wire beans in Spring.

This chapter will cover the following topics:

- The dependency injection pattern
- Types of dependency injection patterns
- Resolving dependency using the Abstract Factory pattern
- Lookup-method injection pattern
- Configuring beans using the Factory pattern
- Configuring dependencies
- Common best practices for configuring dependencies in an application

The dependency injection pattern

In any enterprise application, coordination between the working objects is very important for a business goal. The relationship between objects in an application represents the dependency of an object, so each object would get the job done with coordination of the dependent objects in the application. Such required dependencies between the objects tend to be complicated and with tight-coupled programming in the application. Spring provides a solution to the tight-coupling code of an application by using the dependency injection pattern. Dependency injection is a design pattern, which promotes the loosely coupled classes in the application. This means that the classes in the system depend on the behavior of others, and do not depend on instantiation of object of the classes. The dependency injection pattern also promotes programming to interface instead of programming to implementation. Object dependencies should be on an interface, and not on concrete classes, because a loosely coupled structure offers you greater reusability, maintainability, and testability.

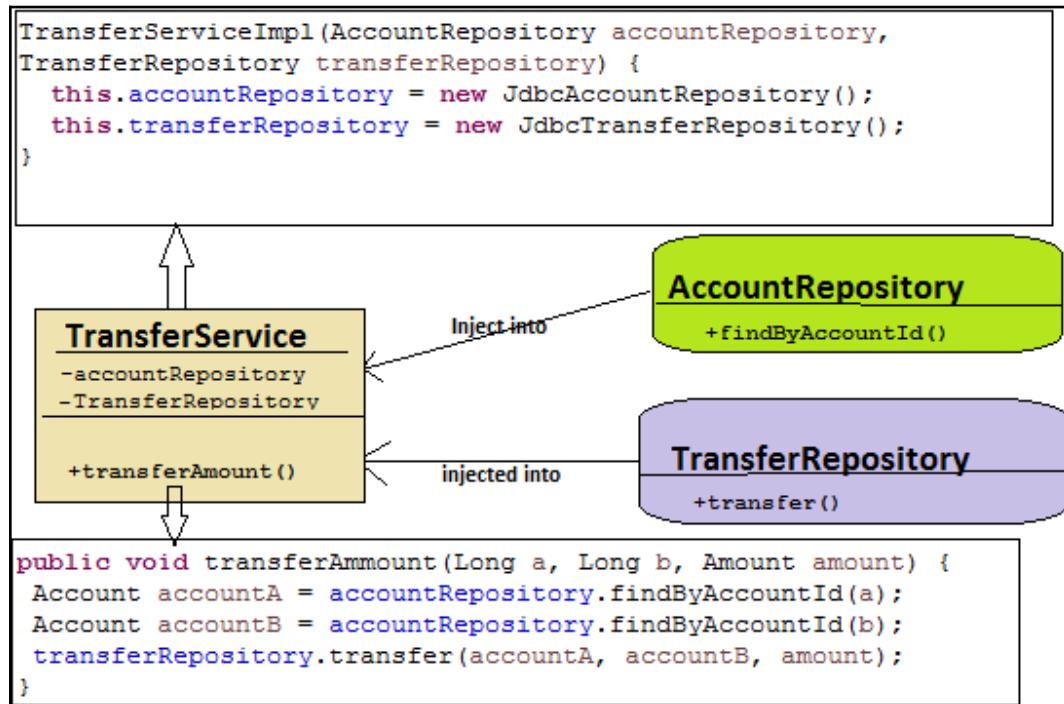
Solving problems using the dependencies injection pattern

In any enterprise application, a common problem to handle is how to configure and wire together the different elements to achieve a business goal--for example, how to bind together the controllers at the web layer with the services and repository interfaces written by different members of the team without knowing about the controllers of the web layers. So, there are a number frameworks that provide a solution for this problem by using lightweight containers to assemble the components from different layers. Examples of such types of frameworks are PicoContainer and Spring Framework.

The containers of PicoContainer and Spring use a number of design patterns to solve the problem of assembling the different components of different layers. Here, I am going to discuss one of these design patterns--the dependency injection pattern. Dependency injection provides us with a decoupled and loosely coupled system. It ensures construction of the dependent object. In the following example, we'll demonstrate how the dependency injection pattern solves the common problems related to collaboration between the various layered components.

Without dependency injection

In the following Java example, first of all, let's see what is a dependency between two classes? Take a look at the following class diagram:



TransferService has dependencies with AccountRepository and TransferRepository for transferAmount() method with Direct Instantiation of repositories classes.

As seen in the preceding diagram, the **TransferService** class contains two member variables, **AccountRepository** and **TransferRepository**. These are initialized by the **TransferService** constructor. **TransferService** controls which implementation of the repositories is used. It also controls their construction. In this situation, **TransferService** is said to have a hard-coded dependency on the following example:

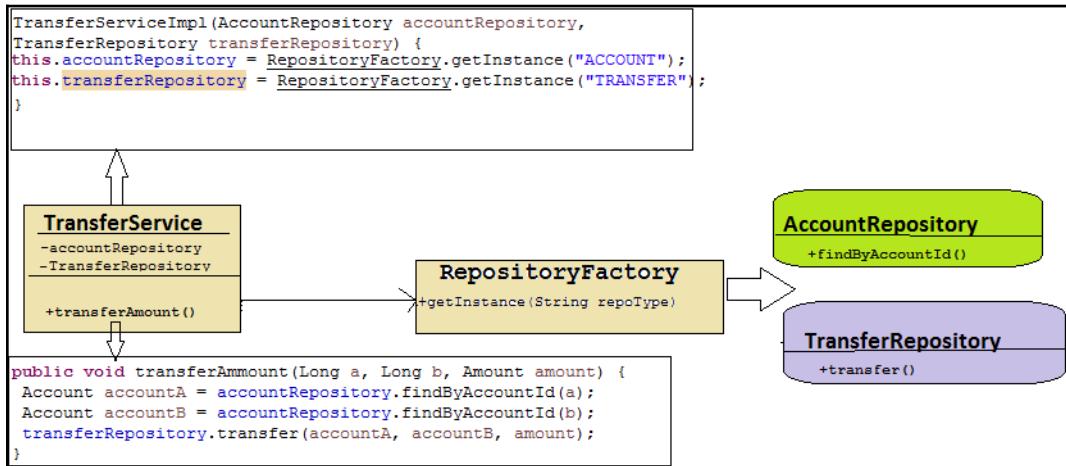
Following is the TransferServiceImpl.java file:

```
public class TransferServiceImpl implements TransferService {  
    AccountRepository accountRepository;  
    TransferRepository transferRepository;  
    public TransferServiceImpl(AccountRepository accountRepository,  
    TransferRepository transferRepository) {  
        super();  
        // Specify a specific implementation in the constructor  
        instead of using dependency injection  
        this.accountRepository = new JdbcAccountRepository();  
        this.transferRepository = new JdbcTransferRepository();  
    }  
    // Method within this service that uses the accountRepository  
    and  
    transferRepository  
    @Override  
    public void transferAmmount(Long a, Long b, Amount amount) {  
        Account accountA = accountRepository.findById(accountId(a));  
        Account accountB = accountRepository.findById(accountId(b));  
        transferRepository.transfer(accountA, accountB, amount);  
    }  
}
```

In the preceding example, the TransferServiceImpl class has dependencies of two classes, that is AccountRepository and TransferRepository. The TransferServiceImpl class has two member variables of the dependent classes, and initializes them through its constructor by using the JDBC implementation of repositories such as JdbcAccountRepository and JdbcTransferRepository. The TransferServiceImpl class is tightly coupled with the JDBC implementation of repositories; in case the JDBC implementation is changed to a JPA implementation, you have to change your TransferServiceImpl class as well.

According to the SOLID (Single Responsibility Principle, Open Closed Principle, Liskov's Substitution Principle, Interface Segregation Principle, Dependency Inversion Principle) principles, a class should have a single responsibility in the application, but in the preceding example, the TransferServiceImpl class is also responsible for constructing the objects of JdbcAccountRepository and JdbcTransferRepository classes. We can't use direct instantiation of objects in the class.

In our first attempt to avoid the direct instantiation logic in the `TransferServiceImpl` class, we can use a Factory class that creates instances of `TransferServiceImpl`. According to this idea, `TransferServiceImpl` minimizes the dependency from `AccountRepository` and `TransferRepository`--earlier we had a tightly coupled implementation of the repositories, but now it refers only to the interface, as shown in the following diagram:



`TransferService` has dependencies with `AccountRepository` and `TransferRepository` for `transferAmount()` method with Factory of repositories classes.

But the `TransferServiceImpl` class is, again, tightly coupled with the implementation of the `RepositoryFactory` class. Moreover, this process is not suitable for cases where we have more number of dependencies, which increases either the Factory classes or the complexity of the Factory class. The repository classes can also have other dependencies.

The following code uses the Factory class to get the `AccountRepository` and `TransferRepository` classes:

Following is the `TransferServiceImpl.java` file:

```

package com.packt.patterninspring.chapter4.bankapp.service;
public class TransferServiceImpl implements TransferService {
    AccountRepository accountRepository;
    TransferRepository transferRepository;
    public TransferServiceImpl(AccountRepository accountRepository,
    TransferRepository transferRepository) {
        this.accountRepository = RepositoryFactory.getInstance();
        this.transferRepository = RepositoryFactory.getInstance();
    }
}
  
```

```

    }
    @Override
    public void transferAmount(Long a, Long b, Amount amount) {
        Account accountA = accountRepository.findById(a);
        Account accountB = accountRepository.findById(b);
        transferRepository.transfer(accountA, accountB, amount);
    }
}

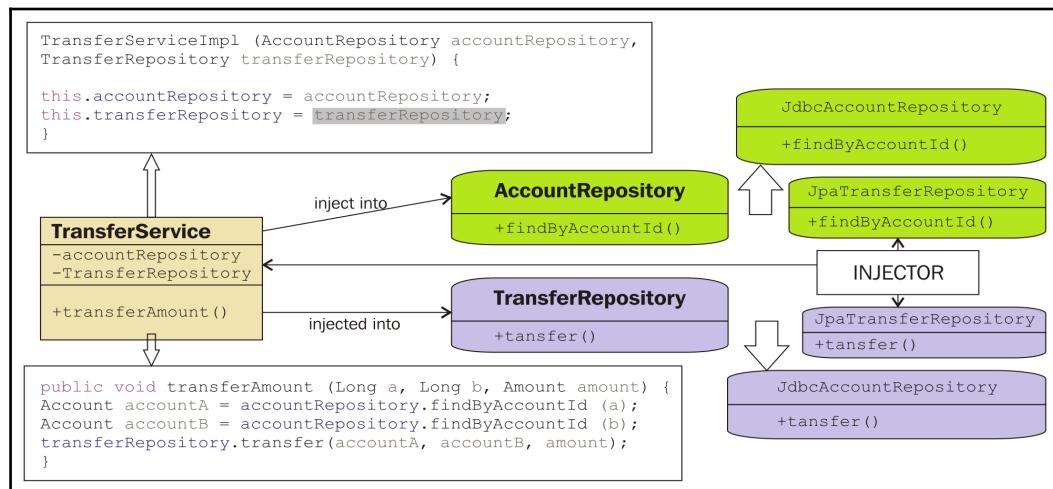
```

In the preceding code example, we have minimized tight coupling, and removed direction object instantiation from the `TransferServiceImpl` class, but this is not the optimal solution.

With dependency injection pattern

The Factory idea avoids direct instantiation of an object of a class, and we also have to create another module that is responsible for wiring the dependencies between classes. This module is known as a **dependency injector**, and is based on the **Inversion of Control (IoC)** pattern. According to the IoC Framework, the Container it is responsible for object instantiation, and to resolve the dependencies among classes in the application. This module has its own life cycle of construction and destruction for the object defined under its scope.

In the following diagram, we have used the dependency injection pattern to resolve the dependencies of the `TransferServiceImpl` class:



Using dependency injection design pattern to resolve dependencies for TransferService.

In the following example, we have used an interface to resolve the dependencies:

Following is the `TransferServiceImpl.java` file:

```
package com.packt.patterninspring.chapter4.bankapp.service;
public class TransferServiceImpl implements TransferService {
    AccountRepository accountRepository;
    TransferRepository transferRepository;
    public TransferServiceImpl(AccountRepository accountRepository,
        TransferRepository transferRepository) {
        this.accountRepository = accountRepository;
        this.transferRepository = transferRepository;
    }
    @Override
    public void transferAmmount(Long a, Long b, Amount amount) {
        Account accountA = accountRepository.findById(a);
        Account accountB = accountRepository.findById(b);
        transferRepository.transfer(accountA, accountB, amount);
    }
}
```

In the `TransferServiceImpl` class, we passed references of the `AccountRepository` and `TransferRepository` interfaces to the constructor. Now the `TransferServiceImpl` class is loosely coupled with the implementation repository class (use any flavor, either JDBC or JPA implementation of repository interfaces), and the framework is responsible for wiring the dependencies with the involved dependent class. Loose coupling offers us greater reusability, maintainability, and testability.

The Spring Framework implements the dependency injection pattern to resolve dependencies among the classes in a Spring application. Spring DI is based on the IoC concept, that is, the Spring Framework has a container where it creates, manages, and destructs the objects; it is known as a Spring IoC container. The objects lying within the Spring container are known as **Spring beans**. There are many ways to wire beans in a Spring application. Let's take a look at the three most common approaches for configuring the Spring container.

In the following section, we'll look at the types of the dependency injection pattern; you can configure the dependencies by using either one of them.

Types of dependency injection patterns

The following are the types of dependency injections that could be injected into your application:

- Constructor-based dependency injection
- Setter-based dependency injection

Constructor-based dependency injection pattern

Dependency injection is a design pattern to resolve the dependencies of dependent classes, and dependencies are nothing but object attributes. The injector has to be constructed for the dependent objects by using one of the ways constructor injection or setter injection. A constructor injection is one of the ways of fulfilling these object attributes at the time of creation to instantiate the object. An object has a public constructor that takes dependent classes as constructor arguments to inject the dependencies. You can declare more than one constructor into the dependent class. Earlier, only the PicoContainer Framework is used a constructor-based dependency injection to resolve dependencies. Currently, the Spring Framework also supports constructor injections to resolve dependencies.

Advantages of the constructor injection pattern

The following are the advantages if you use a constructor injection in your Spring application:

- Constructor-based dependency injection is more suitable for mandatory dependencies, and it makes a strong dependency contract
- Constructor-based dependency injection provides a more compact code structure than others
- It supports testing by using the dependencies passed as constructor arguments to the dependent class
- It favors the use of immutable objects, and does not break the information hiding principle

Disadvantages of constructor injection pattern

The following is the only drawback of this constructor-based injection pattern:

- It may cause circular dependency. (Circular dependency means that the dependent and the dependency class are also dependents on each other, for example, class A depends on Class B and Class B depends on Class A)

Example of constructor-based dependency injection pattern

Let's see the following example for constructor-based dependency injection. In the following code, we have a `TransferServiceImpl` class, and its constructor takes two arguments:

```
public class TransferServiceImpl implements TransferService {  
    AccountRepository accountRepository;  
    TransferRepository transferRepository;  
    public TransferServiceImpl(AccountRepository accountRepository,  
        TransferRepository transferRepository) {  
        this.accountRepository = accountRepository;  
        this.transferRepository = transferRepository;  
    }  
    // ...  
}
```

The repositories will also be managed by the Spring container, and, as such, will have the `datasource` object for database configuration injected into them by the container, as follows:

Following is the `JdbcAccountRepository.java` file:

```
public class JdbcAccountRepository implements AccountRepository{  
    JdbcTemplate jdbcTemplate;  
    public JdbcAccountRepository(DataSource dataSource) {  
        this.jdbcTemplate = new JdbcTemplate(dataSource);  
    }  
    // ...  
}
```

Following is the `JdbcTransferRepository.java` file:

```
public class JdbcTransferRepository implements TransferRepository{  
    JdbcTemplate jdbcTemplate;  
    public JdbcTransferRepository(DataSource dataSource) {  
        this.jdbcTemplate = new JdbcTemplate(dataSource);  
    }  
    // ...  
}
```

You can see in the preceding code the JDBC implementation of the repositories as `AccountRepository` and `TransferRepository`. These classes also have one argument constructor to inject the dependency with the `DataSource` class.

Let's see another way of implementing a dependency injection in the enterprise application, which is setter injection.

Setter-based dependency injection

The injector of the container has another way to wire the dependency of the dependent object. In setter injection, one of the ways to fulfil these dependencies is by providing a setter method in the dependent class. Object has a public setter methods that takes dependent classes as method arguments to inject dependencies. For setter-based dependency injection, the constructor of the dependent class is not required. There are no changes required if you change the dependencies of the dependent class. Spring Framework and PicoContainer Framework support setter injection to resolve the dependencies.

Advantages of setter injection

The following are the advantages if you use the setter injection pattern in your Spring application:

- Setter injection is more readable than the constructor injection
- Setter injection solves the circular dependency problem in the application
- Setter injection allows costly resources or services to be created as late as possible, and only when required
- Setter injection does not require the constructor to be changed, but dependencies are passed through public properties that are exposed

Disadvantage of the setter injection

The following are the drawbacks of the setter injection pattern:

- Security is lesser in the setter injection pattern, because it can be overridden
- A setter-based dependency injection does not provide a code structure as compact as the constructor injection
- Be careful whenever you use setter injection, because it is not a required dependency

Example of a setter-based dependency injection

Let's see the following example for setter-based dependency injection. The following TransferServiceImpl class, has setter methods with one argument of the repository type:

Following is the TransferServiceImpl.java file:

```
public class TransferServiceImpl implements TransferService {  
    AccountRepository accountRepository;  
    TransferRepository transferRepository;  
    public void setAccountRepository(AccountRepository  
        accountRepository) {  
        this.accountRepository = accountRepository;  
    }  
    public void setTransferRepository(TransferRepository  
        transferRepository) {  
        this.transferRepository = transferRepository;  
    }  
    // ...  
}
```

Similarly, let's define a setter for the repositories' implementations as follows:

Following is the JdbcAccountRepository.java file:

```
public class JdbcAccountRepository implements AccountRepository{  
    JdbcTemplate jdbcTemplate;  
    public setDataSource(DataSource dataSource) {  
        this.jdbcTemplate = new JdbcTemplate(dataSource);  
    }  
    // ...  
}
```

Following is the `JdbcTransferRepository.java` file:

```
public class JdbcTransferRepository implements TransferRepository{  
    JdbcTemplate jdbcTemplate;  
    public setDataSource(DataSource dataSource) {  
        this.jdbcTemplate = new JdbcTemplate(dataSource);  
    }  
    // ...  
}
```

You can see in the preceding code the JDBC implementation of the repositories as `AccountRepository` and `TransferRepository`. These classes have a setter method with one argument to inject the dependency with the `DataSource` class.

Constructor versus setter injection and best practices

The Spring Framework provides support for both types of dependency injection patterns. Both, the constructor and setter injection pattern, assemble the elements in the system. The choice between the setter and constructor injections depends on your application requirement, and the problem at hand. Let's see the following table, which lists some differences between the constructor and setter injections, and some best practices to select which one is suitable in your application.

Constructors injection

A class with constructor takes arguments; it is very compact sometimes, and clear to understand what it creates.

This is a better choice when the dependency is mandatory.

It allows you to hide the object attributes that are immutable, because it does not have setters for these object attributes. To ensure the immutable nature of the object, use the constructor injection pattern instead of the setter injection.

It creates circular dependency in your application.

Setter injection

Here, the object is constructed, but it is not clear whether its attributes are initialized or not.

This is suitable when the dependency is not mandatory.

It doesn't ensure the immutable nature of the object, the object.

It solves the problem of circular dependency in your application. In this case, the setter injection is a better choice than constructor.

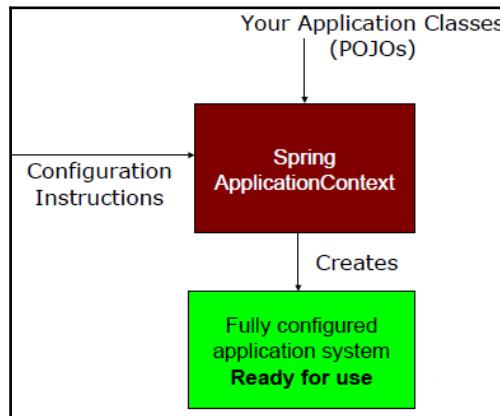
It is not suitable for scalar value dependencies in the application.

If you have simple parameters such as strings and integers as dependencies, the setter injection is better to use, because each setter name indicates what the value is supposed to do.

In the next section, you'll learn how to configure the injector to find the beans and wire them together, and how the injector manages the beans. Here, I will use the Spring configuration for the dependency injection pattern.

Configuring the dependency injection pattern with Spring

In this section, I will explain the process required to configure dependencies in an application. The mainstream injectors are Google Guice, Spring, and Weld. In this chapter, I am using the Spring Framework, so, we will see the Spring configuration here. The following diagram is a high-level view of how Spring works:



How Spring works using dependency injection pattern

In the preceding diagram, the **Configuration Instruction** is the meta configuration of your application. Here, we define the dependencies in **Your Application Classes (POJOs)**, and initialize the Spring container to resolve the dependency by combining the POJOs and **Configuration Instructions**, and finally, you have a fully configured and executable system or application.

As you have seen in the preceding diagram, the Spring container creates the beans in your application, and assembles them for relationships between those objects via the DI pattern. The Spring container creates the beans based on the configuration that we give to the framework, so, it's your responsibility to tell Spring which beans to create, and how to wire them together.

Spring is very flexible in configuring the dependency of Spring beans. The following are three ways to configure the metadata of your application:

1. **Dependency injection pattern with Java-based configuration**—it is an explicit configuration in Java.
2. **Dependency injection pattern with Annotation-based configuration**—it is an implicit bean discovery, and automatic wiring.
3. **Dependency injection pattern with XML-based configuration**—it is an explicit configuration in XML.

Spring provides three choices to wire beans in Spring. You must select one of the choices, but no single choice is the best match for any application. It depends on your application, and you can also mix and match these choices into a single application. Let's now discuss the dependency injection pattern with Java-based configuration in detail.

Dependency injection pattern with Java-based configuration

As of Spring 3.0, it provides a Java-based Spring configuration to wire the Spring beans. Take a look at the following Java configuration class (`AppConfig.java`) to define the Spring bean and their dependencies. The Java-based configuration for dependency injection is a better choice, because it is more powerful and type-safe.

Creating a Java configuration class - AppConfig.java

Let's create an `AppConfig.java` configuration class for our example:

```
package com.packt.patterninspring.chapter4.bankapp.config;
import org.springframework.context.annotation.Configuration;
@Configuration
public class AppConfig {
    /**
}
```

The preceding `AppConfig` class is annotated with the `@Configuration` annotation, which indicates that it is a configuration class of the application that contains the details on bean definitions. This file will be loaded by the Spring application context to create beans for your application.

Let's now see how you can declare the `TransferService`, `AccountRepository` and `TransferRepository` beans in `AppConfig`.

Declaring Spring beans into configuration class

To declare a bean in a Java-based configuration, you have to write a method for the desired type object creation in the configuration class, and annotate that method with `@Bean`. Let's see the following changes made in the `AppConfig` class to declare the beans:

```
package com.packt.patterninspring.chapter4.bankapp.config;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
@Configuration
public class AppConfig {
    @Bean
    public TransferService transferService() {
        return new TransferServiceImpl();
    }
    @Bean
    public AccountRepository accountRepository() {
        return new JdbcAccountRepository();
    }
    @Bean
    public TransferRepository transferRepository() {
        return new JdbcTransferRepository();
    }
}
```

```
}
```

In the preceding configuration file, I declared three methods to create instances for `TransferService`, `AccountRepository`, and `TransferRepository`. These methods are annotated with the `@Bean` annotation to indicate that they are responsible for instantiating, configuring, and initializing a new object to be managed by the Spring IoC container. Each bean in the container has a unique bean ID; by default, a bean has an ID same as the `@Bean` annotated method name. In the preceding case, the beans will be named as `transferService`, `accountRepository`, and `transferRepository`. You can also override that default behavior by using the `name` attribute of the `@Bean` annotation as follows:

```
@Bean(name="service")
public TransferService transferService(){
    return new TransferServiceImpl();
}
```

Now "service" is the bean name of that bean `TransferService`.

Let's see how you can inject dependencies for the `TransferService`, `AccountRepository`, and `TransferRepository` beans in `AppConfig`.

Injecting Spring beans

In the preceding code, I declared the beans `TransferService`, `AccountRepository`, and `TransferRepository`; these beans had no dependencies. But, actually, the `TransferService` bean depends on `AccountRepository` and `TransferRepository`. Let's see the following changes made in the `AppConfig` class to declare the beans:

```
package com.packt.patterninspring.chapter4.bankapp.config;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
@Configuration
public class AppConfig {
    @Bean
    public TransferService transferService() {
        return new TransferServiceImpl(accountRepository(),
            transferRepository());
    }
    @Bean
    public AccountRepository accountRepository() {
        return new JdbcAccountRepository();
    }
}
```

```
@Bean  
public TransferRepository transferRepository() {  
    return new JdbcTransferRepository();  
}  
}
```

In the preceding example, the simplest way to wire up beans in a Java-based configuration is to refer to the referenced bean's method. The `transferService()` method constructs the instance of the `TransferServiceImpl` class by calling the arguments constructor that takes `AccountRepository` and `TransferRepository`. Here, it seems that the constructor of the `TransferServiceImpl` class is calling the `accountRepository()` and `transferRepository()` methods to create instances of `AccountRepository` and `TransferRepository` respectively, but it is not an actual call to create instances. The Spring container creates instances of `AccountRepository` and `TransferRepository`, because the `accountRepository()` and `transferRepository()` methods are annotated with the `@Bean` annotation. Any call to the bean method by another bean method will be intercepted by Spring to ensure the default singleton scope (this will be discussed further in Chapter 5, *Understanding the Bean Life cycle and Used Patterns*) of the Spring beans by that method is returned rather than allowing it to be invoked again.

Best approach to configure the dependency injection pattern with Java

In the previous configuration example, I declared the `transferService()` bean method to construct an instance of the `TransferServiceImpl` class by using its arguments constructor. The bean methods, `accountRepository()` and `transferRepository()`, are passed as arguments of the constructor. But in an enterprise application, a lot of configuration files depend on the layers of the application architecture. Suppose the service layer and the infrastructure layer have their own configuration files. That means that the `accountRepository()` and `transferRepository()` methods may be in different configuration files, and the `transferService()` bean method may be in another configuration file. Passing bean methods into the constructor is not a good practice for configuration of the dependency injection pattern with Java. Let's see a different and the best approach to configuring the dependency injection:

```
package com.packt.patterninspring.chapter4.bankapp.config;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
@Configuration
```

```
public class AppConfig {  
    @Bean  
    public TransferService transferService(AccountRepository  
    accountRepository, TransferRepository transferRepository) {  
        return new TransferServiceImpl(accountRepository,  
        transferRepository);  
    }  
    @Bean  
    public AccountRepository accountRepository() {  
        return new JdbcAccountRepository();  
    }  
    @Bean  
    public TransferRepository transferRepository() {  
        return new JdbcTransferRepository();  
    }  
}
```

In the preceding code, the `transferService()` method asks for `AccountRepository` and `TransferRepository` as parameters. When Spring calls `transferService()` to create the `TransferService` bean, it autowires `AccountRepository` and `TransferRepository` into the configuration method. With this approach, the `transferService()` method can still inject `AccountRepository` and `TransferRepository` into the constructor of `TransferServiceImpl` without explicitly referring to the `accountRepository()` and `transferRepository()` `@Bean` methods.

Let's now take a look at dependency injection pattern with XML-based configuration.

Dependency injection pattern with XML-based configuration

Spring provides dependency injection with XML-based configuration from the very beginning. It is the primary way of configuring a Spring application. According to me, every developer should have an understanding of how to use XML with a Spring application. In this section, I am going to explain the same example as discussed in the previous section of Java-based configuration with reference to XML-based configuration.

Creating an XML configuration file

In the section on Java-based configuration, we had created an `AppConfig` class annotated with the `@Configuration` annotation. Similarly, for XML-based configuration, we will now create an `applicationContext.xml` file rooted with a `<beans>` element. The following simplest possible example shows the basic structure of XML-based configuration metadata:

Following is the `applicationContext.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd">
    <!-- Configuration for bean definitions go here -->
</beans>
```

The preceding XML file is a configuration file of the application which contains the details on bean definitions. This file is also loaded by the XML-flavored implementation of `ApplicationContext` to create beans for your application. Let's see how you can declare the `TransferService`, `AccountRepository` and `TransferRepository` beans in the preceding XML file.

Declaring Spring beans in an XML file

As with Java, we have to declare a class as a Spring bean into Spring's XML-based configuration by using an element of the Spring-beans schema as a `<bean>` element. The `<bean>` element is the XML analogue to `JavaConfig`'s `@Bean` annotation. We add the following configuration to the XML-based configuration file:

```
<bean id="transferService"
      class="com.packt.patterninspring.chapter4.
            bankapp.service.TransferServiceImpl"/>
<bean id="accountRepository"
      class="com.packt.patterninspring.chapter4.
            bankapp.repository.jdbc.JdbcAccountRepository"/>
<bean id="transferService"
      class="com.packt.patterninspring.chapter4.
            bankapp.repository.jdbc.JdbcTransferRepository"/>
```

In the preceding code, I have created a very simple bean definition. In this configuration, the `<bean>` element has an `id` attribute to identify the individual bean definition. The `class` attribute is expressed as the fully qualified class name to create this bean. The value of the `id` attribute refers to collaborating objects. So let's see how to configure the collaborating beans to resolve the dependencies in the application.

Injecting Spring beans

Spring provides these two ways to define the DI pattern to inject the dependency with the dependent bean in an application:

- Using constructor injection
- Using setter injection

Using constructor injection

For the DI pattern with the construction injection, Spring provides you two basic options as the `<constructor-arg>` element and c-namespace introduced in Spring 3.0. c-namespace has less verbosity in the application, which is the only difference between them--you can choose any one. Let's inject the collaborating beans with the construction injection as follows:

```
<bean id="transferService"
      class="com.packt.patterninspring.chapter4.
      bankapp.service.TransferServiceImpl">
    <constructor-arg ref="accountRepository"/>
    <constructor-arg ref="transferRepository"/>
</bean>
<bean id="accountRepository"
      class="com.packt.patterninspring.chapter4.
      bankapp.repository.jdbc.JdbcAccountRepository"/>
<bean id="transferRepository"
      class="com.packt.patterninspring.chapter4.
      bankapp.repository.jdbc.JdbcTransferRepository"/>
```

In the preceding configuration, the `<bean>` element of `TransferService` has two `<constructor-arg>`. This tells it to pass a reference to the beans whose IDs are `accountRepository` and `transferRepository` to the constructor of `TransferServiceImpl`.

As of Spring 3.0, the c-namespace, similarly, has a more succinct way of expressing constructor args in XML. For using this namespace, we have to add its schema in the XML file, as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:c="http://www.springframework.org/schema/c"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd">
    <bean id="transferService"
          class="com.packt.patterninspring.chapter4.
                bankapp.service.TransferServiceImpl"
          c:accountRepository-ref="accountRepository" c:transferRepository-
          ref="transferRepository"/>
    <bean id="accountRepository"
          class="com.packt.patterninspring.chapter4.
                bankapp.repository.jdbc.JdbcAccountRepository"/>
    <bean id="transferRepository"
          class="com.packt.patterninspring.chapter4.
                bankapp.repository.jdbc.JdbcTransferRepository"/>
    <!-- more bean definitions go here -->
</beans>
```

Let's see how to set up these dependencies with the setter injection.

Using setter injection

Using the injection, Spring also provides you with two basic options as the `<property>` element and p-namespace introduced in Spring 3.0. The p-namespace also reduced verbosity of code in the application, which is the only difference between them, you can choose any one. Let's inject the collaborating beans with the setter injection as follows:

```
<bean id="transferService"
      class="com.packt.patterninspring.chapter4.
            bankapp.service.TransferServiceImpl">
    <property name="accountRepository" ref="accountRepository"/>
    <property name="transferRepository" ref="transferRepository"/>
</bean>
<bean id="accountRepository"
      class="com.packt.patterninspring.chapter4.
            bankapp.repository.jdbc.JdbcAccountRepository"/>
<bean id="transferRepository"
      class="com.packt.patterninspring.chapter4.
            bankapp.repository.jdbc.JdbcTransferRepository"/>
```

In the preceding configuration, the `<bean>` element of `TransferService` has two `<property>` elements which tell it to pass a reference to the beans whose IDs are `accountRepository` and `transferRepository` to the setter methods of `TransferServiceImpl`, as follows:

```
package com.packt.patterninspring.chapter4.bankapp.service;

import com.packt.patterninspring.chapter4.bankapp.model.Account;
import com.packt.patterninspring.chapter4.bankapp.model.Amount;
import com.packt.patterninspring.chapter4.bankapp.
    repository.AccountRepository;
import com.packt.patterninspring.chapter4.bankapp.
    repository.TransferRepository;

public class TransferServiceImpl implements TransferService {
    AccountRepository accountRepository;
    TransferRepository transferRepository;
    public void setAccountRepository(AccountRepository
        accountRepository) {
        this.accountRepository = accountRepository;
    }
    public void setTransferRepository(TransferRepository
        transferRepository) {
        this.transferRepository = transferRepository;
    }
    @Override
    public void transferAmmount(Long a, Long b, Amount amount) {
        Account accountA = accountRepository.findById(a);
        Account accountB = accountRepository.findById(b);
        transferRepository.transfer(accountA, accountB, amount);
    }
}
```

In the preceding file, if you use this Spring bean without setter methods, the properties `accountRepository` and `transferRepository` will be initialized as null without injecting the dependency.

As of Spring 3.0, the p-namespace, similarly, has a more succinct way of expressing property in XML. For using this namespace, we have to add its schema in the XML file as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd">
    <bean id="transferService"
          class="com.packt.patterninspring.chapter4.bankapp.
          service.TransferServiceImpl"
          p:accountRepository-ref="accountRepository" p:transferRepository-
          ref="transferRepository"/>
    <bean id="accountRepository"
          class="com.packt.patterninspring.chapter4.
          bankapp.repository.jdbc.JdbcAccountRepository"/>
    <bean id="transferRepository"
          class="com.packt.patterninspring.chapter4.
          bankapp.repository.jdbc.JdbcTransferRepository"/>
    <!-- more bean definitions go here -->
</beans>
```

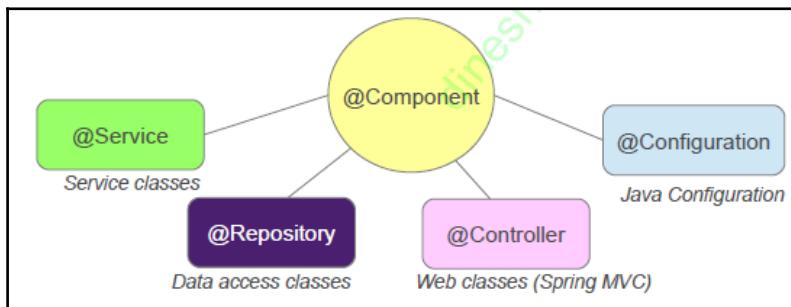
Let's now take a look at the dependency injection pattern with Annotation-based configuration.

Dependency injection pattern with Annotation-based configuration

As discussed in the previous two sections, we defined the DI pattern with Java-and XML-based configurations, and these two options define dependencies explicitly. It creates the Spring beans by using either the @Bean annotated method in the AppConfig Java file, or the <bean> element tag in the XML configuration file. By these methods, you can also create the bean for those classes which lie outside the application, that is, classes that exist in third-party libraries. Now let's discuss another way to create Spring beans, and define the dependencies between them by using implicit configuration through the Stereotype annotations.

What are Stereotype annotations?

The Spring Framework provides you with some special annotations. These annotations are used to create Spring beans automatically in the application context. The main stereotype annotation is `@Component`. By using this annotation, Spring provides more Stereotype meta annotations such as `@Service`, used to create Spring beans at the Service layer, `@Repository`, which is used to create Spring beans for the repositories at the DAO layer, and `@Controller`, which is used to create Spring beans at the controller layer. This is depicted in the following diagram:



By using these annotations, Spring creates automatic wiring in these two ways:

- **Component scanning:** In this, Spring automatically searches the beans to be created in the Spring IoC container
- **Autowiring:** In this, Spring automatically searches the bean dependencies in the Spring IoC container

Implicitly, the DI pattern configuration reduces the verbosity of an application, and minimizes explicit configuration. Let's demonstrate component scanning and autowiring in the same example as discussed previously. Here, Spring will create the beans for `TransferService`, `TransferRepository`, and `AccountRepository` by discovering them, and automatically inject them to each other as per the defined dependencies.

Creating auto searchable beans using Stereotype annotations

Let's see the following `TransferService` interface. Its implementation is annotated with `@Component`. Please refer to the following code:

```
package com.packt.patterninspring.chapter4.bankapp.service;
public interface TransferService {
    void transferAmmount(Long a, Long b, Amount amount);
}
```

The preceding interface is not important for this approach of configuration--I have taken it just for loose coupling in the application. Let's see its implementation, which is as follows:

```
package com.packt.patterninspring.chapter1.bankapp.service;
import org.springframework.stereotype.Component;
@Component
public class TransferServiceImpl implements TransferService {
    @Override
    public void transferAmmount(Long a, Long b, Amount amount) {
        //business code here
    }
}
```

You can see in the preceding code that `TransferServiceImpl` is annotated with the `@Component` annotation. This annotation is used to identify this class as a component class, which means, it is eligible to scan and create a bean of this class. Now there is no need to configure this class explicitly as a bean either by using XML or Java configuration--Spring is now responsible for creating the bean of the `TransferServiceImpl` class, because it is annotated with `@Component`.

As mentioned earlier, Spring provides us meta annotations for the `@Component` annotation as `@Service`, `@Repository`, and `@Controller`. These annotations are based on a specific responsibility at different layers of the application. Here, `TransferService` is the service layer class; *as a best practice of Spring configuration*, we have to annotate this class with the specific annotation, `@Service`, rather than with the generic annotation, `@Component`, to create the bean of this class. The following is the code for the same class annotated with the `@Service` annotation:

```
package com.packt.patterninspring.chapter1.bankapp.service;
import org.springframework.stereotype.Service;
@Service
public class TransferServiceImpl implements TransferService {
    @Override
```

```
public void transferAmmount(Long a, Long b, Amount amount) {  
    //business code here  
}  
}
```

Let's see other classes in the application--these are the implementation classes of AccountRepository--and the TransferRepository interfaces are the repositories working at the DAO layer of the application. *As a best practice*, these classes should be annotated with the @Repository annotation rather than using the @Component annotation as shown next.

JdbcAccountRepository.java implements the AccountRepository interface:

```
package  
com.packt.patterninspring.chapter4.bankapp.repository.jdbc;  
import org.springframework.stereotype.Repository;  
import com.packt.patterninspring.chapter4.bankapp.model.Account;  
import com.packt.patterninspring.chapter4.bankapp.model.Amount;  
import com.packt.patterninspring.chapter4.bankapp.repository.  
    AccountRepository;  
@Repository  
public class JdbcAccountRepository implements AccountRepository {  
    @Override  
    public Account findByAccountId(Long accountId) {  
        return new Account(accountId, "Arnav Rajput", new  
            Amount(3000.0));  
    }  
}
```

And JdbcTransferRepository.java implements the TransferRepository interface:

```
package  
com.packt.patterninspring.chapter4.bankapp.repository.jdbc;  
import org.springframework.stereotype.Repository;  
import com.packt.patterninspring.chapter4.bankapp.model.Account;  
import com.packt.patterninspring.chapter4.bankapp.model.Amount;  
import com.packt.patterninspring.chapter4.bankapp.  
    repository.TransferRepository;  
@Repository  
public class JdbcTransferRepository implements TransferRepository  
{  
    @Override  
    public void transfer(Account accountA, Account accountB, Amount  
        amount) {  
        System.out.println("Transferring amount from account A to B via  
            JDBC implementation");  
    }  
}
```

```
    }  
}
```

In Spring, you have to enable component scanning in your application, because it is not enabled by default. You have to create a configuration Java file, and annotate it with `@Configuration` and `@ComponentScan`. This class is used to search out classes annotated with `@Component`, and to create beans from them.

Let's see how Spring scans the classes which are annotated with any stereotype annotations.

Searching beans using component scanning

The following minimum configuration is required to search beans using component scanning in a Spring application:

```
package com.packt.patterninspring.chapter4.bankapp.config;  
  
import org.springframework.context.annotation.ComponentScan;  
import org.springframework.context.annotation.Configuration;  
  
@Configuration  
@ComponentScan  
public class AppConfig {  
}
```

The `AppConfig` class defines a Spring wiring configuration class same as the Java-based Spring configuration in the previous section. There is one thing to be observed here--the `AppConfig` file has one more `@ComponentScan`, as earlier it had only the `@Configuration` annotation. The configuration file `AppConfig` is annotated with `@ComponentScan` to enable component scanning in Spring. The `@ComponentScan` annotation scans those classes that are annotated with `@Component` by default in the same package as the configuration class. Since the `AppConfig` class is in the `com.packt.patterninspring.chapter4.bankapp.config` package, Spring will scan only this package and its sub packages. But our component application classes are in the `com.packt.patterninspring.chapter1.bankapp.service` and `com.packt.patterninspring.chapter4.bankapp.repository.jdbc` packages, and these are not subpackages of `com.packt.patterninspring.chapter4.bankapp.config`. In this case, Spring allows to override the default package scanning of the `@ComponentScan` annotation by setting a base package for component scanning. Let's specify a different base package.

You only need to specify the package in the `value` attribute of `@ComponentScan`, as shown here:

```
@Configuration  
@ComponentScan("com.packt.patterninspring.chapter4.bankapp")  
public class AppConfig {  
}
```

Or you can define the base packages with the `basePackages` attribute, as follows:

```
@Configuration  
@ComponentScan(basePackages="com.packt.patterninspring.  
chapter4.bankapp")  
public class AppConfig {  
}
```

In the `@ComponentScan` annotation, the `basePackages` attribute can accept an array of Strings, which means that we can define multiple base packages to scan component classes in the application. In the previous configuration file, Spring will scan all classes of `com.packt.patterninspring.chapter4.bankapp` package, and all the subpackages underneath this package. As a best practice, always define the specific base packages where the components classes exist. For example, in the following code, I define the base packages for the service and repository components:

```
package com.packt.patterninspring.chapter4.bankapp.config;  
import org.springframework.context.annotation.ComponentScan;  
import org.springframework.context.annotation.Configuration;  
@Configuration  
@ComponentScan(basePackages=  
{"com.packt.patterninspring.chapter4.  
bankapp.repository.jdbc", "com.packt.patterninspring.  
chapter4.bankapp.service"})  
public class AppConfig {  
}
```

Now Spring scans only `com.packt.patterninspring.chapter4.bankapp.repository.jdbc` and `com.packt.patterninspring.chapter4.bankapp.service` packages, and its subpackages if they exist. instead of doing a wide range scanning like in the earlier examples.

Rather than specify the packages as simple String values of the `basePackages` attribute of `@ComponentScan`, Spring allows you to specify them via classes or interfaces as follows:

```
package com.packt.patterninspring.chapter4.bankapp.config;
```

```
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import com.packt.patterninspring.chapter4.bankapp.
    repository.AccountRepository;
import com.packt.patterninspring.chapter4.
    bankapp.service.TransferService;
@Configuration
@ComponentScan(basePackageClasses=
{TransferService.class,AccountRepository.class})
public class AppConfig {

}
```

As you can see in the preceding code, the `basePackages` attribute has been replaced with `basePackageClasses`. Now Spring will identify the component classes in those packages where `basePackageClasses` will be used as the base package for component scanning.

It should find the `TransferServiceImpl`, `JdbcAccountRepository`, and `JdbcTransferRepository` classes, and automatically create the beans for these classes in the Spring container. Explicitly, there is no need to define the bean methods for these classes to create Spring beans. Let's turn on component scanning via XML configuration, then you can use the `<context:component-scan>` element from Spring's context namespace. Here is a minimal XML configuration to enable component scanning:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">
    <context:component-scan base-
        package="com.packt.patterninspring.chapter4.bankapp" />
</beans>
```

In the preceding XML file, the `<context:component-scan>` element is same the `@ComponentScan` annotation in the Java-based configuration for component scanning.

Annotating beans for autowiring

Spring provides support for automatic bean wiring. This means that Spring automatically resolves the dependencies that are required by the dependent bean by finding other collaborating beans in the application context. Bean Autowiring is another way of DI pattern configuration. It reduces verbosity in the application, but the configuration is spread throughout the application. Spring's `@Autowired` annotation is used for auto bean wiring. This `@Autowired` annotation indicates that autowiring should be performed for this bean.

In our example, we have `TransferService` which has dependencies of `AccountRepository` and `TransferRepository`. Its constructor is annotated with `@Autowired` indicating that when Spring creates the `TransferService` bean, it should instantiate that bean by using its annotated constructor, and pass in two other beans, `AccountRepository` and `TransferRepository`, which are dependencies of the `TransferService` bean. Let's see the following code:

```
package com.packt.patterninspring.chapter4.bankapp.service;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.packt.patterninspring.chapter4.bankapp.model.Account;
import com.packt.patterninspring.chapter4.bankapp.model.Amount;
import com.packt.patterninspring.chapter4.bankapp.
    repository.AccountRepository;
import com.packt.patterninspring.chapter4.
    bankapp.repository.TransferRepository;
@Service
public class TransferServiceImpl implements TransferService {
    AccountRepository accountRepository;
    TransferRepository transferRepository;
    @Autowired
    public TransferServiceImpl(AccountRepository accountRepository,
        TransferRepository transferRepository) {
        super();
        this.accountRepository = accountRepository;
        this.transferRepository = transferRepository;
    }
    @Override
    public void transferAmmount(Long a, Long b, Amount amount) {
        Account accountA = accountRepository.findById(a);
        Account accountB = accountRepository.findById(b);
        transferRepository.transfer(accountA, accountB, amount);
    }
}
```



Note--As of Spring 4.3, the `@Autowired` annotation is no more required if you define only one construct with arguments in that class. If class has multiple argument constructors, then you have to use the `@Autowired` annotation on any one of them.

The `@Autowired` annotation is not limited to the construction; it can be used with the setter method, and can also be used directly in the field, that is, an `autowired` class property directly. Let's see the following line of code for setter and field injection.

Using `@Autowired` with setter method

Here you can annotate the setter method's `setAccountRepository` and `setTransferRepository` with the `@Autowired` annotation. This annotation can be used with any method. There is no specific reason to use it with the setter method only. Please refer to the following code:

```
public class TransferServiceImpl implements TransferService {  
    //...  
    @Autowired  
    public void setAccountRepository(AccountRepository  
        accountRepository) {  
        this.accountRepository = accountRepository;  
    }  
    @Autowired  
    public void setTransferRepository(TransferRepository  
        transferRepository) {  
        this.transferRepository = transferRepository;  
    }  
    //...  
}
```

Using `@Autowired` with the fields

Here you can annotate those class properties which are required for this class to achieve a business goal. Let's see the following code:

```
public class TransferServiceImpl implements TransferService {  
    @Autowired  
    AccountRepository accountRepository;  
    @Autowired  
    TransferRepository transferRepository;  
    //...  
}
```

In the preceding code, the `@Autowired` annotation resolves the dependency by type and then by name if the property name is the same as the bean name in the Spring container. By default, the `@Autowired` dependency is a required dependency--it raises an exception if the dependency is not resolved, it doesn't matter whether we have used it with a constructor or with the setter method. You can override the required behavior of the `@Autowired` annotation by using the `required` attribute of this annotation. You can set this attribute with the Boolean value `false` as follows:

```
@Autowired(required = false)
public void setAccountRepository(AccountRepository
accountRepository) {
    this.accountRepository = accountRepository;
}
```

In the preceding code, we have set the `required` attribute with the Boolean value `false`. In this case, Spring will attempt to perform autowiring, but if there are no matching beans, it will leave the bean unwired. But as a best practice of code, you should avoid setting its value as `false` until it is absolutely necessary.

The Autowiring DI pattern and disambiguation

The `@Autowiring` annotation reduces verbosity in the code, but it may create some problems when two of the same type of beans exist in the container. Let's see what happens in that situation, with the following example:

```
@Service
public class TransferServiceImpl implements TransferService {
    @Autowired
    public TransferServiceImpl(AccountRepository accountRepository) {
        ...
    }
}
```

The preceding snippet of code shows that the `TransferServiceImpl` class has a dependency with a bean of type `AccountRepository`, but the Spring container contains two beans of the same type, that is, the following:

```
@Repository
public class JdbcAccountRepository implements AccountRepository
{
}
@Repository
public class JpaAccountRepository implements AccountRepository {...}
```

As seen from the preceding code, there are two implementations of the `AccountRepository` interface--one is `JdbcAccountRepository` and another is `JpaAccountRepository`. In this case, the Spring container will throw the following exception at startup time of the application:

```
At startup: NoSuchBeanDefinitionException, no unique bean of type
[AccountRepository] is defined: expected single bean but found
2...
```

Resolving disambiguation in Autowiring DI pattern

Spring provides one more annotation, `@Qualifier`, to overcome the problem of disambiguation in autowiring DI. Let's see the following snippet of code with the `@Qualifier` annotation:

```
@Service
public class TransferServiceImpl implements TransferService {
    @Autowired
    public TransferServiceImpl( @Qualifier("jdbcAccountRepository")
        AccountRepository accountRepository) { ... }
```

Now I have wired the dependency by name rather than by type by using the `@Qualifier` annotation. So, Spring will search the bean dependency with the name "`jdbcAccountRepository`" for the `TransferServiceImpl` class. I have given the names of the beans as follows:

```
@Repository("jdbcAccountRepository")
public class JdbcAccountRepository implements AccountRepository
{ ... }
@Repository("jpaAccountRepository")
public class JpaAccountRepository implements AccountRepository
{ ... }
```

`@Qualifier`, also available with the method injection and field injection component names, should not show implementation details unless there are two implementations of the same interface.

Let's now discuss some best practices to choose the DI pattern configuration for your Spring application.

Resolving dependency with Abstract Factory pattern

If you want to add the `if...else` conditional configuration for a bean, you can do so, and also add some custom logic if you are using Java configuration. But in the case of an XML configuration, it is not possible to add the `if...then...else` conditions. Spring provides the solution for conditions in an XML configuration by using the Abstract Factory Pattern. Use a factory to create the bean(s) you want, and use any complex Java code that you need in the factory's internal logic.

Implementing the Abstract Factory Pattern in Spring (FactoryBean interface)

The Spring Framework provides the `FactoryBean` interface as an implementation of the Abstract Factory Pattern. A `FactoryBean` is a pattern to encapsulate interesting object construction logic in a class. The `FactoryBean` interface provides a way to customize the Spring IoC container's instantiation logic. You can implement this interface for objects that are themselves factories. Beans implementing `FactoryBean` are auto-detected.

The definition of this interface is as follows:

```
public interface FactoryBean<T> {  
    T getObject() throws Exception;  
    Class<T> getObjectType();  
    boolean isSingleton();  
}
```

As per the preceding definition of this interface, the dependency injection using the `FactoryBean` and it causes `getObject()` to be invoked transparently. The `isSingleton()` method returns `true` for singleton, else it returns `false`. The `getObjectType()` method returns the object type of the object returned by the `getObject()` method.

Implementation of FactoryBean interface in Spring

FactoryBean is widely used within Spring as the following:

- EmbeddedDatabaseFactoryBean
- JndiObjectFactoryBean
- LocalContainerEntityManagerFactoryBean
- DateTimeFormatterFactoryBean
- ProxyFactoryBean
- TransactionProxyFactoryBean
- MethodInvokingFactoryBean

Sample implementation of FactoryBean interface

Suppose you have a TransferService class whose definition is thus:

```
package com.packt.patterninspring.chapter4.bankapp.service;
import com.packt.patterninspring.chapter4.
    bankapp.repository.IAccountRepository;
public class TransferService {
    IAccountRepository accountRepository;
    public TransferService(IAccountRepository accountRepository) {
        this.accountRepository = accountRepository;
    }
    public void transfer(String accountA, String accountB, Double
        amount) {
        System.out.println("Amount has been transferred");
    }
}
```

And you have a FactoryBean whose definition is thus:

```
package com.packt.patterninspring.chapter4.bankapp.repository;
import org.springframework.beans.factory.FactoryBean;
public class AccountRepositoryFactoryBean implements
    FactoryBean<IAccountRepository> {
    @Override
    public IAccountRepository getObject() throws Exception {
        return new AccountRepository();
    }
    @Override
    public Class<?> getObjectType() {
        return IAccountRepository.class;
    }
    @Override
```

```
    public boolean isSingleton() {  
        return false;  
    }  
}
```

You could wire up an `AccountRepository` instance using a hypothetical `AccountRepositoryFactoryBean` like this:

```
<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
       xmlns:c="http://www.springframework.org/schema/c"  
       xsi:schemaLocation="http://www.springframework.org/schema/beans  
                           http://www.springframework.org/schema/beans/spring-beans.xsd">  
    <bean id="transferService" class="com.packt.patterninspring.  
                                chapter4.bankapp.service.TransferService">  
        <constructor-arg ref="accountRepository"/>  
    </bean>  
    <bean id="accountRepository"  
          class="com.packt.patterninspring.chapter4.  
                bankapp.repository.AccountRepositoryFactoryBean"/>  
</beans>
```

In the preceding example, the `TransferService` class depends on the `AccountRepository` bean, but in the XML file, we have defined `AccountRepositoryFactoryBean` as an `accountRepository` bean. The `AccountRepositoryFactoryBean` class implements the `FactoryBean` interface of Spring. The result of the `getObject` method of `FactoryBean` will be passed, and not the actual `FactoryBean` itself. Spring injects that object returned by `FactoryBean`'s `getObjectType()` method, and the object type returned by `FactoryBean`'s `getObjectType()`; the scope of this bean is decided by the `FactoryBean`'s `isSingleton()` method.

The following is the same configuration for the `FactoryBean` interface in a Java Configuration:

```
package com.packt.patterninspring.chapter4.bankapp.config;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
import com.packt.patterninspring.chapter4.bankapp.  
    repository.AccountRepositoryFactoryBean;  
import com.packt.patterninspring.chapter4.  
    bankapp.service.TransferService;  
@Configuration  
public class AppConfig {  
    public TransferService transferService() throws Exception{
```

```

        return new TransferService(accountRepository().getObject());
    }
    @Bean
    public AccountRepositoryFactoryBean accountRepository() {
        return new AccountRepositoryFactoryBean();
    }
}

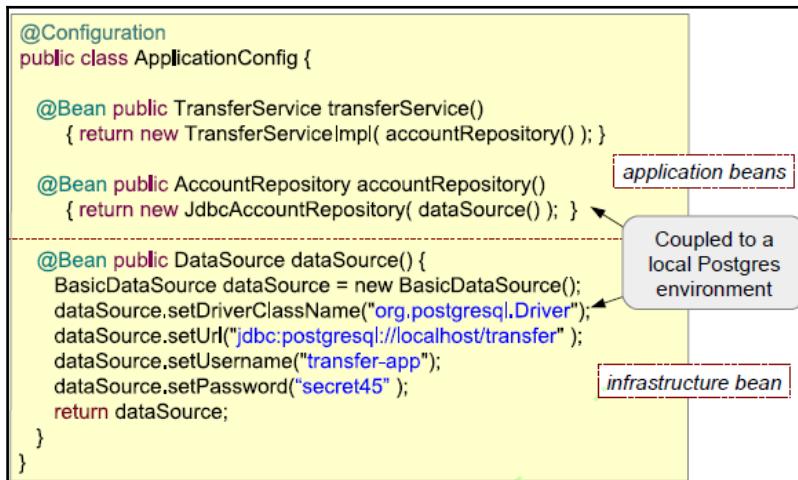
```

As other normal beans in the Spring container, the Spring FactoryBean also has all the other characteristics of any other Spring bean, including the life cycle hooks and services that all beans in the Spring container enjoy.

Best practices for configuring the DI pattern

The following are the best practices for configuring the DI pattern:

- Configuration files should be separated categorically. Application beans should be separate from infrastructure beans. Currently, it's a bit difficult to follow.



- Always specify the component name; never rely on generated names by the container.
- It is a best practice to give a name along with a description of what the pattern does, where to apply it, and the problems it addresses.

- The best practices for component scanning are as follows:
 - The components are scanned at startup, and it scans the JAR dependencies as well.
 - **Bad practice:** It scans all the packages of com and org. It increases the startup time of the application. Avoid such type of component scanning:

```
@ComponentScan (( {{ "org", "com" } } ))
```

- **Optimized:** It scans only the specific packages as defined by us.

```
@ComponentScan ( {  
    "com.packt.patterninspring.chapter4.  
    bankapp.repository",  
    "com.packt.patterninspring.chapter4.bankapp.service"  
})
```

- Best practices in choosing implicit configuration:
 - Choose annotation-based configurations for frequently changing beans
 - It allows for very rapid development
 - It is a single place to edit the configuration
- Best practices in choosing explicit via Java configuration:
 - It is centralized in one place
 - Strong type checking enforced by the compiler
 - Can be used for all classes
- Spring XML Best Practices: XML has been around for a long time, there are many shortcuts and useful techniques available in XML configuration as well, they are listed follow:
 - factory-method and factory-bean attributes
 - Bean Definition Inheritance
 - Inner Beans
 - p and c namespaces
 - Using collections as Spring beans

Summary

After reading this chapter, you should now have a good idea about DI design patterns, and the best practices for applying those patterns. Spring deals with the plumbing part, so, you can focus on solving the domain problem by using the dependency injection pattern. The DI pattern frees the object of the burden of resolving its dependencies. Your object is handed everything that it needs to work. The DI pattern simplifies your code, improves code reusability, and testability. It promotes programming to interfaces, and conceals the implementation details of dependencies. The DI pattern allows for centralized control over the object's life cycle. You can configure DI via two ways--explicit configuration and implicit configuration. Explicit configuration can be configured through XML-or Java-based configuration; it provides centralized configuration. But implicit configuration is based on annotations. Spring provides stereotype annotations for Annotation-based configuration. This configuration reduces the verbosity of code in the application, but it spreads out across the application files.

In the upcoming [Chapter 5, Understanding the Bean Life Cycle and Used Patterns](#), we will explore the life cycle of the Spring bean in the container.

12

Accessing a Database with Spring and JDBC Template Patterns

In earlier chapters, you learned about Spring core modules like the Spring IoC container, the DI pattern, container life cycle, and the used design patterns. Also you have seen how Spring makes magic using AOP. Now is the right time to move into the battlefield of real Spring applications with persisting data. Do you remember your first application during college days where you dealt with database access? That time, you probably, had to write boring boilerplate code to load database drivers, initialize your data-access framework, open connections, handle various exceptions, and to close connections. You also had to be very careful about that code. If anything went wrong, you would not have been able to make a database connection in your application, even though you would've invested a lot of time in such boring code, apart from writing the actual SQL and business code.

Because we always try to make things better and simpler, we have to focus on the solution to that tedious work for data-access. Spring comes with a solution for the tedious and boring work for data-access--it removes the code of data access. Spring provides data-access frameworks to integrate with a variety of data-access technologies. It allows you to use either JDBC directly or any **object-relational mapping (ORM)** framework, like Hibernate, to persist your data. Spring handles all the low-level code for data access work in your application; you can just write your SQL, application logic, and manage your application's data rather than investing time in writing code for making and closing database connections, and so on.

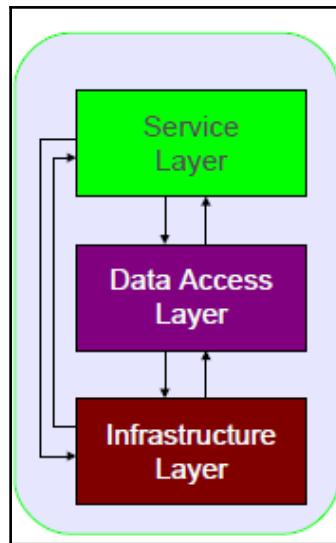
Now, you can choose any technology, such as JDBC, Hibernate, the **Java Persistence API (JPA)**, or others. to persist your application's data. Irrespective of what you choose, Spring provides support for all these technologies for your application. In this chapter, we will explore Spring's support for JDBC. It will cover the following points:

- The best approach to designing your data access
- Implementing the template design pattern
- Problems with the traditional JDBC
- Solving problems with the Spring `JdbcTemplate`
- Configuring the data source
- Using the object pool design pattern to maintain database connections
- Abstracting database access by the DAO pattern
- Working with `JdbcTemplate`
- The Jdbc callback interfaces
- Best practices for configuring `JdbcTemplate` in the application

Before we go on to discuss more about JDBC and the template design pattern, let's first see the best approach to define the data-access tier in the layered architecture.

The best approach to designing your data-access

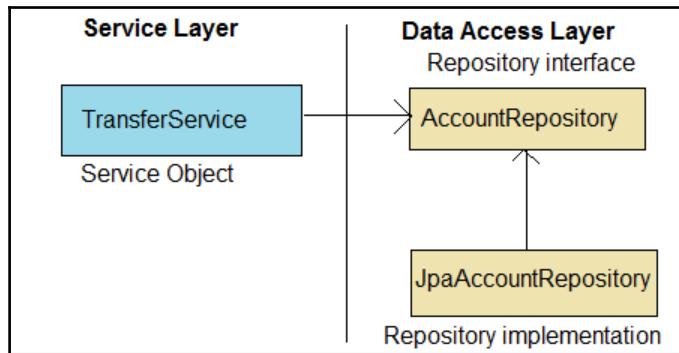
In previous chapters, you have seen that one of Spring's goals is to allow you to develop applications by following one of the OOPs principles of coding to interfaces. Any enterprise application needs to read data and write data to any kind of database, and to meet this requirement, we have to write the persistence logic. Spring allows you to avoid the scattering of persistence logic across all the modules in your application. For this, we can create a different component for data access and persistence logic, and this component is known as a **data access object (DAO)**. Let's see, in the following diagram, the best approach to create modules in layered applications:



As you can see in the preceding diagram, for a better approach, many enterprise applications consist of the following three logical layers:

- **The service layer** (or application layer): This layer of the application exposes high-level application functions like use-cases and business logic. All application services are defined here.
- **The data access layer**: This layer of the application defines an interface to the application's data repository (such as a Relational or NoSQL database). This layer has the classes and interfaces which have the data-access logic's data persisting in the application.
- **The infrastructure layer**: This layer of the application exposes low-level services to the other layers, such as configuring DataSource by using the database URL, user credentials, and so on. Such configuration comes under this layer.

In the previous figure, you can see that the **Service Layer** collaborates with the **Data Access Layer**. To avoid coupling between the application logic and data-access logic, we should expose their functionality through interfaces, as interfaces promote decoupling between the collaborating components. If we use the data-access logic by implementing interfaces, we can configure any particular data-access strategy to the application without making any changes in the application logic in the **Service Layer**. The following diagram shows the proper approach to designing our data-access layer:



As shown in the preceding figure, your application service objects, that is, **TransferService**, don't handle their own data access. Instead, they delegate data access to the repositories. The repository's interface, that is, **AccountRepository** in your application, keeps it loosely coupled to the service object. You could configure any variant of the implementations—either the Jpa implementation of **AccountRepository** (**JpaAccountRepository**), or the Jdbc implementation of **AccountRepository** (**JdbcAccountRepository**).

Spring not only provides loose coupling between the application components working at the different layers in the layered architecture, but also helps to manage the resources in the enterprise layered architecture application. Let's see how Spring manages the resources, and what design pattern is using by Spring to solve the resource management problem.

The resource management problem

Let's understand the resource management problem with the help of a real example. You must've ordered pizza online sometime. If so, what are the steps involved in the process, from the time of ordering the pizza till its delivery? There are many steps to this process--We first go to the online portal of the pizza company, select the size of the pizza and the toppings. After that, we place our order and check out. The order is accepted by the nearest pizza shop; they prepare our pizza accordingly, put the toppings on accordingly, wrap this pizza in the bag, the delivery boy comes to your place and hands over the pizza to you, and, finally, you enjoy your pizza with your friend. Even though there are many steps to this process, you're actively involved in only a couple of them. The pizza company is responsible for cooking the pizza and delivering it smoothly. You are involved only when you need to be, and other steps are taken care of by the pizza company. As you saw in this example, there are many steps involved in managing this process, and we also have to assign the resources to each step accordingly such that it is treated as a complete task without any break in the flow. This is a perfect scenario for a powerful design pattern, the template method pattern. The Spring framework implements this template design pattern to handle such type scenarios in the DAO layer of an application. Let's see what problems we face if we don't use Spring, and work with the traditional application instead.

In a traditional application, we work with the JDBC API to access the data from the database. It is a simple application where we access and persist the data using the JDBC API, and for this application, the following steps are required:

1. Define the connection parameters.
2. Access a data source, and establish a connection.
3. Begin a transaction.
4. Specify the SQL statement.
5. Declare the parameters, and provide parameter values.
6. Prepare and execute the statement.
7. Set up the loop to iterate through the results.
8. Do the work for each iteration--execute the business logic.
9. Process any exception.
10. Commit or roll back the transaction.
11. Close the connection, statement, and resultset.

If you use the Spring Framework for the same application, then you have to write the code for some steps of the preceding list of steps, while spring takes care of all the steps involving the low-level processes such as establishing a connection, beginning a transaction, processing any exception in the data layer, and closing the connection. Spring manages these steps by using the Template method design pattern, which we'll study in the next section.

Implementing the template design pattern

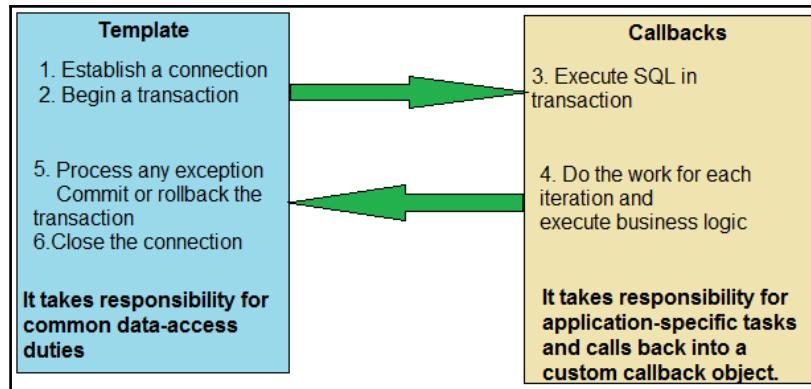
Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

-GOF Design Pattern

We discussed the Template method design pattern in [Chapter 3, Consideration of Structural and Behavioral Patterns](#). It is widely used, and comes under the structural design pattern of the GOF design pattern family. This pattern defines the outline or skeleton of an algorithm, and leaves the details to specific implementations later. This pattern hides away large amounts of boilerplate code. Spring provides many template classes, such as `JdbcTemplate`, `JmsTemplate`, `RestTemplate`, and `WebServiceTemplate`. Mostly, this pattern hides the low-level resource management as discussed earlier in the pizza example.

In the example, the process is ordering a pizza for home delivery from an online portal. The process followed by the pizza company has some fixed steps for each customer, like taking the order, preparing the pizza, adding the toppings according to the customer's specifications, and delivering it to the customer's address. We can add these steps, or define these steps to a specific algorithm. The system can then implement this algorithm accordingly.

Spring implements this pattern to access data from a database. In a database, or any other technology, there are some steps that are always common, such as establishing a connection to the database, handling transactions, handling exceptions, and some clean up actions which are required for each data access process. But there are also some steps which are not fixed, but depend on the application's requirement. It is the responsibility of the developer to define these steps. But spring allows us to separate the fixed and dynamic parts of the data-access process into different parts as templates and callbacks. All fixed steps come under the template, and dynamic custom steps come under callbacks. The following figure describes the two in detail:



As you can see in the preceding figure, all the fixed parts of the process for data access wraps to the template classes of the Spring Framework as open and close connection, open and close statements, handling exceptions, and managing resources. But the other steps like writing SQLs, declaring connection parameters, and so on are parts of the callbacks, and callbacks are handled by the developer.

Spring provides several implementations of the Template method design pattern such as `JdbcTemplate`, `JmsTemplate`, `RestTemplate`, and `WebServiceTemplate`, but in this chapter, I will explain only its implementation for the JDBC API as `JdbcTemplate`. There is another variant of `JdbcTemplate`—`NamedParameterJdbcTemplate`, which wraps a `JdbcTemplate` to provide named parameters instead of the traditional JDBC "?" placeholders.

Problems with the traditional JDBC

The following are the problems we have to face whenever we work with the traditional JDBC API:

- **Redundant results due to error-prone code:** The traditional JDBC API required a lot of tedious code to work with the data access layer. Let's see the following code to connect the Database and execute the desired query:

```

public List<Account> findByAccountNumber(Long accountNumber) {
    List<Account> accountList = new ArrayList<Account>();
    Connection conn = null;
    String sql = "select account_name,
    account_balance from ACCOUNT where account_number=?";
    try {
        DataSource dataSource = DataSourceUtils.getDataSource();
        conn = dataSource.getConnection();

```

```
PreparedStatement ps = conn.prepareStatement(sql);
ps.setLong(1, accountNumber);
ResultSet rs = ps.executeQuery();
while (rs.next()) {
    accountList.add(new Account(rs.getString(
        "account_name"), ...));
}
} catch (SQLException e) { /* what to be handle here? */ }
finally {
    try {
        conn.close();
    } catch (SQLException e) { /* what to be handle here ?*/ }
}
return accountList;
}
```

As you can see in the preceding code, there are some lines which are highlighted; only this bold code matters—the rest is boilerplate. Also, this code handles the SQLException in the application inefficiently, because the developer doesn't know what should be handled there. Let's now look at another problem in the traditional JDBC code.

- **Leads to poor exception handling:** In the preceding code, the exceptions in the application are handled very poorly. The developers are not aware of what exceptions are to be handled here. SQLException is a checked Exception, which means it forces the developers to handle errors, but if you can't handle it, you must declare it. It is a very bad way of handling exceptions, and the intermediate methods must declare exception(s) from all methods in the code. It is a form of tight coupling.

Solving problems with Spring's JdbcTemplate

Spring's JdbcTemplate solves both the problems listed in the last section.

JdbcTemplate greatly simplifies the use of the JDBC API, and it eliminates repetitive boilerplate code. It alleviates the common causes of bugs, and handles SQLExceptions properly without sacrificing power. It provides full access to the standard JDBC constructs. Let's see the same code using Spring's JdbcTemplate class to solve these two problems:

- **Removing redundant code from the application using JdbcTemplate:** Suppose you want a count of the accounts in a bank. The following code is required if you use the JdbcTemplate class:

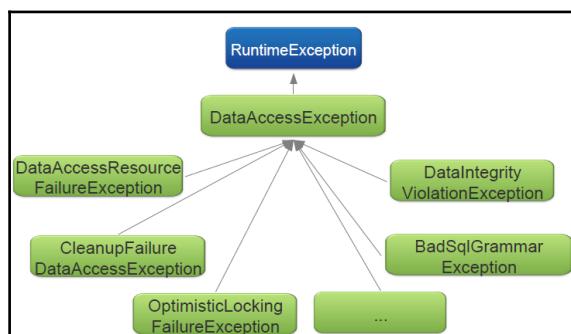
```
int count = jdbcTemplate.queryForObject("SELECT COUNT(*)  
FROM ACCOUNT", Integer.class);
```

If you want to access the list of accounts for a particular user ID:

```
List<Account> results = jdbcTemplate.query(someSql,  
new RowMapper<Account>() {  
    public Account mapRow(ResultSet rs, int row) throws  
        SQLException {  
            // map the current row to an Account object  
        }  
});
```

As you can see in the preceding code, you don't need to write the code for Open and Close database connection, for preparing a statement to execute query, and so on.

- **Data Access Exceptions:** Spring provides a consistent exception hierarchy to handle technology-specific exceptions like `SQLException` to its own exception class hierarchy with `DataAccessException` as the root exception. Spring wraps these original exceptions into different unchecked exceptions. Now Spring does not force the developers to handle these exceptions at development time. Spring provides the `DataAccessException` hierarchy to hide whether you are using JPA, Hibernate, JDBC, or similar. Actually, it is a hierarchy of sub-exceptions, and not just one exception for everything. It is consistent across all the supported data access technologies. The following diagram depicts the Spring Data Access Exception hierarchy:



- As you can see in the preceding figure, Spring's `DataAccessException` extends the `RuntimeException`, that is, it is an unchecked exception. In an enterprise application, unchecked exceptions can be thrown up the call hierarchy to the best place to handle it. The good thing is that the methods in between don't know about it in the application.

Let's first discuss how to configure Spring with a data source to be able to connect the database, before declaring the templates and repositories in a Spring application.

Configuring the data source and object pool pattern

In the Spring Framework, `DataSource` is part of the JDBC API, and it provides a connection to the database. It hides many boilerplate codes for connection pooling, exception handling, and transaction management issues from the application code. As a developer, you let it focus on your business logic only. Don't worry about connection pooling, exception handling, and managing transactions; it is the responsibility of the application administrators how they set up the container managed data source in production. You just write the code, and test that code.

In an enterprise application, we can retrieve `DataSource` in several ways. We can use the JDBC driver to retrieve `DataSource`, but it is not the best approach to create `DataSource` in the production environment. As performance is one of the key issues during application development, Spring implements the object pool pattern to provide `DataSource` to the application in a very efficient way. The object pool pattern says that *creation of objects is expensive rather than reuse*.

Spring allows us to implement the object pool pattern for reusing the `DataSource` object in the application. You can use either the application server and container-managed pool (JNDI), or you can create a container by using third-party libraries such as DBCP, c3p0, and so on. These pools help to manage the available data sources in a better way.

In your Spring application, there are several options to configure the data-source beans, and they are as follows:

- Configuring data source using a JDBC driver
- Implementing the object pool design pattern to provide data source objects
 - Configuring the data source using JNDI
 - Configuring the data source using pool connections
 - Implementing the Builder pattern to create an embedded data source
- Let's see how to configure a data-source bean in a Spring application.

Configuring a data source using a JDBC driver

Using a JDBC driver to configure a data-source bean is the simplest data source in Spring. The three data source classes provided by Spring are as follows:

- `DriverManagerDataSource`: It always creates a new connection for every connection request
- `SimpleDriverDataSource`: It is similar to the `DriverManagerDataSource` except that it works with the JDBC driver directly
- `SingleConnectionDataSource`: It returns the same connection for every connection request, but it is not a pooled data source

Let's see the following code for configuring a data source bean using the `DriverManagerDataSource` class of Spring in your application:

In Java-based configuration, the code is as follows:

```
DriverManagerDataSource dataSource = new
DriverManagerDataSource();
dataSource.setDriverClassName("org.h2.Driver");
dataSource.setUrl("jdbc:h2:tcp://localhost/bankDB");
dataSource.setUsername("root");
dataSource.setPassword("root");
```

In XML-based configuration, the code will be like this:

```
<bean id="dataSource"
      class="org.springframework.jdbc.datasource
      .DriverManagerDataSource">
    <property name="driverClassName" value="org.h2.Driver"/>
    <property name="url" value="jdbc:h2:tcp://localhost/bankDB"/>
    <property name="username" value="root"/>
    <property name="password" value="root"/>
</bean>
```

The data source defined in the preceding code is a very simple data source, and we can use it in the development environment. It is not a suitable data source for production. I, personally, prefer to use JNDI to configure the data source for the production environment. Let's see how.

Let's implement the object pool design pattern to provide data source objects *by* configuring the data source *using* JNDI.

In a Spring application, you can configure a data source by using the JNDI lookup. Spring provides the `<jee:jndi-lookup>` element from Spring's JEE namespace. Let's see the code for this configuration.

In XML configuration, the code is given as follows:

```
<jee:jndi-lookup id="dataSource"
      jndi-name="java:comp/env/jdbc/datasource" />
```

In Java configuration, the code is as follows:

```
@Bean
public JndiObjectFactoryBean dataSource() {
    JndiObjectFactoryBean jndiObject = new JndiObjectFactoryBean();
    jndiObject.setJndiName("jdbc/datasource");
    jndiObject.setResourceRef(true);
    jndiObject.setProxyInterface(javax.sql.DataSource.class);
    return jndiObject;
}
```

Application servers like WebSphere or JBoss allow you to configure data sources to be prepared via JNDI. Even a web container like Tomcat allows you to configure data sources to be prepared via JNDI. These servers manage the data sources in your application. It is beneficial, because the performance of the data source will be greater, as the application servers are often pooled. And they can be managed completely external to the application. This is one of the best ways to configure a data source to be retrieved via JNDI. If you are not able to retrieve through the JNDI lookup in production, you can choose another, better option, which we'll discuss next.

Configuring the data source using pool connections

The following open-sources technologies provide pooled data sources:

- Apache commons DBCP
- c3p0
- BoneCP

The following code configures DBCP's BasicDataSource.

The XML-based DBCP configuration is given as follows:

```
<bean id="dataSource"
      class="org.apache.commons.dbcp.BasicDataSource"
      destroy-method="close">
    <property name="driverClassName" value="org.h2.Driver"/>
    <property name="url" value="jdbc:h2:tcp://localhost/bankDB"/>
    <property name="username" value="root"/>
    <property name="password" value="root"/>
    <property name="initialSize" value="5"/>
    <property name="maxActive" value="10"/>
</bean>
```

The Java-based DBCP configuration is as follows:

```
@Bean
public BasicDataSource dataSource() {
    BasicDataSource dataSource = new BasicDataSource();
    dataSource.setDriverClassName("org.h2.Driver");
    dataSource.setUrl("jdbc:h2:tcp://localhost/bankDB");
    dataSource.setUsername("root");
    dataSource.setPassword("root");
    dataSource.setInitialSize(5);
    dataSource.setMaxActive(10);
    return dataSource;
}
```

As you can see in the preceding code, there are many other properties which are introduced for a pooled data sources provider. The properties of the `BasicDataSource` class in Spring are listed next:

- `initialSize`: This is the number of connections created at the time of initialization of the pool.
- `maxActive`: This is the maximum number of connections that can be allocated from the pool at the time of initialization of the pool. If you set this value to 0, that means there's no limit.
- `maxIdle`: This is the maximum number of connections that can be idle in the pool without extras being released. If you set this value to 0, that means there's no limit.
- `maxOpenPreparedStatements`: This is the maximum number of prepared statements that can be allocated from the statement pool at the time of initialization of the pool. If you set this value to 0, that means there's no limit.
- `maxWait`: This is the maximum waiting time for a connection to be returned to the pool before an exception is thrown. If you set it to 1, it means wait indefinitely.
- `minEvictableIdleTimeMillis`: This is the maximum time duration a connection can remain idle in the pool before it's eligible for eviction.
- `minIdle`: This is the minimum number of connections that can remain idle in the pool without new connections being created.

Implementing the Builder pattern to create an embedded data source

In application development, the embedded database is very useful, because it doesn't require a separate database server that your application connects. Spring provides one more data source for embedded databases. It is not powerful enough for the production environment. We can use the embedded data source for the development and testing environment. In Spring, the `jdbcTemplate` namespace configures an embedded database, H2, as follows:

In XML configuration, H2 is configured as follows:

```
<jdbc:embedded-database id="dataSource" type="H2">
    <jdbc:script location="schema.sql"/>
    <jdbc:script location="data.sql"/>
</jdbc:embedded-database>
```

In Java configuration, H2 is configured as follows:

```
@Bean
public DataSource dataSource() {
    EmbeddedDatabaseBuilder builder =
        new
    EmbeddedDatabaseBuilder().setType(EmbeddedDatabaseType.H2);
    builder.addScript("schema.sql");
    builder.addScript("data.sql");
    return builder.build();
}
```

As you can see in the preceding code, Spring provides the `EmbeddedDatabaseBuilder` class. It actually implements the Builder design pattern to create the object of the `EmbeddedDatabaseBuilder` class.

Let's see one more design pattern in the next section.

Abstracting database access using the DAO pattern

The data access layer works as an aspect between the business layer and the database. Data accessing depends on the business call, and it varies depending on the source of the data for example database, flat files, XML, and so on. So, we can abstract all access by providing an interface. This is known as the data access object pattern. From the application's point of view, it makes no difference when it accesses a relational database or parses XML files using a DAO.

In an earlier version, EJB provided entity beans managed by the container; they were distributed, secure, and transactional components. These beans were very transparent to the client, that is, for the service layer in the application, they had automatic persistence without the care of underlying database. But mostly, the features offered by these entity beans were not required for your application, as you needed to persist data to the database. Due to *this*, some non-required features of the entity beans, like network traffic, increased, and your application's performance was impacted. And that time, the entity beans needed to run inside the EJB containers, which is why it was very difficult to test.

In a nutshell, if you are working with the traditional JDBC API or earlier EJB versions, you will face the following problems in your application:

- In a traditional JDBC application, you merge the business tier logic with persistence logic.
- The Persistence tier or DAO layer is not consistent for the service layer or business tier. But DAO should be consistent for the service layer in an enterprise application.
- In a traditional JDBC application, you have to handle a lot of boilerplate code like making and closing connection, preparing statement, handling exceptions, and so on. It degrades reusability and increases development time.
- With EJB, the entity bean was created *as* an overhead to the application, and was difficult to test.

Let's see how spring solves these problems.

The DAO pattern with the Spring Framework

Spring provides a comprehensive JDBC module to design and develop JDBC-based DAOs. These DAOs in the application take care of all the boilerplate code of the JDBC API, and help to provide a consistent API for data access. In the Spring JDBC, DAO is a generic object to access data for the business tier, and it provides a consistent interface to the services at the business tier. The main goal behind the DAO's classes is to abstract the underlying data access logic from the services at the business tier.

In our previous example, we saw how the pizza company helped us to understand the resource management problem, and now, we will continue with our bank application. Let's see the following example on how to implement DAOs in an application. Suppose, in our bank application, we want the total number accounts in a branch in the city. For this, we will first create an interface for the DAO. It promotes programming to interface, as discussed earlier. It is one of the best practices of the design principles. This DAO interface will be injected with the services at the business tier, and we can create a number of concrete classes of the DAO interface according to the underlying databases in the application. That means our DAO layer will be consistent for the business layer. Let's create a DAO interface as following:

```
package com.packt.patterninspring.chapter7.bankapp.dao;
public interface AccountDao {
    Integer totalAccountsByBranch(String branchName);
}
```

Let's see a concrete implementation of the DAO interface using Spring's `JdbcDaoSupport` class:

```
package com.packt.patterninspring.chapter7.bankapp.dao;

import org.springframework.jdbc.core.support.JdbcDaoSupport;
public class AccountDaoImpl extends JdbcDaoSupport implements
    AccountDao {
    @Override
    public Integer totalAccountsByBranch(String branchName) {
        String sql = "SELECT count(*) FROM Account WHERE branchName =
            "+branchName;
        return this.getJdbcTemplate().queryForObject(sql,
            Integer.class);
    }
}
```

In the preceding code, you can see that the `AccountDaoImpl` class implements the `AccountDao` DAO interface, and it extends Spring's `JdbcDaoSupport` class to ease development with JDBC-based. This class provides a `JdbcTemplate` to its subclasses by using `getJdbcTemplate()`. The `JdbcDaoSupport` class is associated with a data source, and supplies the `JdbcTemplate` object for use in the DAO.

Working with `JdbcTemplate`

As you learned earlier, Spring's `JdbcTemplate` solves two main problems in the application. It solves the redundant code problem as well as poor exception handling of the data access code in the application. Without `JdbcTemplate` in your application, only 20% of the code is required for querying a row, but 80% is boilerplate which handles exceptions and manages resources. If you use `JdbcTemplate`, then there is no need to worry about the 80% boilerplate code. Spring's `JdbcTemplate`, in a nutshell, is responsible for the following:

- Acquisition of the connection
- Participation in the transaction
- Execution of the statement
- Processing of the result set
- Handling any exceptions
- Release of the connection

Let's see when to use `JdbcTemplate` in the application, and how to create it.

When to use `JdbcTemplate`

`JdbcTemplate` is useful in standalone applications, and anytime when JDBC is needed. It is suitable in utility or test code to clean up messy legacy code. Also, in any layered application, you can implement a repository or data access object. Let's see how to create it in an application.

Creating a JdbcTemplate in an application

If you want to create an object of the `JdbcTemplate` class to access data in your Spring application, you need to remember that it requires a `DataSource` to create the database connection. Let's create a template once, and reuse it. Do not create one for each thread, it is thread-safe after construction:

```
JdbcTemplate template = new JdbcTemplate(dataSource);
```

Let's configure a `JdbcTemplate` bean in Spring with the following `@Bean` method:

```
@Bean
public JdbcTemplate jdbcTemplate(DataSource dataSource) {
    return new JdbcTemplate(dataSource);
}
```

In the preceding code, we use the constructor injection to inject the `DataSource` with the `JdbcTemplate` bean in the Spring application. The `dataSource` bean being referenced can be any implementation of `javax.sql.DataSource`. Let's see how to use the `JdbcTemplate` bean in your JDBC-based repository to access the database in your application.

Implementing a JDBC-based repository

We can use the Spring's `JdbcTemplate` class to implement the repositories in a Spring application. Let's see how to implement the repository class based on the JDBC template:

```
package com.packt.patterninspring.chapter7.bankapp.repository;

import java.sql.ResultSet;
import java.sql.SQLException;

import javax.sql.DataSource;

import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.stereotype.Repository;
import com.packt.patterninspring.chapter7.bankapp.model.Account;
@Repository
public class JdbcAccountRepository implements AccountRepository{
    JdbcTemplate jdbcTemplate;
    public JdbcAccountRepository(DataSource dataSource) {
        super();
        this.jdbcTemplate = new JdbcTemplate(dataSource);
```

```
}

@Override
public Account findAccountById(Long id) {
    String sql = "SELECT * FROM Account WHERE id = "+id;
    return jdbcTemplate.queryForObject(sql,
        new RowMapper<Account>() {
            @Override
            public Account mapRow(ResultSet rs, int arg1) throws
SQLException {
                Account account = new Account(id);
                account.setName(rs.getString("name"));
                account.setBalance(new Long(rs.getInt("balance")));
                return account;
            }
        });
}
```

In the preceding code, the `DataSource` bean is injected with the `JdbcAccountRepository` class by using the constructor injection. By using this `DataSource`, we created a `JdbcTemplate` object for accessing the data. The following methods are provided by `JdbcTemplate` to access data from the database:

- `queryForObject(...)`: This is a query for simple java types (`int`, `long`, `String`, `Date` ...) and for custom domain objects.
- `queryForMap(...)`: This is used when expecting a single row. `JdbcTemplate` returns each row of a `ResultSet` as a `Map`.
- `queryForList(...)`: This is used when expecting multiple rows.



Note that `queryForInt` and `queryForLong` have been deprecated since Spring 3.2; you can just use `queryForObject` instead (API improved in Spring 3).

Often, it is useful to map relational data into domain objects, for example, a `ResultSet` to an `Account` in the last code. Spring's `JdbcTemplate` supports this by using a callback approach. Let's discuss Jdbc callback interfaces in the next section.

Jdbc callback interfaces

Spring provides three callback interfaces for JDBC as follows:

- **Implementing RowMapper:** Spring provides a `RowMapper` interface for mapping a single row of a `ResultSet` to an object. It can be used for both single and multiple row queries. It is parameterized as of Spring 3.0:

```
public interface RowMapper<T> {  
    T mapRow(ResultSet rs, int rowNum)  
    throws SQLException;  
}
```

- Let's understand this with the help of an example.

Creating a RowMapper class

In the following example, a class, `AccountRowMapper`, implements the `RowMapper` interface of the Spring Jdbc module:

```
package com.packt.patterninspring.chapter7.bankapp.rowmapper;  
  
import java.sql.ResultSet;  
import java.sql.SQLException;  
import org.springframework.jdbc.core.RowMapper;  
import com.packt.patterninspring.chapter7.bankapp.model.Account;  
public class AccountRowMapper implements RowMapper<Account>{  
    @Override  
    public Account mapRow(ResultSet rs, int id) throws SQLException  
{  
    Account account = new Account();  
    account.setId(new Long(rs.getInt("id")));  
    account.setName(rs.getString("name"));  
    account.setBalance(new Long(rs.getInt("balance")));  
    return account;  
}  
}
```

In the preceding code, a class, `AccountRowMapper`, maps a row of the result set to the domain object. This row-mapper class implements the `RowMapper` callback interface of the Spring Jdbc module.

Query for single row with JdbcTemplate

Let's now see how the row-mapper maps a single row to the domain object in the application in the following code:

```
public Account findAccountById(Long id){  
    String sql = "SELECT * FROM Account WHERE id = "+id;  
    return jdbcTemplate.queryForObject(sql, new AccountRowMapper());  
}
```

Here, there is no need to add typecasting for the Account object. The AccountRowMapper class maps the rows to the Account objects.

Query for multiple rows

The following code shows how the row mapper maps multiple rows to the list of domain objects:

```
public List<Account> findAccountById(Long id){  
    String sql = "SELECT * FROM Account ";  
    return jdbcTemplate.queryForList(sql, new AccountRowMapper());  
}
```

RowMapper is the best choice when each row of a ResultSet maps to a domain object.

Implementing RowCallbackHandler

Spring provides a simpler RowCallbackHandler interface when there is no return object. It is used to stream rows to a file, converting the rows to XML, and filtering them before adding to a collection. But filtering in SQL is much more efficient, and is faster than the JPA equivalent for big queries. Let's look at the following example:

```
public interface RowCallbackHandler {  
    void processRow(ResultSet rs) throws SQLException;  
}
```

Example for using a RowCallbackHandler

The following code is an example of a RowCallbackHandler in the application:

```
package com.packt.patterninspring.chapter7.bankapp.callbacks;  
import java.sql.ResultSet;  
import java.sql.SQLException;  
import org.springframework.jdbc.core.RowCallbackHandler;  
public class AccountReportWriter implements RowCallbackHandler {
```

```
    public void processRow(ResultSet resultSet) throws SQLException
    {
        // parse current row from ResultSet and stream to output
        //write flat file, XML
    }
}
```

In preceding code, we have created a `RowCallbackHandler` implementation; the `AccountReportWriter` class implements this interface to process the result set returned from the database. Let's see the following code how to use `AccountReportWriter` call back class:

```
@Override
public void generateReport(Writer out, String branchName) {
    String sql = "SELECT * FROM Account WHERE branchName = "+
        branchName;
    jdbcTemplate.query(sql, new AccountReportWriter());
}
```

`RowCallbackHandler` is the best choice when no value should be returned from the callback method for each row, especially for large queries.

Implementing ResultSetExtractor

Spring provides a `ResultSetExtractor` interface for processing an entire `ResultSet` at once. Here, you are responsible for iterating the `ResultSet`, for example, for mapping the entire `ResultSet` to a single object. Let's see the following example:

```
public interface ResultSetExtractor<T> {
    T extractData(ResultSet rs) throws SQLException,
        DataAccessException;
}
```

Example for using a ResultSetExtractor

The following line of code implements the `ResultSetExtractor` interface in the application:

```
package com.packt.patterninspring.chapter7.bankapp.callbacks;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;
```

```
import org.springframework.dao.DataAccessException;
import org.springframework.jdbc.core.ResultSetExtractor;

import com.packt.patterninspring.chapter7.bankapp.model.Account;

public class AccountExtractor implements
    ResultSetExtractor<List<Account>> {
    @Override
    public List<Account> extractData(ResultSet resultSet) throws
        SQLException, DataAccessException {
        List<Account> extractedAccounts = null;
        Account account = null;
        while (resultSet.next()) {
            if (extractedAccounts == null) {
                extractedAccounts = new ArrayList<>();
                account = new Account(resultSet.getLong("ID"),
                    resultSet.getString("NAME"), ...);
            }
            extractedAccounts.add(account);
        }
        return extractedAccounts;
    }
}
```

This preceding class, `AccountExtractor`, implements `ResultSetExtractor`, and it is used to create an object for the entire data of the result set returned from the database. Let's see how to use this class in your application:

```
public List<Account> extractAccounts() {
    String sql = "SELECT * FROM Account";
    return jdbcTemplate.query(sql, new AccountExtractor());
}
```

The previous code is responsible for accessing all the accounts of a bank, and for preparing a list of accounts by using the `AccountExtractor` class. This class implements the `ResultSetExtractor` callback interface of the Spring Jdbc module.

`ResultSetExtractor` is the best choice when multiple rows of a `ResultSet` map to a single object.

Best practices for Jdbc and configuring JdbcTemplate

Instances of the `JdbcTemplate` class are thread-safe once configured. As a best practice of configuring the `JdbcTemplate` in a Spring application, it should be constructed in the constructor injection or setter injection of the data source bean in your DAO classes by passing that data source bean as a constructor argument of the `JdbcTemplate` class. This leads to DAOs that look, in part, like the following:

```
@Repository  
public class JdbcAccountRepository implements AccountRepository{  
    JdbcTemplate jdbcTemplate;  
    public JdbcAccountRepository(DataSource dataSource) {  
        super();  
        this.jdbcTemplate = new JdbcTemplate(dataSource);  
    }  
    //...  
}  
Let's see some best practices to configure a database and write the code for the DAO layer:
```

- If you want to configure the embedded database at the time of development of the application, as the best practice, the embedded database will always be assigned a uniquely generated name. This is because in the Spring container, the embedded database is made available by configuring a bean of type `javax.sql.DataSource`, and that data source bean is injected to the data access objects.
- Always use object pooling; this can be achieved in two ways:
 - **Connection pooling:** It allows the pool manager to keep the connections in a *pool* after they are closed
 - **Statement pooling:** It allows the driver to reuse the prepared Statement objects.
 - Choose the commit mode carefully
 - Consider removing the auto-commit mode for your application, and use manual commit instead to better control the commit logic, as follows:

```
Connection.setAutoCommit(false);
```

Summary

An application without data is like a car without fuel. Data is the heart of an application. Some applications may exist in the world without data, but these applications are simply showcase applications such as static blogs. Data is an important part of an application, and you need to develop data-access code for your application. This code should very simple, robust, and customizable.

In a traditional Java application, you could use JDBC to access the data. It is a very basic way, but sometimes, it is very messy to define specifications, handle JDBC exceptions, make database connections, load drivers, and so on. Spring simplifies these things by removing the boilerplate code and simplifying JDBC exception handling. You just write your SQL that should be executed in the application, and the rest is managed by the Spring framework.

In this chapter, you have seen how Spring provides support at the backend for data access and data persistence. JDBC is useful, but using the JDBC API directly is a tedious and error-prone task. `JdbcTemplate` simplifies data access, and enforces consistency. Data access with Spring uses the layered architecture principles—the higher layers should not know about data management. It isolates `SQLException` via Data Access Exceptions, and creates a hierarchy to make them easier to handle.

In the next chapter, we'll continue to discuss data access and persistence with the ORM framework, like Hibernate and JPA.

13

Improving Application Performance Using Caching Patterns

In previous chapters, we have seen how Spring works in the backend to access data for the application. We also saw how the Spring JDBC Module provides the `JdbcTemplate` helper class for database access. Spring provides support for integration with ORM solutions such as Hibernate, JPA, JDO, and so on, and manages transactions across application. Now, in this chapter, we will see how Spring provides caching support to improve application performance.

Do you ever face a volley of questions from your wife when you return home very late in the night from your office? Yes, I know it is very irritating to answer so many questions when you are tired and exhausted. It is even more irritating when you're asked the same questions over and over again..

Some questions can be answered with a *Yes* or *No*, but for some questions, you have to explain in detail. Consider what will happen if you are asked another lengthy question again after some time! Similarly, there are some stateless components in an application, where the components have been designed in such a way that they ask the same questions over and over again to complete each task individually. Similar to some questions asked by your wife, some questions in the system take a while to fetch the appropriate data--it may have some major complex logic behind it, or maybe, it has to fetch data from the database, or call a remote service.

If we know that the answer of a question is not likely to change frequently, we can remember the answer to that question for later when it is asked again by the same system. It doesn't make sense to go through the same channel to fetch it again, as it will impact your application's performance, and will be a wasteful use of your resources. In an enterprise application, caching is a way to store those frequently needed answers so that we fetch from the cache instead of going through the proper channel to get the answer for the same question over and over again. In this chapter, we will discuss Spring's Cache Abstraction feature, and how Spring declaratively supports caching implementation. It will cover the following points:

- What is a cache?
- Where do we do this caching?
- Understanding the cache abstraction
- Enabling caching via the Proxy pattern
- Declarative Annotation-based caching
- Declarative XML-based caching
- Configuring the cache storage
- Implementing custom cache annotations
- Caching best practices

Let's begin.

What is cache?

In very simple terms, **cache** is a memory block where we store preprocessed information for the application. In this context, a key-value storage, such as a map, may be a cache in the application. In Spring, cache is an interface to abstract and represent caching. A cache interface provides some methods for placing objects into a cache storage, it can retrieve from the cache storage for given key, it can update the object in the cache storage for a given key, it remove the object from the cache storage for a given key. This cache interface provides many functions to operate with cache.

Where do we use caching?

We use caching in cases where a method always returns the same result for the same argument(s). This method could do anything such as calculate data on the fly, execute a database query, and request data via RMI, JMS, and a web-service, and so on. A unique key must be generated from the arguments. That's the cache key.

Understanding cache abstraction

Basically, caching in Java applications is applied to the Java methods to reduce the number of executions for the same information available in the cache. That means, whenever these Java methods are invoked, the cache abstraction applies the cache behavior to these methods based on the given arguments. If the information for the given argument is already available in the cache, then it is returned without having to execute the target method. If the required information is not available in the cache, then the target method is executed, and the result is cached and returned to the caller. Cache abstraction also provides other cache-related operations such as updating and/or removing the contents in the cache. These operations are useful when the data changes in the application sometimes.

Spring Framework provides cache abstraction for Spring applications by using the `org.springframework.cache.Cache` and `org.springframework.cache.CacheManager` interfaces. Caching requires the use of an actual storage to store the cache data. But cache abstraction only provides caching logic. It doesn't provide any physical storage to store the cached data. So, developers need to implement the actual storage for caching in the application. If you have a distributed application, then you will need to configure your cache provider accordingly. It depends on the use cases of your application. You can either make a copy of the same data across nodes for a distributed application, or you can make a centralized cache.

There are several cache providers in the market, which you could use as per as your application requirement. Some of them are as follows:

- Redis
- OrmLiteCacheClient
- Memcached
- In Memory Cache
- Aws DynamoDB Cache Client
- Azure Cache Client

To implement cache abstraction in your application, you have to take care of the following tasks:

- **Caching declaration:** This means that you have to recognize those methods in the application that need to be cached, and annotate these methods either with caching annotations, or you can use XML configuration by using Spring AOP

- **Cache configuration:** This means that you have to configure the actual storage for the cached data--the storage where the data is stored and read from

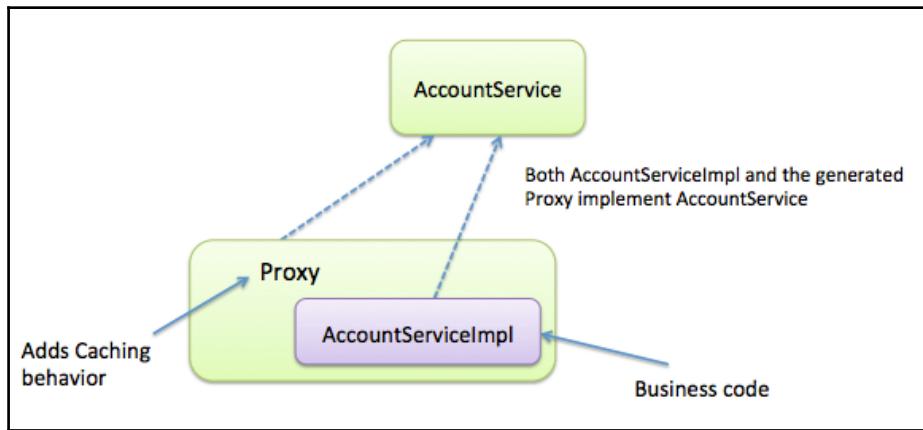
Let's now see how we can enable Spring's cache abstraction in a Spring application.

Enabling caching via the Proxy pattern

You can enable Spring's cache abstraction in the following two ways:

- Using Annotation
- Using the XML namespace

Spring transparently applies caching to the methods of Spring beans by using AOP. Spring applies proxy around the Spring beans where you declare the methods that need to be cached. This proxy adds the dynamic behavior of caching to the Spring beans. The following diagram illustrates the caching behavior:



In the preceding diagram, you can see that Spring applies **Proxy** to the **AccountServiceImpl** class to add the caching behavior. Spring uses the GoF proxy pattern to implement caching in the application.

Let's look at how to enable this feature in a Spring application.

Enabling the caching proxy using Annotation

As you already know, Spring provides lots of features, but they are, mostly, disabled. You must enable these feature before using it. If you want to use Spring's cache abstraction in your application, you have to enable this feature. If you are using Java configuration, you can enable cache abstraction of Spring by adding the `@EnableCaching` annotation to one of your configuration classes. The following configuration class shows the `@EnableCaching` annotation:

```
package com.packt.patterninspring.chapter9.bankapp.config;

import org.springframework.cache.CacheManager;
import org.springframework.cache.annotation.EnableCaching;
import org.springframework.cache.concurrent.
    ConcurrentMapCacheManager;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan(basePackages=
{"com.packt.patterninspring.chapter9.bankapp"})
@EnableCaching //Enable caching
public class AppConfig {
    @Bean
    public AccountService accountService() { ... }

    //Declare a cache manager
    @Bean
    public CacheManager cacheManager() {
        CacheManager cacheManager = new ConcurrentMapCacheManager();
        return cacheManager;
    }
}
```

In the preceding Java configuration file, we added the `@EnableCaching` annotation to the configuration class `AppConfig.java`; this annotation indicates to the Spring Framework to enable Spring cache behavior for the application.

Let's now look at how to enable Spring's cache abstraction by using XML configuration.

Enabling the Caching Proxy using the XML namespace

If you're configuring your application with XML, you can enable annotation-driven caching with the `<cache:annotation-driven>` element from Spring's cache namespace, as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:jdbc="http://www.springframework.org/schema/jdbc"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:cache="http://www.springframework.org/schema/cache"
    xsi:schemaLocation="http://www.springframework.org/schema/jdbc
    http://www.springframework.org/schema/jdbc/spring-jdbc-4.3.xsd
    http://www.springframework.org/schema/cache
    http://www.springframework.org/schema/cache/spring-cache-4.3.xsd
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-4.3.xsd
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx-4.3.xsd">
    <!-- Enable caching -->
    <cache:annotation-driven />
    <context:component-scan base-
        package="com.packt.patterninspring.chapter9.bankapp"/>
    <!-- Declare a cache manager -->
    <bean id="cacheManager"
        class="org.springframework.cache.concurrent.
        ConcurrentMapCacheManager" />
</beans>
```

As seen in the preceding configuration files, whether you use Java configuration or XML configuration, the annotation `@EnableCaching` and namespace `<cache:annotation-driven>` enables Spring's cache abstraction by creating an aspect with pointcuts that trigger off of Spring's caching annotations.

Let's see how to use Spring's caching annotations to define cache boundaries.

Declarative Annotation-based caching

In Spring applications, Spring's abstraction provides the following Annotations for caching declaration:

- `@Cacheable`: This indicates that before execution of the actual method, look at the return value of that method in the cache. If the value is available, return this cached value, if the value is not available, then invoke the actual method, and put the returned value into the cache.
- `@CachePut`: This updates the cache without checking if the value is available or not. It always invokes the actual method.
- `@CacheEvict`: This is responsible for triggering cache eviction.
- `@Caching`: This is used for grouping multiple annotations to be applied on a method at once.
- `@CacheConfig`: This indicates to Spring to share some common cache-related settings at the class level.

Let us now take a closer look at each annotation.

The `@Cacheable` annotation

`@Cacheable` marks a method for caching. Its result is stored in a cache. For all subsequent invocations of that method with the same arguments, it will fetch data from the cache using a key. The method will not be executed. The following are the `@Cacheable` attributes:

- **value**: This is the name of cache to use
- **key**: This is the key for each cached data item
- **condition**: This is a SpEL expression to evaluate true or false; if it is false, then the result of caching is not applied to the method call
- **unless**: This too is a SpEL expression; if it is true, it prevents the return value from being put in the cache

You can use SpEL and argument(s) of method. Let's look at the following code for the simplest declaration of the `@Cacheable` annotation. It requires the name of the cache associated with that method. Please refer to the following code:

```
@Cacheable("accountCache ")
public Account findAccount(Long accountId) { ... }
```

In the preceding code, the `findAccount` method is annotated with the `@Cacheable` annotation. This means that this method is associated with a cache. The name of the cache is `accountCache`. Whenever this method is called for a particular `accountId`, the cache is checked for the return value of this method for the given `accountId`. You can also give multiple names to the cache as shown next:

```
@Cacheable({"accountCache ", "saving-accounts"})
public Account findAccount(Long accountId) {...}
```

The `@CachePut` annotation

As mentioned earlier, the `@Cacheable` and `@CachePut` annotations both have the same goal, that is, to populate a cache. But their working is slightly different from each other. `@CachePut` marks a method for caching, and its result is stored in a cache. For each invocation of that method with the same arguments, it always invokes the actual method without checking whether the return value of that method is available in the cache or not. The following are `@CachePut` attributes:

- **value:** This is the name of the cache to use
- **key:** This is the key for each cached data item
- **condition:** This is a SpEL expression to evaluate true or false; if false, then the result of caching is not applied to the method call
- **unless:** This is also a SpEL expression; if it is true, it prevents the return value from being put in the cache

You can also use SpEL and argument(s) of method for the `@CachePut` annotation. The following code is the simplest declaration of the `@CachePut` annotation:

```
@CachePut("accountCache ")
public Account save(Account account) {...}
```

In the preceding code, when `save()` is invoked, it saves the `Account`. Then the returned `Account` is placed in the `accountCache` cache.

As mentioned earlier, the cache is populated by the method based on the argument of the method. It is actually a default cache key. In case of the `@Cachable` annotation, the `findAccount(Long accountId)` method has `accountId` as an argument, the `accountId` is used as the cache key for this method. But in case of the `@CachePut` annotation, the only parameter of `save()` is an `Account`. It is used as the cache key. It doesn't seem fine to use `Account` as a cache key. In this case, you need the cache key to be the ID of the newly saved `Account` and not the `Account` itself. So, you need to customize the key generation behavior. Let's see how you can customize the cache key.

Customizing the cache key

You can customize the cache key by using a key attribute of `@Cacheable` and the `@CachePut` annotation. The cache key is derived by a SpEL expression using properties of the object as highlighted key attribute in the following snippet of code. Let's look at the following examples:

```
@Cacheable(cacheNames=" accountCache ", key="#accountId")
public Account findAccount(Long accountId)

@Cacheable(cacheNames=" accountCache ", key="#account.accountId")
public Account findAccount(Account account)

@CachePut(value=" accountCache ", key="#account.accountId")
Account save(Account account);
```

You can see in the preceding code snippets how we have created the cache key by using the key attribute of the `@Cacheable` annotation.

Let's see another attribute of these annotations in a Spring application.

Conditional caching

Spring's caching annotations allow you to turn off caching for some cases by using the condition attribute of `@Cacheable` and `@CachePut` annotations. These are given a SpEL expression to evaluate the conditional value. If the value of the conditional expression is true, the method is cached. If the value of the conditional expression is false, the method is not cached, but is executed every time without performing any caching operations no matter what values in the cache or what arguments are used. Let's see an example. The following method will be cached only if the passed argument has a value greater than or equal to 2000:

```
@Cacheable(cacheNames="accountCache", condition="#accountId >= 2000")
public Account findAccount(Long accountId);
```

There is one more attribute of the `@Cacheable` and `@CachePut` annotations--unless. This is also given a SpEL expression. This attribute may seem the same as the condition attribute but there is some difference between them. Unlike condition, the unless expressions are evaluated after the method has been called. It prevents the value from being placed in the cache. Let's see the following example--We only want to cache when the bank name does not contain HDFC:

```
@Cacheable(cacheNames="accountCache", condition="#accountId >= 2000", unless="#result.bankName.contains('HDFC')")
public Account findAccount(Long accountId);
```

As you can see in the preceding code snippet, we have used both attributes--condition and unless. But the unless attribute has a SpEL expression as `#result.bankName.contains('HDFC')`. In this expression, the result is a SpEL extension or cache SpEL metadata. The following is a list of the caching metadata that is available in SpEL:

Expression	Description
<code>#root.methodName</code>	The name of the cached method
<code>#root.method</code>	The cached method, that is, the method being invoked
<code>#root.target</code>	It evaluates the target object being invoked
<code>#root.targetClass</code>	It evaluates the class of the target object being invoked
<code>#root.caches</code>	An array of caches against which the current method is executed
<code>#root.args</code>	An array of the arguments passed into the cached method

#result

The return value from the cached method; only available in unless expressions for @CachePut



Spring's @CachePut and @Cacheable annotations should never be used on the same method, because they have different behaviors. The @CachePut annotation forces the execution of the cache method in order to update the caches. But the @Cacheable annotation executes the cached method only if the return value of the method is not available on the cache.

You have seen how to add information to the cache by using Spring's @CachePut and @Cacheable annotations in a Spring application. But how can we remove that information from the cache? Spring's cache abstraction provides another annotation for removing cached data from the cache--the @CacheEvict annotation. Let's see how to remove the cached data from the cache by using the @CacheEvict annotation.

The @CacheEvict annotation

Spring's cache abstraction not only allows populating caches, but also allows removing the cached data from the cache. There is a stage in the application where you have to remove stale or unused data from the cache. In that case, you can use the @CacheEvict annotation, because it doesn't add anything to the cache unlike the @Cacheable annotation. The @CacheEvict annotation is used only to perform cache eviction. Let's see how this annotation makes the `remove()` method of AccountRepository as a cache eviction:

```
@CacheEvict ("accountCache ")
void remove(Long accountId);
```

As you can see in the preceding code snippet, the value associated with the argument, `accountId`, is removed from the `accountCache` cache when the `remove()` method is invoked. The following are @Cacheable attributes:

- **value:** This is an array of names of the cache to use
- **key:** This is a SpEL expression to evaluate the cache key to be used
- **condition:** This is a SpEL expression to evaluate true or false; if it is false, then the result of caching is not being applied to the method call
- **allEntries:** This implies that if the value of this attribute is true, all entries will be removed from the caches

- **beforeInvocation:** This means that if the value of this attribute is true, the entries are removed from the cache before the method is invoked, and if the value of this attribute is false (the default), the entries are removed after a successful method invocation



We can use the `@CacheEvict` annotation on any method, even a `void` one, because it only removes the value from the cache. But in case of the `@Cacheable` and `@CachePut` annotations, we have to use a non-void return value method, because these annotations require a result to be cached.

The `@Caching` annotation

Spring's cache abstraction allows you to use multiple annotations of the same type for caching a method by using the `@Caching` annotation in a Spring application. The `@Caching` annotation groups other annotations such as `@Cacheable`, `@CachePut`, and `@CacheEvict` for the same method. For example:

```
@Caching(evict = {  
    @CacheEvict("accountCache"),  
    @CacheEvict(value="account-list", key="#account.accountId") })  
public List<Account> findAllAccount() {  
    return (List<Account>) accountRepository.findAll();  
}
```

The `@CacheConfig` annotation

Spring's cache abstraction allows you to annotate `@CacheConfig` at the class level to avoid repeated mentioning in each method. In some cases, applying customizations of the caches to all methods can be quite tedious. Here, you can use the `@CacheConfig` annotation to all operations of the class. For example:

```
@CacheConfig("accountCache")  
public class AccountServiceImpl implements AccountService {  
  
    @Cacheable  
    public Account findAccount(Long accountId) {  
        return (Account) accountRepository.findOne(accountId);  
    }  
}
```

You can see in the preceding code snippet that the `@CacheConfig` annotation is used at the class level, and it allows you to share the `accountCache` cache with all the `cacheable` methods.



Since Spring's cache abstraction module uses proxies, you should use the cache annotations only with public visibility methods. In all non-public methods, these annotations do not raise any error, but non-public methods annotated with these annotations do not show any caching behaviors.

We have already seen that Spring also offers XML namespace to configure and implement cache in a Spring application. Let's see how in the next section.

Declarative XML-based caching

To keep your configuration codes of caching separate from business codes, and to maintain loose coupling between the Spring-specific annotations and your source code, XML-based caching configuration is much more elegant than the annotation-based one. So, to configure Spring cache with XML, let's use the cache namespace along with the AOP namespace, because caching is an AOP activity, and it uses the Proxy pattern behind the declarative caching behavior.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:cache="http://www.springframework.org/schema/cache"
    xsi:schemaLocation="http://www.springframework.org/schema/cache
        http://www.springframework.org/schema/cache/spring-cache-4.3.xsd
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop-4.3.xsd">
    <!-- Enable caching -->
    <cache:annotation-driven />
    <!-- Declare a cache manager -->
    <bean id="cacheManager" class="org.springframework.cache.concurrent.ConcurrentMapCacheManager" />
</beans>
```

You can see in the preceding XML file that we have included the `cache` and `aop` namespaces. The `cache` namespace defines the caching configurations by using the following elements:

XML element	Caching Description
<code><cache:annotation-driven></code>	It is equivalent to <code>@EnableCaching</code> in Java configuration, and is used to enable the caching behavior of Spring.
<code><cache:advice></code>	It defines caching advice
<code><cache:caching></code>	It is equivalent to the <code>@Caching</code> annotation, and is used to group a set of caching rules within the caching advice
<code><cache:cacheable></code>	It is equivalent to the <code>@Cacheable</code> annotation; it makes any method cacheable
<code><cache:cache-put></code>	It is equivalent to the <code>@CachePut</code> annotation, and is used to populate a cache
<code><cache:cache-evict></code>	It is equivalent to the <code>@CacheEvict</code> annotation, and is used for cache eviction.

Let's see the following example based on XML-based configuration.

Create a configuration file, `spring.xml` as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:cache="http://www.springframework.org/schema/cache"
       xsi:schemaLocation="http://www.springframework.org/schema/cache
                           http://www.springframework.org/schema/cache/spring-cache-4.3.xsd
                           http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context.xsd
                           http://www.springframework.org/schema/aop
                           http://www.springframework.org/schema/aop/spring-aop-4.3.xsd">
    <context:component-scan base-
        package="com.packt.patterninspring.chapter9.bankapp.service,
                 com.packt.patterninspring.chapter9.bankapp.repository"/>

    <aop:config>
        <aop:advisor advice-ref="cacheAccount" pointcut="execution(*
```

```
com.packt.patterninspring.chapter9.bankapp.service.*.*(..))"/>
</aop:config>
<cache:advice id="cacheAccount">
    <cache:caching>
        <cache:cacheable cache="accountCache" method="findOne" />
        <cache:cache-put cache="accountCache" method="save"
            key="#result.id" />
        <cache:cache-evict cache="accountCache" method="remove" />
    </cache:caching>
</cache:advice>

<!-- Declare a cache manager -->
<bean id="cacheManager"
    class="org.springframework.cache.concurrent.
    ConcurrentMapCacheManager" />
</beans>
```

In the preceding XML configuration file, the highlighted code is the Spring cache configuration. In the cache configuration, the first thing that you see is the declared `<aop:config>` then `<aop:advisor>`, which have references to the advice whose ID is `cacheAccount`, and also has a pointcut expression to match the advice. The advice is declared with the `<cache:advice>` element. This element can have many `<cache:caching>` elements. But, in our example, we have only one `<cache:caching>` element, which has a `<cache:cacheable>` element, a `<cache:cache-put>`, and one `<cache:cache-evict>` element; each declare a method from the pointcut as being cacheable.

Let's see the Service class of the application with cache annotations:

```
package com.packt.patterninspring.chapter9.bankapp.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cache.annotation.CacheEvict;
import org.springframework.cache.annotation.CachePut;
import org.springframework.cache.annotation.Cacheable;
import org.springframework.stereotype.Service;

import com.packt.patterninspring.chapter9.bankapp.model.Account;
import com.packt.patterninspring.chapter9.
bankapp.repository.AccountRepository;

@Service
public class AccountServiceImpl implements AccountService{
    @Autowired
    AccountRepository accountRepository;
```

```
@Override  
@Cacheable("accountCache")  
public Account findOne(Long id) {  
    System.out.println("findOne called");  
    return accountRepository.findAccountById(id);  
}  
  
@Override  
@CachePut("accountCache")  
public Long save(Account account) {  
    return accountRepository.save(account);  
}  
  
@Override  
@CacheEvict("accountCache")  
public void remove(Long id) {  
    accountRepository.findAccountById(id);  
}
```

In the preceding file definition, we have used Spring's cache annotations to create the cache in the application. Now let's see how to configure the cache storage in an application.

Configuring the cache storage

Spring's cache abstraction provides a lot of storage integration. Spring provides CacheManager for each memory storage. You can just configure CacheManager with the application. Then the CacheManager is responsible for controlling and managing the Caches. Let's explore how to set up the CacheManager in an application.

Setting up the CacheManager

You must specify a cache manager in the application for storage, and some cache provider given to the CacheManager, or you can write your own CacheManager. Spring provides several cache managers in the org.springframework.cache package, for example, ConcurrentMapCacheManager, which creates a ConcurrentHashMap for each cache storage unit.

```
@Bean  
public CacheManager cacheManager() {  
    CacheManager cacheManager = new ConcurrentMapCacheManager();
```

```
        return cacheManager;
    }
```

`SimpleCacheManager`, `ConcurrentMapCacheManager`, and others are cache managers of the Spring Framework's cache abstraction. But Spring provides support for integration with third-party cache managers, as we will see in the following section.

Third-party cache implementations

Spring's `SimpleCacheManager` is ok for testing, but has no cache control options (overflow, eviction). So we have to use third-party alternatives like the following:

- Terracotta's EhCache
- Google's Guava and Caffeine
- Pivotal's Gemfire

Let's see one of the configurations of third-party cache managers.

Ehcach-based cache

`Ehcach` is one of the most popular cache providers. Spring allows you to integrate with Ehcache by configuring `EhCacheCacheManager` in the application. Take for example, the following Java configuration:

```
@Bean
public CacheManager cacheManager(CacheManager ehCache) {
    EhCacheCacheManager cmgr = new EhCacheCacheManager();
    cmgr.setCacheManager(ehCache);
    return cmgr;
}
@Bean
public EhCacheManagerFactoryBean ehCacheManagerFactoryBean() {
    EhCacheManagerFactoryBean eh = new EhCacheManagerFactoryBean();
    eh.setConfigLocation(new
        ClassPathResource("resources/ehcache.xml"));
    return eh;
}
```

In the preceding code, the bean method, `cacheManager()`, creates an object of `EhCacheCacheManager`, and set it with the `CacheManager` of `Ehcache`. Here, `Ehcache`'s `CacheManager` is injected into Spring's `EhCacheCacheManager`. The second bean method, `ehCacheManagerFactoryBean()`, creates and returns an instance of `EhCacheManagerFactoryBean`. Because it's a Factory bean, it will return an instance of `CacheManager`. An XML file, `ehcache.xml`, has the `Ehcache` configuration. Let's refer to the following code for `ehcache.xml`:

```
<ehcache>
    <cache name="accountCache" maxBytesLocalHeap="50m"
          timeToLiveSeconds="100">
    </cache>
</ehcache>
```

The contents of the `ehcache.xml` file vary from application to application, but you need to declare, at least, a minimal cache. For example, the following `Ehcache` configuration declares a cache named **accountCache** with 50 MB of maximum heap storage and a time-to-live of 100 seconds:

XML-based configuration

Let's create XML based configuration for the `Ehcache`, and it is configuring here `EhCacheCacheManager`. Please refer to the following code:

```
<bean id="cacheManager"
      class="org.springframework.cache.ehcache.EhCacheCacheManager"
      p:cache-manager-ref="ehcache"/>

<!-- EhCache library setup -->
<bean id="ehcache"
      class="org.springframework.cache.ehcache.
      EhCacheManagerFactoryBean" p:config-
      location="resources/ehcache.xml"/>
```

Similarly, in case of the XML configuration, you have to configure the cache manager for `ehcache`, configure the `EhCacheManagerFactoryBean` class, and set the `config-location` value with `ehcache.xml`, which has the `Ehcache` configuration as defined in the previous section.

There are many more third-party caching storages which have integration support with the Spring Framework. In this chapter, I have discussed only the ECache manager.

In the following section, we'll discuss how Spring allows you to create your own custom annotation for caching.

Creating custom caching annotations

Spring's cache abstraction allows you to create custom caching annotations for your application to recognize the cache method for the cache population or cache eviction. Spring's `@Cacheable` and `@CacheEvict` annotations are used as Meta annotations to create custom cache annotation. Let's see the following code for custom annotations in an application:

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD})
@Cacheable(value="accountCache", key="#account.id")
public interface SlowService { }
```

In the preceding code snippet, we have defined a custom annotation named as `SlowService`, which is annotated with Spring's `@Cacheable` annotation. If we use `@Cacheable` in the application, then we have to configure it as the following code:

```
@Cacheable(value="accountCache", key="#account.id")
public Account findAccount(Long accountId)
```

Let's replace the preceding configuration with our defined custom annotation, with the following code:

```
@SlowService
public Account findAccount(Long accountId)
```

As you can see, we use only the `@SlowService` annotation to make a method cacheable in the application.

Now let's move on to the next section, where we'll see which are the best practices we should consider at the time of cache implementation in an application.

Top caching best practices to be used in a web application

In your enterprise web application, proper use of caching enables the web page to be rendered very fast, minimizes the database hits, and reduces the consumption of the server's resources such as memory, network, and so on. Caching is a very powerful technique to boost your application's performance by storing stale data in the cache memory. The following are the best practices which should be considered at the time of design and development of a web application:

- In your Spring web application, Spring's cache annotations such as `@Cacheable`, `@CachePut`, and `@CacheEvict` should be used on concrete classes instead of application interfaces. However, you can annotate the interface method as well, using interface-based proxies. Remember that Java annotations are not inherited from interfaces, which means that if you are using class-based proxies by setting the attribute `proxy-target-class="true"`, then Spring cache annotations are not recognized by the proxying.
- If you have annotated any method with the `@Cacheable`, `@CachePut`, or `@CacheEvict` annotations, then never call it directly by another method of the same class if you want to benefit from the cache in the application. This is because in direct calling of a cached method, the Spring AOP proxy is never applied.
- In an enterprise application, Java Maps or any key/value collections should never be used as a Cache. Any key/value collection cannot be a Cache. Sometimes, developers use java map as a custom caching solution, but it is not a caching solution, because Cache provides more than a key/value storage, like the following:
 - Cache provides eviction policies
 - You can set the max size limit of Cache
 - Cache provides a persistent store
 - Cache provides weak reference keys

- Cache provides statistics
 - The Spring Framework provides the best declarative approach to implement and configure the Cache solution in an application. So, always use the cache abstraction layer--it provides flexibility in the application. We know that the `@Cacheable` annotation allows you to separate business logic code from the caching cross-cutting concern.
 - Be careful whenever you use cache in the application. Always use cache in a place where it is actually required such as a web service or an expensive database call, because every caching API has an overhead.
 - At the time of cache implementation in an application, you have to ensure that the data in the cache is in sync with the data storage. You can use distributed cache managers like Memcached for proper cache strategy implementation to provide considerable performance.
 - You should use cache only as second option if data fetching is very difficult from the database because of slow database queries. It is because, whenever we use caching behavior in the application, first the value is checked in the cache if not available then it execute actual method, so it would be unnecessary.
- In this chapter, we saw how caching helps to improve the performance of an application. Caching mostly works on the service layer of the application. In your application, there is a data returned by a method; we can cache that data if the application code calls it over and over again from the same requirement. Caching is a great way to avoid execution of the application method for the same requirements. The return value of the method for a specific parameter is stored in a cache whenever this method is invoked for the first time. For further calls of the same method for same parameter, the value is retrieved from that cache. Caching improves application performance by avoiding some resource and time consuming operations for same answers like performing a database query.

Summary

Spring provides Cache Manager to manage caching in a Spring application. In this chapter, you have seen how to define the caching manager for a particular caching technology. Spring provides some annotations for caching such as `@Cacheable`, `@CachePut`, and `@CacheEvict`, which we can use in our Spring application. We can also configure caching in the Spring application by using the XML configuration. Spring framework provides cache namespace to achieve this. The `<cache:cacheable>`, `<cache:cache-put>`, and `<cache:cache-evict>` elements are used instead of the corresponding annotations.

Spring makes it possible to manage caching in an application by using Aspect-Oriented Programming. Caching is a cross-cutting concern for the Spring Framework. That means, caching is as an aspect in the Spring application. Spring implements caching by using around advice of the Spring AOP module.

In the next Chapter 10, *Implementing MVC Pattern in a Web Application using Spring*, we will explore how Spring we can use in the web layer and with the MVC pattern.

14

Demystifying Microservices

Microservices is an architecture style and an approach for software development to satisfy modern business demands. Microservices are not invented, it is more of an evolution from the previous architecture styles.

We will start the chapter by taking a closer look at the evolution of microservices architecture from the traditional monolithic architectures. We will also examine the definition, concepts, and characteristics of microservices.

By the end of this chapter, you will have learned about the following:

- Evolution of microservices
- Definition of microservices architecture with examples
- Concepts and characteristics of microservices architecture

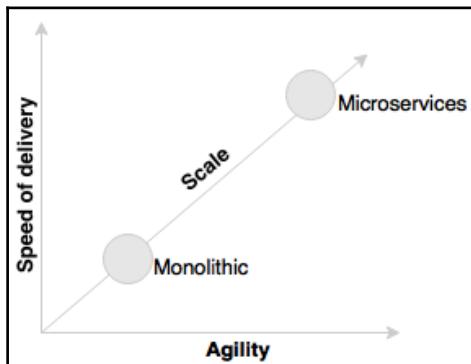
Evolution of microservices

Microservices is one of the increasingly popular architecture patterns next to **Service Oriented Architecture (SOA)**, complemented by DevOps and Cloud. Its evolution has been greatly influenced by the disruptive digital innovation trends in modern business and the evolution of technology in the last few years. We will examine these two catalysts--business demands and technology--in this section.

Business demand as a catalyst for microservices evolution

In this era of digital transformation, enterprises are increasingly adopting technologies as one of the key enablers for radically increasing their revenue and customer base. Enterprises are primarily using social media, mobile, cloud, big data, and **Internet of Things (IoT)** as vehicles for achieving the disruptive innovations. Using these technologies, enterprises are finding new ways to quickly penetrate the market, which severely pose challenges to the traditional IT delivery mechanisms.

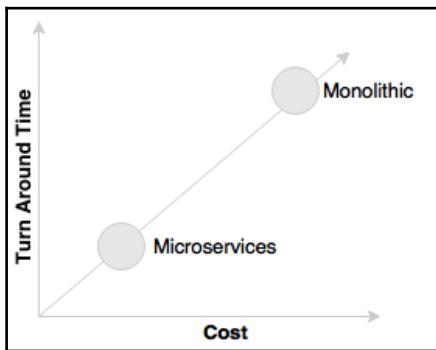
The following graph shows the state of traditional development and microservices against the new enterprise challenges, such as agility, speed of delivery, and scale:



Microservices promises more agility, speed of delivery, and scale compared to traditional monolithic applications.

Gone are the days where businesses invested in large application developments with turnaround times of a few years. Enterprises are no longer interested in developing consolidated applications for managing their end-to-end business functions as they did a few years ago.

The following graph shows the state of traditional monolithic application and microservices in comparison with the turnaround time and cost:

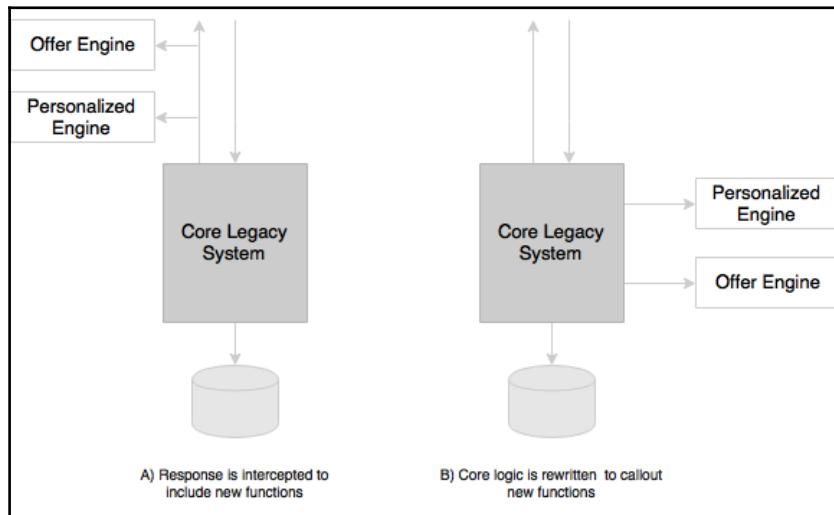


Microservices provides an approach for developing quick and agile applications, resulting in a lesser overall cost.

Today, for instance, airlines are not investing in rebuilding their core mainframe reservation systems as another monolithic monster. Financial institutions are not rebuilding their core banking systems. Retailers and other industries are not rebuilding heavyweight supply chain management applications, such as their traditional ERP's. Focus has been shifted from building large applications to building quick win, point solutions that cater to the specific needs of the business in the most agile way possible.

Let's take an example of an online retailer running with a legacy monolithic application. If the retailer wants to innovate their sales by offering their products personalized to a customer based on the customer's past shopping preferences and much more, or they want to enlighten customers by offering products to customer based on propensity to buy a product.

In such cases, enterprises want to quickly develop a personalization engine or an offer engine based on their immediate needs, and plug them into their legacy application, as shown here:



As shown in the preceding diagram, rather than investing on rebuilding the **Core Legacy System**, this will either be done by passing the responses through the new functions, as shown in the diagram marked **A**, or modifying the **Core Legacy System** to call out these functions as part of the processing, as shown in the diagram marked **B**. These functions are typically written as microservices.

This approach gives organizations a plethora of opportunities to quickly try out new functions with lesser cost in an experimental mode. Business can later validate key performance indicators change or replace these implementations if required.



Modern architectures are expected to maximize the ability to replace its parts and minimize the cost of replacing them. Microservices' approach is a means to achieve this.

Technology as a catalyst for microservices evolution

Emerging technologies have made us rethink the way we build software systems. For example, a few decades ago, we couldn't even imagine a distributed application without a two-phase commit. Later, **NoSQL** databases made us think differently.

Similarly, these kinds of paradigm shifts in technology have reshaped all layers of software architecture.

The emergence of HTML 5, CSS3, and the advancement of mobile applications repositioned user interfaces. Client-side JavaScript frameworks, such as Angular, Ember, React, Backbone, and more, are immensely popular due to their capabilities around responsive and adaptive designs.

With cloud adoptions steamed into the mainstream, **Platform as a Services (PaaS)** providers, such as Pivotal CF, AWS, Sales Force, IBM Bluemix, Redhat OpenShift, and more, made us rethink the way we build middleware components. The container revolution created by **Docker** radically influenced the infrastructure space. Container orchestration tools, such as **Mesosphere DCOS**, made infrastructure management much easier. Serverless added further easiness in application managements.

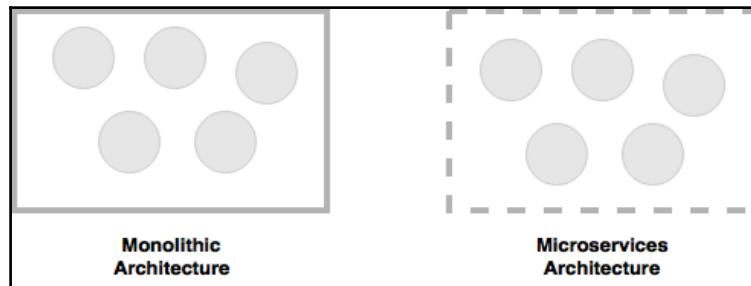
Integration landscape has also changed with the emerging **Integration Platform as a Services (iPaaS)**, such as Dell Boomi, Informatica, MuleSoft, and more. These tools helped organizations stretch integration boundaries beyond the traditional enterprise.

NoSQL and **NewSQL** have revolutionized the space of the database. A few years ago, we had only a few popular databases, all based on relational data modeling principles. Today, we have a long list of databases: **Hadoop**, **Cassandra**, **CouchDB**, **Neo 4j**, and **NuoDB**, to name a few. Each of these databases addresses certain specific architectural problems.

Imperative architecture evolution

Application architecture has always been evolving alongside with demanding business requirements and evolution of technologies.

Different architecture approaches and styles, such as mainframes, client server, *n*-tier, and service oriented were popular at different times. Irrespective of the choice of architecture styles, we always used to build one or the other forms of monolithic architectures. Microservices architecture evolved as a result of modern business demands, such as agility, speed of delivery, emerging technologies, and learning from previous generations of architectures:



Microservices help us break the boundaries of the monolithic application and build a logically independent smaller system of systems, as shown in the preceding diagram.



If we consider the monolithic application as a set of logical subsystems encompassed with a physical boundary, microservices are a set of independent subsystems with no enclosing physical boundary.

What are Microservices?

Microservices are an architectural style used by many organizations today as a game changer to achieve high degrees of agility, speed of delivery, and scale. Microservices gives us a way to develop physically separated modular applications.

Microservices are not invented. Many organizations, such as Netflix, Amazon, and eBay had successfully used the divide and conquer technique for functionally partitioning their monolithic applications into smaller atomic units, each performing a single function. These organizations solved a number of prevailing issues they were experiencing with their monolithic application. Following the success of these organizations, many other organizations started adopting this as a common pattern for refactoring their monolithic applications. Later, evangelists termed this pattern microservices architecture.

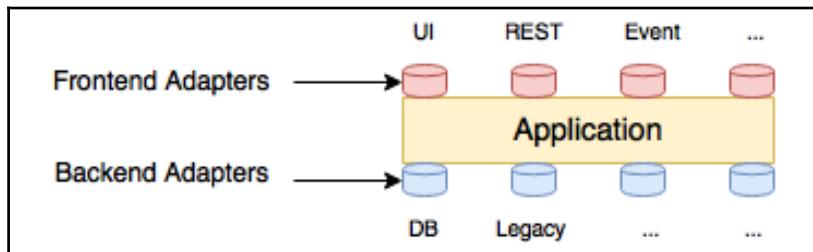
Microservices originated from the idea of **Hexagonal Architecture**, which was coined by Alister Cockburn back in 2005. Hexagonal Architecture, or Hexagonal pattern, is also known as the **Ports and Adapters** pattern.



Read more about Hexagonal Architecture here:
<http://alistair.cockburn.us/Hexagonal+architecture>

In simple terms, Hexagonal architecture advocates to encapsulate business functions from the rest of the world. These encapsulated business functions are unaware of their surroundings. For example, these business functions are not even aware of input devices or channels and message formats used by those devices. Ports and adapters at the edge of these business functions convert messages coming from different input devices and channels to a format that is known to the business function. When new devices are introduced, developers can keep adding more and more ports and adapters to support those channels without touching business functions. One may have as many ports and adapters to support their needs. Similarly, external entities are not aware of business functions behind these ports and adapters. They will always interface with these ports and adapters. By doing so, developers enjoy the flexibility to change channels and business functions without worrying too much about future proofing interface designs.

The following diagram shows the conceptual view of Hexagonal Architecture:



In the preceding diagram, the application is completely isolated and exposed through a set of frontend adapters, as well as a set of backend adapters. Frontend adaptors are generally used for integrating UI and other APIs, whereas backend adapters are used for connecting to various data sources. Ports and adapters on both sides are responsible for converting messages coming in and going out to appropriate formats expected by external entities. Hexagonal architecture was the inspiration for microservices.

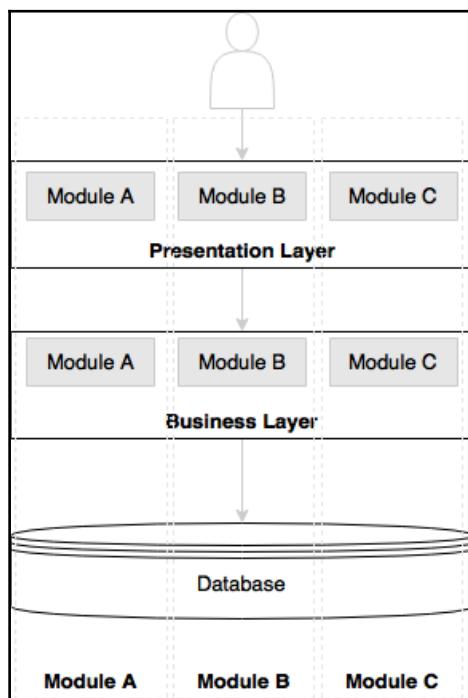
When we look for a definition for microservices, there is no single standard way of describing them. Martin Fowler defines microservices as follows:

"The microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies."--(<http://www.martinfowler.com/articles/microservices.html>)

The definition used in this book is as follows:

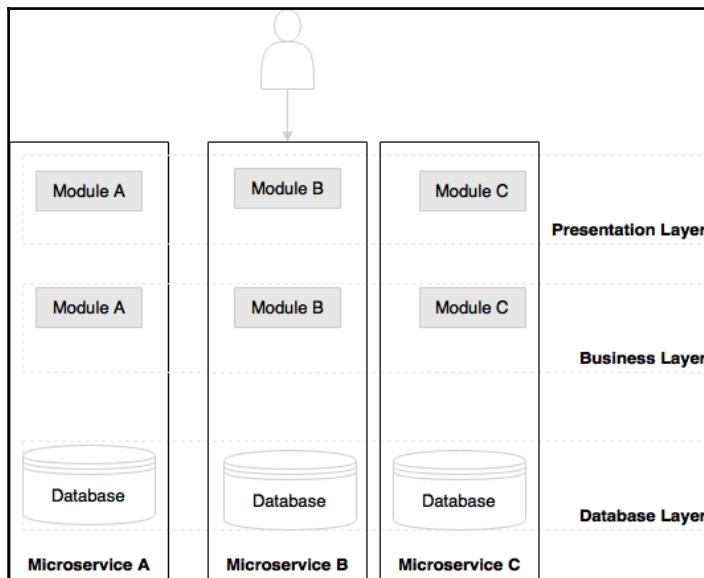


Microservices is an architectural style or an approach for building IT systems as a set of business capabilities that are autonomous, self-contained, and loosely coupled.



The preceding diagram depicts a traditional n -tier application architecture, having a **Presentation Layer**, **Business Layer**, and **Database Layer**. Modules A, B, and C represent three different business capabilities. The layers in the diagram represent separation of architecture concerns. Each layer holds all three business capabilities pertaining to that layer. The presentation layer has web components of all three modules, the business layer has business components of all three modules, and the database host tables of all three modules. In most cases, layers are physically spreadable, whereas modules within a layer are hardwired.

Let's now examine a microservices-based architecture:



As we can see in the preceding diagram, the boundaries are inverted in the microservices architecture. Each vertical slice represents a microservice. Each microservice will have its own presentation layer, business layer, and database layer. Microservices are aligned towards business capabilities. By doing so, changes to one microservice does not impact others.

There is no standard for communication or transport mechanisms for microservices. In general, microservices communicate with each other using widely adopted lightweight protocols, such as HTTP and REST, or messaging protocols, such as **JMS** or **AMQP**. In specific cases, one might choose more optimized communication protocols, such as **Thrift**, **ZeroMQ**, **Protocol Buffers**, or **Avro**.

Since microservices are more aligned to business capabilities and have independently manageable life cycles, they are the ideal choice for enterprises embarking on DevOps and cloud. DevOps and cloud are two facets of microservices.

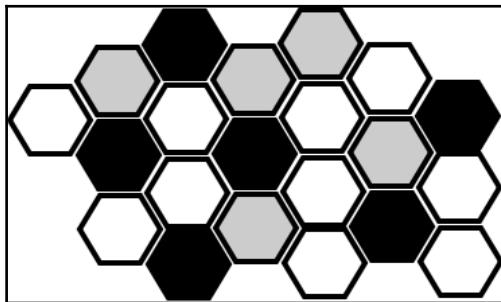


DevOps is an IT realignment to narrow the gap between traditional IT development and operations for better efficiency.

Read more about DevOps at <http://dev2ops.org/2010/02/what-is-devops/>.

Microservices - The honeycomb analogy

A honeycomb is an ideal analogy for representing the evolutionary microservices architecture:



In the real world, bees build a honeycomb by aligning hexagonal wax cells. They start small, using different materials to build the cells. Construction is based on what is available at the time of building. Repetitive cells form a pattern, and result in a strong fabric structure. Each cell in the honeycomb is independent, but also integrated with other cells. By adding new cells, the honeycomb grows organically to a big, solid structure. The content inside the cell is abstracted and is not visible outside. Damage to one cell does not damage other cells, and bees can reconstruct those cells without impacting the overall honeycomb.

Principles of microservices

In this section, we will examine some of the principles of the microservices architecture. These principles are a must have when designing and developing microservices. The two key principles are single responsibility and autonomous.

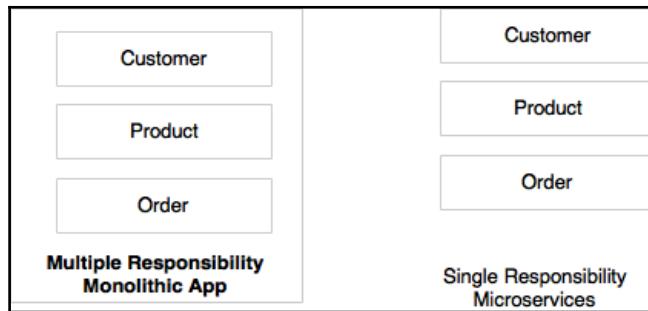
Single responsibility per service

The single responsibility principle is one of the principles defined as part of the **SOLID** design pattern. It states that a unit should only have one responsibility.



Read more about the SOLID design pattern at <http://c2.com/cgi/wiki?PrinciplesOfObjectOrientedDesign>.

It implies that a unit, either a class, a function, or a service, should have only one responsibility. At no point do two units share one responsibility, or one unit perform more than one responsibility. A unit with more than one responsibility indicates tight coupling:



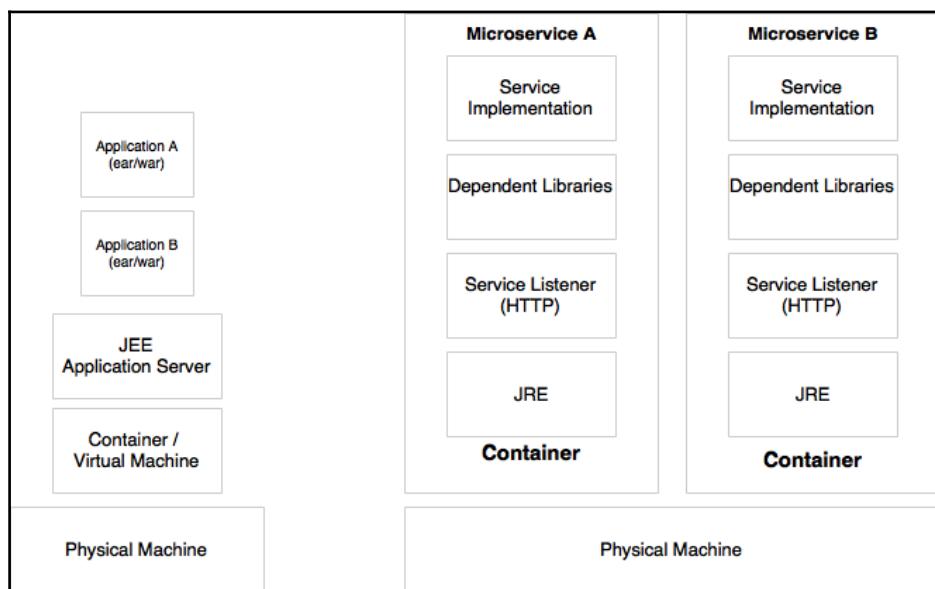
As shown in the preceding diagram, **Customer**, **Product**, and **Order** are different functions of an e-commerce application. Rather than building all of them into one application, it is better to have three different services, each responsible for exactly one business function, so that changes to one responsibility will not impair the others. In the preceding scenario, **Customer**, **Product**, and **Order** were treated as three independent microservices.

Microservices are autonomous

Microservices are self-contained, independently deployable, and autonomous services that take full responsibility of a business capability and its execution. They bundle all dependencies including the library dependencies; execution environments, such as web servers and containers; or virtual machines that abstract the physical resources.

One of the major differences between microservices and SOA is in its level of autonomy. While most of the SOA implementations provide the service-level abstraction, microservices go further and abstract the realization and the execution environment.

In traditional application developments, we build a war or a ear, then deploy it into a JEE application server, such as **JBoss**, **Weblogic**, **WebSphere**, and more. We may deploy multiple applications into the same JEE container. In the microservices approach, each microservice will be built as a fat jar embedding all dependencies and run as a standalone Java process:



Microservices may also get their own containers for execution, as shown in the preceding diagram. Containers are portable, independently manageable, and lightweight runtime environments. **Container** technologies, such as Docker, are an ideal choice for microservices deployments.

Characteristics of microservices

The microservices definition discussed earlier in this chapter is arbitrary. Evangelists and practitioners have strong, but sometimes, different opinions on microservices. There is no single, concrete, and universally accepted definition for microservices. However, all successful microservices implementations exhibit a number of common characteristics. Therefore, it is important to understand these characteristics rather than sticking to theoretical definitions. Some of the common characteristics are detailed in this section.

Services are first class citizens

In the microservices world, services are first class citizens. Microservices expose service endpoints as APIs and abstract all their realization details. The internal implementation logic, architecture, and technologies, including programming language, database, quality of services mechanisms, and more, are completely hidden behind the service API.

Moreover, in the microservices architecture, there is no more application development, instead organizations will focus on service development. In most enterprises, this requires a major cultural shift in the way applications are built.

In a customer profile microservice, the internals, such as data structure, technologies, business logic, and so on, will be hidden. It won't be exposed or visible to any external entities. Access will be restricted through the service endpoints or APIs. For instance, customer profile microservices may expose register customer and get customers as two APIs for others to interact.

Characteristics of service in a microservice

Since microservices are more or less like a flavor of SOA, many of the service characteristics defined in the SOA are applicable to microservices as well.

The following are some of the characteristics of services that are applicable to microservices as well:

- **Service contract:** Similar to SOA, microservices are described through well-defined service contracts. In the microservices world, JSON and REST are universally accepted for service communication. In case of JSON/REST, there are many techniques used to define service contracts. JSON Schema, WADL, Swagger, and RAML are a few examples.
- **Loose coupling:** Microservices are independent and loosely coupled. In most cases, microservices accept an event as input and respond with another event. Messaging, HTTP, and REST are commonly used for interaction between microservices. Message-based endpoints provide higher levels of decoupling.
- **Service abstraction:** In microservices, service abstraction is not just abstraction of service realization, but also provides complete abstraction of all libraries and environment details, as discussed earlier.
- **Service reuse:** Microservices are course grained reusable business services. These are accessed by mobile devices and desktop channels, other microservices, or even other systems.
- **Statelessness:** Well-designed microservices are a stateless, shared nothing with no shared state, or conversational state maintained by the services. In case there is a requirement to maintain state, they will be maintained in a database, perhaps in-memory.

- **Services are discoverable:** Microservices are discoverable. In a typical microservices environment, microservices self-advertise their existence and make themselves available for discovery. When services die, they automatically take themselves out from the microservices ecosystem.
- **Service interoperability:** Services are interoperable as they use standard protocols and message exchange standards. Messaging, HTTP, and more are used as the transport mechanism. REST/JSON is the most popular method to develop interoperable services in the microservices world. In cases where further optimization is required on communications, then other protocols such as Protocol Buffers, Thrift, Avro, or Zero MQ could be used. However, use of these protocols may limit the overall interoperability of the services.
- **Service Composeability:** Microservices are composeable. Service composeability is achieved either through service orchestration or service choreography.



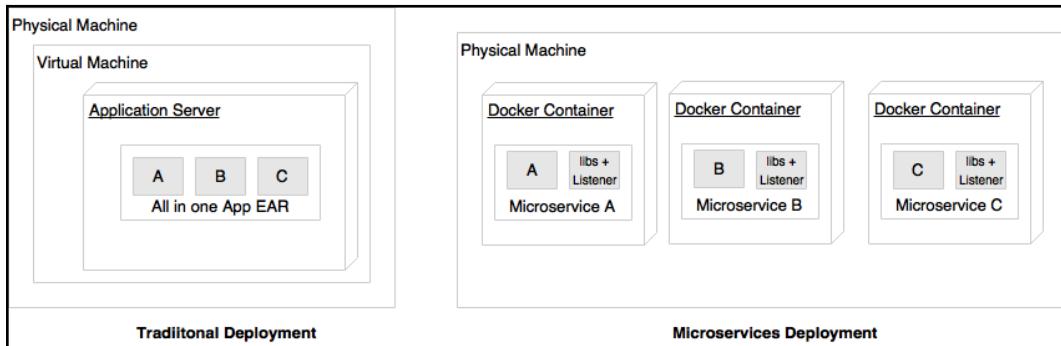
More details on SOA principles can be found at <http://serviceorientation.com/serviceorientation/index>.

Microservices are lightweight

Well-designed microservices are aligned to a single business capability; therefore, they perform only one function. As a result, one of the common characteristics we see in most of the implementations are microservices with smaller footprints.

When selecting supporting technologies, such as web containers, we will have to ensure that they are also lightweight so that the overall footprint remains manageable. For example, Jetty or Tomcat are better choices as application containers for microservices as compared to more complex traditional application servers, such as Weblogic or WebSphere.

Container technologies such as Docker also helps us keep the infrastructure footprint as minimal as possible compared to hypervisors such as VMware or **Hyper-V**.



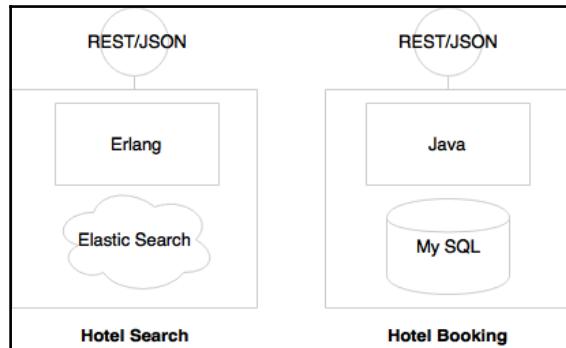
As shown in the preceding diagram, microservices are typically deployed in Docker containers, which encapsulate the business logic and needed libraries. This helps us quickly replicate the entire setup on a new machine, a completely different hosting environment, or even move across different cloud providers. Since there is no physical infrastructure dependency, containerized microservices are easily portable.

Microservices with polyglot architecture

Since microservices are autonomous and abstract everything behind the service APIs, it is possible to have different architectures for different microservices. A few common characteristics that we see in microservices implementations are as follows:

- Different services use different versions of the same technologies. One microservice may be written on Java 1.7 and another one could be on Java 1.8.
- Different languages for developing different microservices, such as one microservice in Java and another one in **Scala**.
- Different architectures such as one microservice using **Redis** cache to serve data while another microservice could use MySQL as a persistent data store.

A polyglot language scenario is depicted in the following diagram:

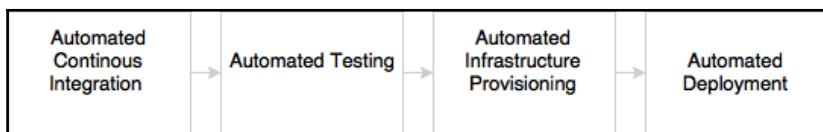


In the preceding example, since **Hotel Search** is expected to have high transaction volumes with stringent performance requirements, it is implemented using **Erlang**. In order to support predictive search, **Elastic Search** is used as the data store. At the same time, Hotel Booking needs more ACID transactional characteristics. Therefore, it is implemented using MySQL and Java. The internal implementations are hidden behind service endpoints defined as REST/JSON over HTTP.

Automation in microservices environment

Most of the microservices implementations are automated to a maximum, ranging from development to production.

Since microservices break monolithic applications into a number of smaller services, large enterprises may see a proliferation of microservices. Large numbers of microservices are hard to manage until and unless automation is in place. The smaller footprint of microservices also helps us automate the microservices development to deployment life cycle. In general, microservices are automated end to end, for example, automated builds, automated testing, automated deployment, and elastic scaling:



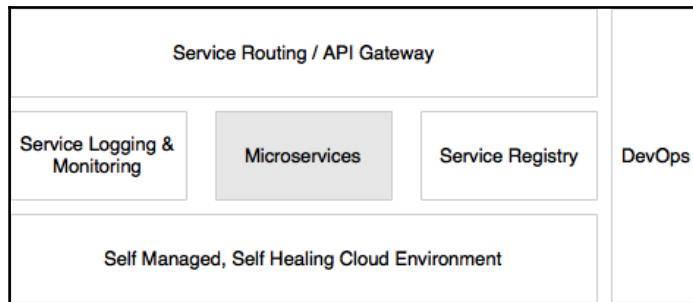
As indicated in the diagram, automations are typically applied during the development, test, release, and deployment phases.

Different blocks in the preceding diagram are explained as follows:

- The development phase will be automated using version control tools, such as Git, together with **continuous integration (CI)** tools, such as Jenkins, Travis CI, and more. This may also include code quality checks and automation of unit testing. Automation of a full build on every code check-in is also achievable with microservices.
- The testing phase will be automated using testing tools such as **Selenium**, **Cucumber**, and other **AB testing strategies**. Since microservices are aligned to business capabilities, the number of test cases to automate will be fewer compared to the monolithic applications; hence, regression testing on every build also becomes possible.
- Infrastructure provisioning will be done through container technologies, such as Docker, together with release management tools, such as Chef or Puppet, and configuration management tools, such as Ansible. Automated deployments are handled using tools such as Spring Cloud, **Kubernetes**, **Mesos**, and Marathon.

Microservices with a supporting ecosystem

Most of the large scale microservices implementations have a supporting ecosystem in place. The ecosystem capabilities include DevOps processes, centralized log management, service registry, API gateways, extensive monitoring, service routing, and flow control mechanisms:



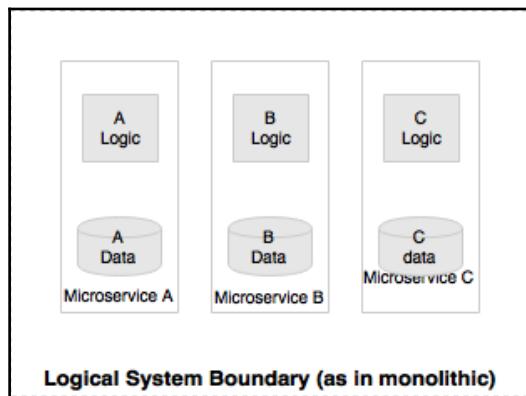
Microservices work well when supporting capabilities are in place, as represented in the preceding diagram.

Microservices are distributed and dynamic

Successful microservices implementations encapsulate logic and data within the service. This results in two unconventional situations:

- Distributed data and logic
- Decentralized governance

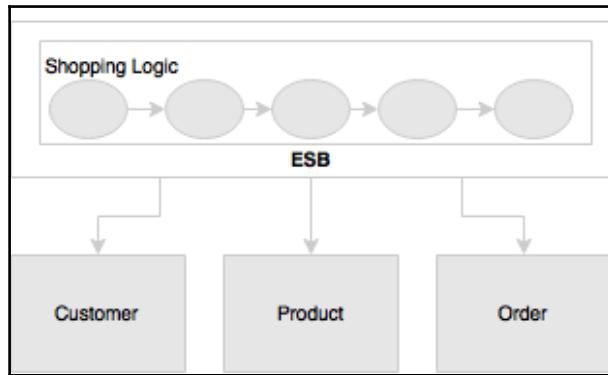
Compared to traditional applications, which consolidate all logic and data into one application boundary, microservices decentralize data and logic. Each service, aligned to a specific business capability, owns its own data and logic:



The dotted line in the preceding diagram implies the logical monolithic application boundary. When we migrate this to microservices, each microservice, **A**, **B**, and **C**, creates its own physical boundaries.

Microservices don't typically use centralized governance mechanisms the way they are used in SOA. One of the common characteristics of microservices implementations are that they are not relying on heavyweight enterprise-level products, such as an **Enterprise Service Bus (ESB)**. Instead, the business logic and intelligence are embedded as a part of the services themselves.

A retail example with ESB is shown as follows:



A typical SOA implementation is shown in the preceding diagram. **Shopping Logic** is fully implemented in the ESB by orchestrating different services exposed by **Customer**, **Order**, and **Product**. In the microservices approach, on the other hand, shopping itself will run as a separate microservice, which interacts with **Customer**, **Product**, and **Order** in a fairly decoupled way.

SOA implementations are heavily relying on static registry and repository configurations to manage services and other artifacts. Microservices bring a more dynamic nature into this. Hence, a static governance approach is seen as an overhead in maintaining up-to-date information. This is why most of the microservices implementations use automated mechanisms to build registry information dynamically from the runtime topologies.

Antifragility, fail fast, and self healing

Antifragility is a technique successfully experimented with at Netflix. It is one of the most powerful approaches to build fail-safe systems in modern software development.



The antifragility concept is introduced by Nassim Nicholas Taleb in his book, *Antifragile: Things That Gain from Disorder*.

In the antifragility practice, software systems are consistently challenged. Software systems evolve through these challenges, and, over a period of time, get better and better to withstand these challenges. Amazon's Game Day exercise and Netflix's **Simian Army** are good examples of such antifragility experiments.

Fail Fast is another concept used to build fault-tolerant, resilient systems. This philosophy advocates systems that expect failures versus building systems that never fail. Importance has to be given to how quickly the system can fail, and, if it fails, how quickly it can recover from that failure. With this approach, the focus is shifted from **Mean Time Between Failures (MTBF)** to **Mean Time To Recover (MTTR)**. A key advantage of this approach is that if something goes wrong, it kills itself, and the downstream functions won't be stressed.

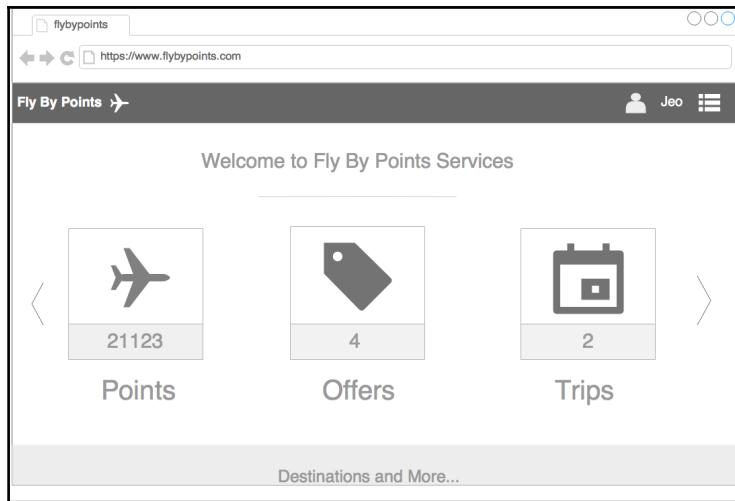
Self-Healing is commonly used in microservices deployments, where the system automatically learns from failures and adjusts itself. These systems also prevent future failures.

Microservices examples

There is no *one size fits all* approach when implementing microservices. In this section, different examples are analyzed to crystalize the microservices concept.

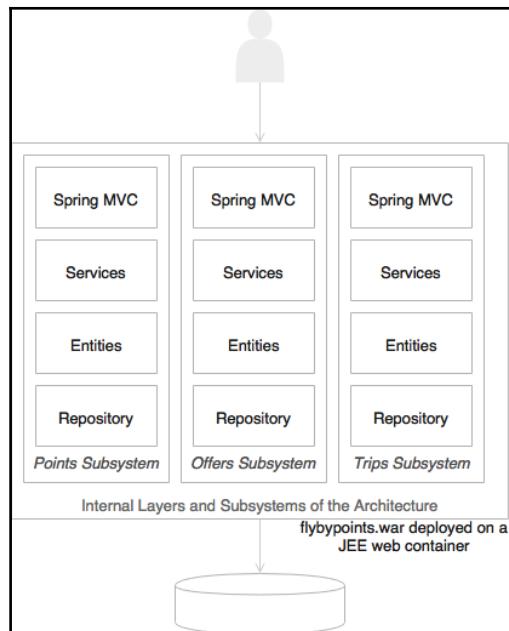
An example of a holiday portal

In the first example, we will review a holiday portal--**Fly By Points**. **Fly By Points** collects points that are accumulated when a customer books a hotel, flight, or car through their online website. When a customer logs in to the **Fly By Points** website, they will be able to see the points accumulated, personalized offers that can be availed by redeeming the points, and upcoming trips, if any:



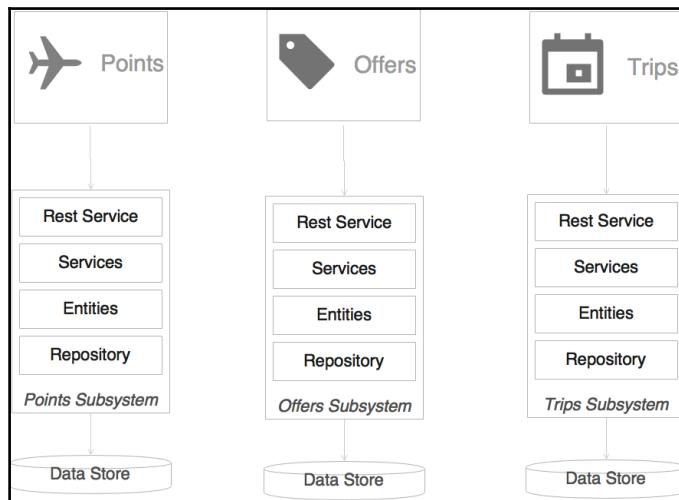
Let's assume that the preceding page is the home page after login. There are **2** upcoming trips for **Jeo**, **4** personalized offers, and **21123** points. When the user clicks on each of the boxes, details will be queried and displayed.

Holiday portal has a Java, Spring-based traditional monolithic application architecture, as follows:



As shown in the preceding diagram, Holiday portal's architecture is web-based and modular with clear separation between layers. Following the usual practice, Holiday portal is also deployed as a single war file deployed on a web server such as **Tomcat**. Data is stored in an all encompassing backing relational database. This is a good fit for the purposes of architecture when the complexities are less. As the business grows, the user base expands and the complexity also increases.

This results in a proportional increase in the transaction volumes. At this point, enterprises should look for rearchitecting the monolithic application to microservices for better speed of delivery, agility, and manageability:



Upon examining the simple microservices version of this application, we will immediately see a few things in this architecture:

- Each subsystem has now become an independent system by itself--a microservice. There are three microservices representing three business functions--**Trips**, **Offers**, and **Points**. Each one has its internal data store and middle layer. The internal structure of each service remains the same.

- Each service encapsulates its own database as well as its own HTTP listener. As opposed to the previous model, there is no web server and there is no war. Instead, each service has its own embedded HTTP listener, such as Jetty, Tomcat, and more.
- Each microservice exposes a rest service to manipulate the resources/entities that belong to that service.

It is assumed that the presentation layer is developed using a client-side JavaScript MVC framework, such as Angular JS. These client-side frameworks are capable of invoking REST calls directly.

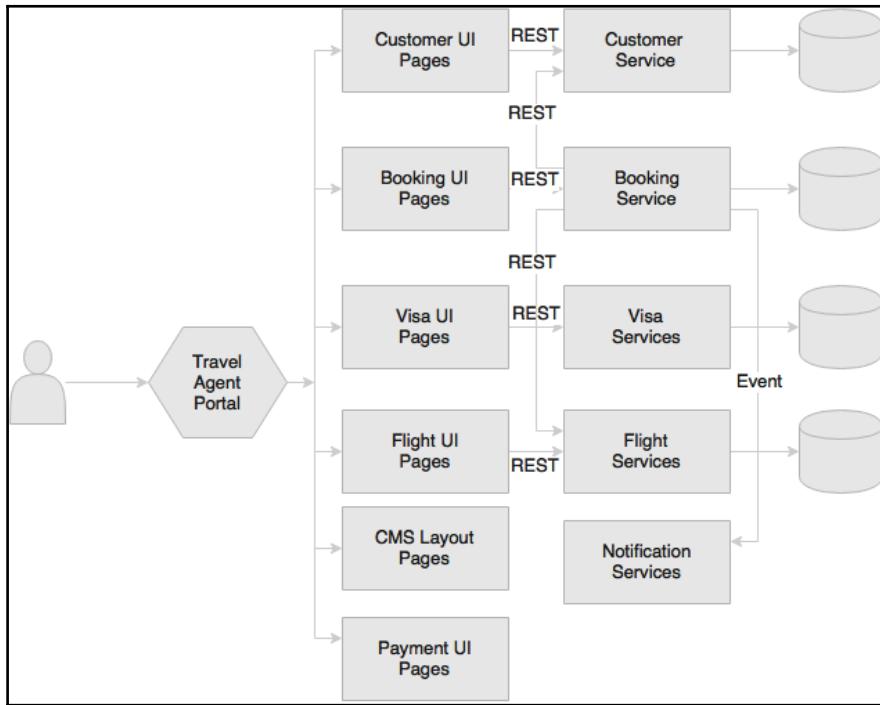
When the web page is loaded, all three boxes, **Trips**, **Offers**, and **Points**, will be displayed with details such as points, number of offers, and number of trips. This will be done by each box independently making asynchronous calls to the respective backend microservices using REST. There is no dependency between the services at the service layer. When the user clicks on any of the boxes, the screen will be transitioned and will load the details of the clicked item. This will be done by making another call to the respective microservice.

An example of a travel agent portal

This third example is a simple travel agent portal application. In this example, we will see both synchronous REST calls as well as asynchronous events.

In this case, portal is just a container application with multiple menu items or links in the portal. When specific pages are requested, for example, when the menu is clicked or a link is clicked, they will be loaded from the specific microservices:

The architecture of the **Travel Agent Portal** backed with multiple microservices is shown as follows:



When a customer requests a booking, the following events will take place internally:

- The travel agent opens the flight UI, searches for a flight, and identifies the right flight for the customer. Behind the scenes, the flight UI will be loaded from the Flight microservice. The flight UI only interacts with its own backend APIs within the Flight microservice. In this case, it makes a REST call to the Flight microservice to load the flights to be displayed.
- The travel agent then queries the customer details by accessing the customer UI. Similar to the flight UI, the customer UI will be loaded from the Customer microservices. Actions in the customer UI will invoke REST calls on the Customer microservice. In this case, the customer details will be loaded by invoking appropriate APIs on the Customer microservice.

- Then the travel agent checks the visa details to see the eligibility to travel to the selected country. This will also follow the same pattern as mentioned in the previous two points.
- Then the travel agent makes a booking using the booking UI from the Booking microservices, which again follows the same pattern.
- The payment pages will be loaded from the Payment microservice. In general, the payment service will have additional constraints, including the **Payment Card Industry Data Security Standard (PCI DSS)** compliance, such as protecting and encrypting data in motion and data at rest. The advantage of the microservices approach is that none of the other microservices need to be considered under the purview of PCI DSS as opposed to the monolithic application, where the complete application comes under the governing rules of PCI DSS. Payment also follows the same pattern described earlier.
- Once the booking is submitted, the **Booking Service** calls the flight service to validate and update the flight booking. This orchestration is defined as a part of the Booking microservices. Intelligence to make a booking is also held within the Booking microservices. As a part of the booking process, it also validates, retrieves, and updates the Customer microservice.
- Finally, the Booking microservice sends a booking event, which the **Notification Services** picks up, and sends a notification to the customer.

The interesting factor here is that we can change the user interface, logic, and data of a microservice without impacting any other microservices.

This is a clean and neat approach. A number of portal applications could be built by composing different screens from different microservices, especially for different user communities. The overall behavior and navigation will be controlled by the portal application.

The approach has a number of challenges unless the pages are designed with this approach in mind. Note that the site layouts and static contents will be loaded from the **Content Management System (CMS)** as layout templates. Alternately, this could be stored in a web server. The site layout may have fragments of User Interfaces, which will be loaded from the microservices at runtime.

Microservices benefits

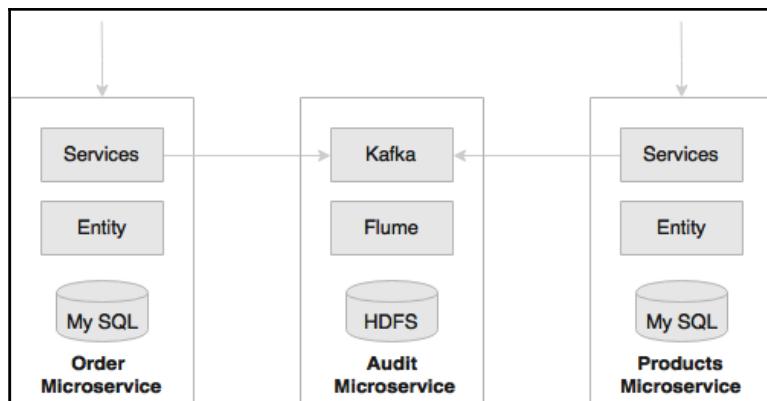
Microservice offers a number of benefits over the traditional multi-tier monolithic architectures. This section explains some of the key benefits of the microservices architecture approach.

Supports polyglot architecture

With microservices, architects and developers get flexibility in choosing the most suitable technology and architecture for a given scenario. This gives the flexibility to design better fit solutions in a more cost-effective way.

Since microservices are autonomous and independent, each service can run with its own architecture or technology, or different versions of technologies.

The following image shows a simple, practical example of polyglot architecture with microservices:



There is a requirement to audit all system transactions and record transaction details such as request and response data, users who initiated the transaction, the service invoked, and so on.

As shown in the preceding diagram, while core services like Order microservice and Product microservice use a relational data store, the Audit microservice persists data in a **Hadoop File System (HDFS)**. A relational data store is neither ideal nor cost effective to store large data volumes, like in the case of audit data. In the monolithic approach, the application generally uses a shared, single database that stores the **Order**, **Product**, and **Audit** data.

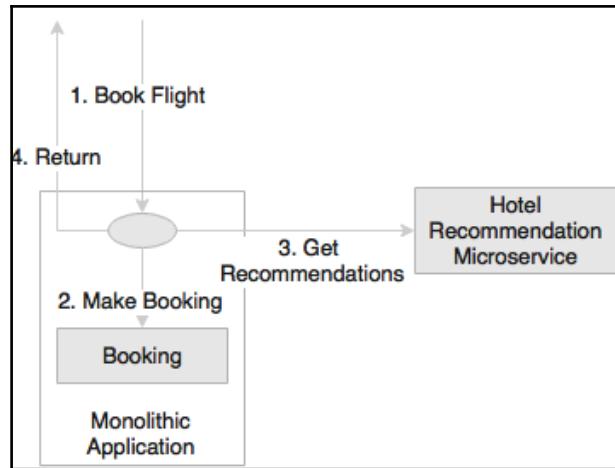
In this example, audit service is a technical microservice using a different architecture. Similarly, different functional services could also use different architectures.

In another example, there could be a Reservation microservice running on Java 7, while a Search microservice could be running on Java 8. Similarly, an Order microservice could be written on Erlang, whereas a Delivery microservice could be on the Go language. None of these are possible with a monolithic architecture.

Enables experimentation and innovation

Modern enterprises are thriving toward quick wins. Microservices is one of the key enablers for enterprises to do disruptive innovation by offering the ability to experiment and Fail Fast.

Since services are fairly simple and smaller in size, enterprises can afford to experiment with new processes, algorithms, business logic, and more. With large monolithic applications, experimentation was not easy, straightforward, or cost effective. Businesses had to spend a large sum of money to build or change an application to try out something new. With microservices, it is possible to write a small microservice to achieve the targeted functionality, and plug it into the system in a reactive style. One can then experiment with the new function for a few months. Moreover, if the new microservice is not working as expected, change or replace it with another one. The cost of change will be considerably less compared to the monolithic approach:



In another example of an airline booking website, the airline wants to show personalized hotel recommendations in their booking page. The recommendations have to be displayed on the booking confirmation page.

As shown in the preceding diagram, it is convenient to write a microservice that can be plugged into the monolithic applications booking flow rather than incorporating this requirement in the monolithic application itself. The airline may choose to start with a simple recommendation service, and keep replacing it with newer versions until it meets the required accuracy.

Elastically and selectively scalable

Since microservices are smaller units of work, it enables us to implement selective scalability and other **Quality of Services (QoS)**.

Scalability requirements may be different for different functions in an application. Monolithic applications are generally packaged as a single war or an ear. As a result, applications can only be scaled as a whole. There is no option to scale a module or a subsystem level. An I/O intensive function, when streamed with high velocity data, could easily bring down the service levels of the entire application.

In the case of microservices, each service could be independently scaled up or down. Since scalability can be selectively applied for each service, the cost of scaling is comparatively less with the microservices approach.

In practice, there are many different ways available to scale an application, and this is largely constraint to the architecture and behavior of the application. The **Scale Cube** defines primarily three approaches to scale an application:

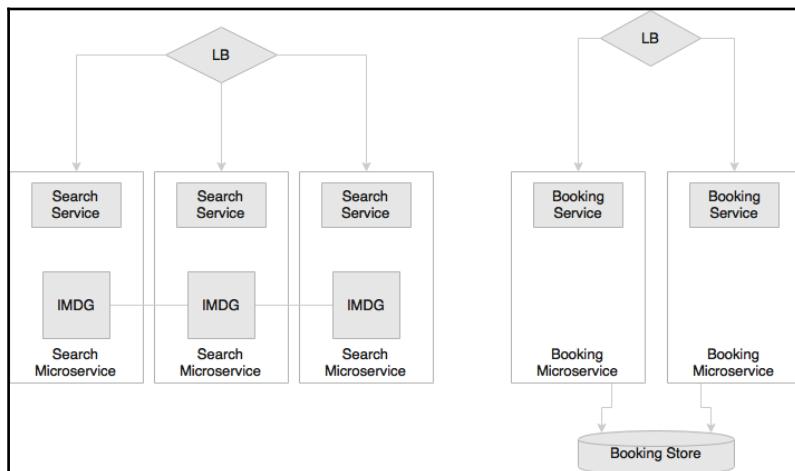
- X-axis scaling, by horizontally cloning the application
- Y-axis scaling, by splitting different functionality
- Z-axis scaling, by partitioning or sharding the data.

Read more about Scale Cube at <http://theartofscalability.com/>.



When Y-axis scaling is applied to monolithic applications, it breaks the monolithic into smaller units aligned with business functions. Many organizations successfully applied this technique to move away from monolithic application. In principle, the resulting units of functions are inline with the microservices characteristics.

For instance, on a typical airline website, statistics indicates that the ratio of flight search versus flight booking could be as high as 500:1. This means one booking transaction for every 500 search transactions. In this scenario, search needs 500 times more scalability than the booking function. This is an ideal use case for selective scaling:



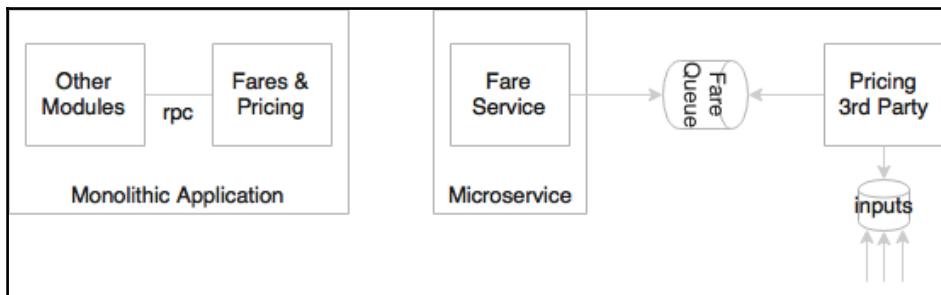
The solution is to treat search requests and booking requests differently. With a monolithic architecture, this is only possible with Z scaling in the scale cube. However, this approach is expensive, as, in Z scale, the entire codebase will be replicated.

In the preceding diagram, Search and Booking are designed as different microservices so that Search can be scaled differently from Booking. In the diagram, Search has three instances and Booking has two instances. Selective scalability is not limited to the number of instances, as shown in the preceding diagram, but also in the way in which the microservices are architected. In the case of Search, an **In-Memory Data Grid (IMDG)** such as Hazelcast can be used as the data store. This will further increase the performance and scalability of Search. When a new Search microservice instance is instantiated, an additional IMDG node will be added to the IMDG cluster. Booking does not require the same level of scalability. In the case of Booking, both instances of the Booking microservices are connected to the same instance of the database.

Allows substitution

Microservices are self-contained independent deployment modules, enabling us to substitute one microservice with another similar microservice.

Many large enterprises follow buy-versus-build policies for implementing software systems. A common scenario is to build most of the functions in-house and buy certain niche capabilities from specialists outside. This poses challenges in the traditional monolithic applications since these application components are highly cohesive. Attempting to plug in third-party solutions to the monolithic applications results in complex integrations. With microservices, this is not an afterthought. Architecturally, a microservice can be easily replaced by another microservice developed, either in-house or even extended by a microservice from a third party:



A pricing engine in the airline business is complex. Fares for different routes are calculated using complex mathematical formulas known as pricing logic. Airlines may choose to buy a pricing engine from the market instead of building the product in-house. In the monolithic architecture, **Pricing** is a function of **Fares** and **Booking**. In most cases, Pricing, Fares, and Bookings are hardwired, making it almost impossible to detach.

In a well-designed microservices system, Booking, Fares, and Pricing will be independent microservices. Replacing the Pricing microservice will have only a minimal impact on any other services, as they are all loosely coupled and independent. Today, it could be a third-party service, tomorrow, it could be easily substituted by another third-party service or another home grown service.

Enables to build organic systems

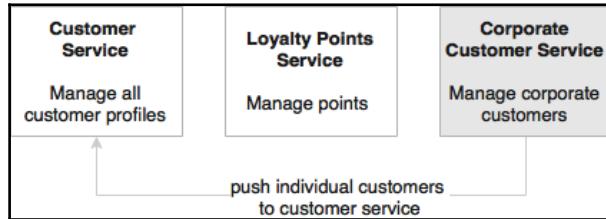
Microservices help us build systems that are organic in nature. This is significantly important when migrating monolithic systems gradually to microservices.

Organic systems are systems that grow laterally over a period of time by adding more and more functions to it. In practice, applications grow unimaginably large in its lifespan, and, in most cases, the manageability of the application reduces dramatically over that same period of time.

Microservices are all about independently manageable services. This enables us to keep adding more and more services as the need arises, with minimal impact on the existing services. Building such systems do not need huge capital investment. Hence, businesses can keep building as part of their operational expenditure.

A loyalty system in an airline was built years ago, targeting individual passengers. Everything was fine until the airline started offering loyalty benefits to their corporate customers. Corporate customers are individuals grouped under corporations. Since the current system's core data model is flat, targeting individuals, the corporate requirement needs a fundamental change in the core data model, and hence, a huge rework to incorporate this requirement.

As shown in the following diagram, in a microservices-based architecture, customer information would be managed by the Customer microservices, and loyalty by the Loyalty Points service:



In this situation, it is easy to add a new Corporate Customer microservice to manage corporate customers. When a corporation is registered, individual members will be pushed to the Customer microservices to manage them as usual. The Corporate Customer microservice provides a corporate view by aggregating data from the Customer microservice. It will also provide services to support corporate-specific business rules. With this approach, adding new services will have only a minimal impact on existing services.

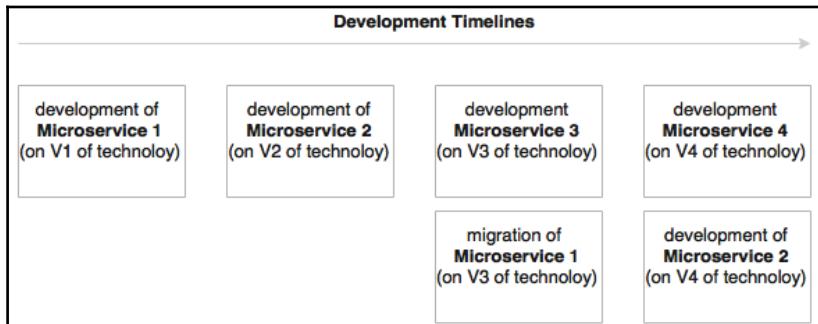
Helps managing technology debt

Since microservices are smaller in size and have minimal dependencies, they allow the migration of services that are using end-of-life technologies with minimal cost.

Technology changes are one of the barriers in software development. In many traditional monolithic applications, due to the fast changes in technology, today's next generation applications could easily become legacy, even before releasing to production. Architects and developers tend to add a lot of protection against technology changes by adding layers of abstractions. However, in reality, this approach doesn't solve the issue, but, instead, it results in over-engineered systems. Since technology upgrades are often risky and expensive, with no direct returns for the business, the business may not be happy to invest in reducing the technology debt of the applications.

With microservices, it is possible to change or upgrade technology for each service individually, rather than upgrading an entire application.

Upgrading an application with, for instance, five million lines written on **EJB 1.1** and **Hibernate** to **Spring, JPA**, and REST services is almost like rewriting the entire application. In the microservices world, this could be done incrementally.



As shown in the preceding diagram, while older versions of the services are running on old versions of technologies, new service developments can leverage the latest technologies. The cost of migrating microservices with end-of-life technologies will be considerably less compared to enhancing monolithic applications.

Allowing co-existence of different versions

Since microservices package the service runtime environment along with the service itself, it enables multiple versions of the service to coexist in the same environment.

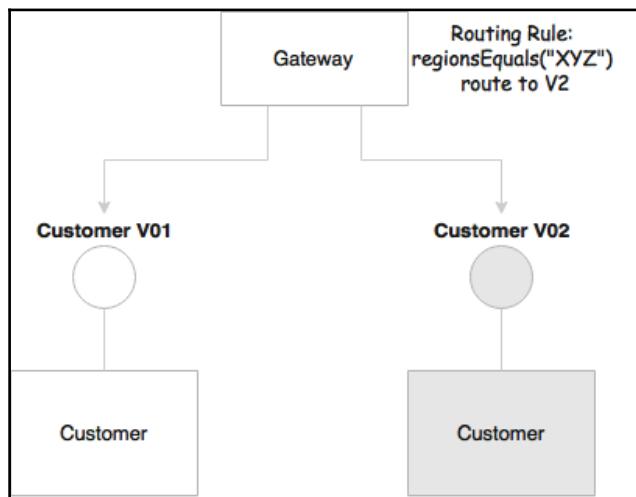
There will be situations where we will have to run multiple versions of the same service at the same time. Zero downtime promote, where one has to gracefully switch over from one version to another, is one example of such a scenario, as there will be a time window where both services will have to be up and running simultaneously.

With monolithic applications, this is a complex procedure, since upgrading new services in one node of the cluster is cumbersome as, for instance, this could lead to class loading issues. A Canary release, where a new version is only released to a few users to validate the new service, is another example where multiple versions of the services have to coexist.

With microservices, both these scenarios are easily manageable. Since each microservice uses independent environments, including the service listeners such as embedded Tomcat or Jetty, multiple versions can be released and gracefully transitioned without many issues. Consumers, when looking up services, look for specific versions of services. For example, in a canary release, a new user interface is released to user A. When user A sends a request to the microservice, it looks up the canary release version, whereas all other users will continue to look up the last production version.

Care needs to be taken at database level to ensure that the database design is always backward compatible to avoid breaking changes.

As shown in the following diagram, version V01 and V02 of the Customer service can coexist as they are not interfering with each other, given their respective deployment environment:



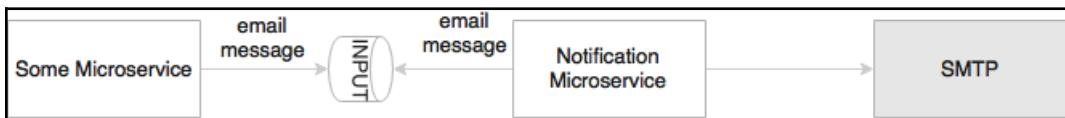
Routing rules can be set at the gateway to divert traffic to specific instances, as shown in the diagram. Alternatively, clients can request specific versions as a part of the request itself. In the diagram, the gateway selects the version based on the region from which the request originated.

Supporting building self-organizing systems

Microservices help us build self-organizing systems. A self-organizing system supporting automated deployment will be resilient and exhibits self-healing and self-learning capabilities.

In a well-architected microservice system, services are unaware of other services. It accepts a message from a selected queue and processes the message. At the end of the process, it may send out another message that triggers other services. This allows us to drop any service into the ecosystem without analyzing the impact on the overall system. Based on the input and output, the service will self-organize into the ecosystem. No additional code changes or service orchestration is required. There is no central brain to control and coordinate the processes.

Imagine an existing notification service that listens to an INPUT queue and sends notifications to a **Simple Mail Transfer Protocol (SMTP)** server as follows:



If later, a personalization engine needs to be introduced to personalize messages before sending it to the customer, the personalization engine is responsible for changing the language of the message to the customer's native language.

The updated service linkage is shown as follows:



With microservices, a new personalization service will be created to do this job. The input queue will be configured as `INPUT` in an external configuration server. The personalization service will pick up the messages from the `INPUT` queue (earlier, this was used by the notification service), and send the messages to the `OUTPUT` queue after completing the process. The notification service's input queue will then send to `OUTPUT`. From the very next moment onward, the system automatically adopts this new message flow.

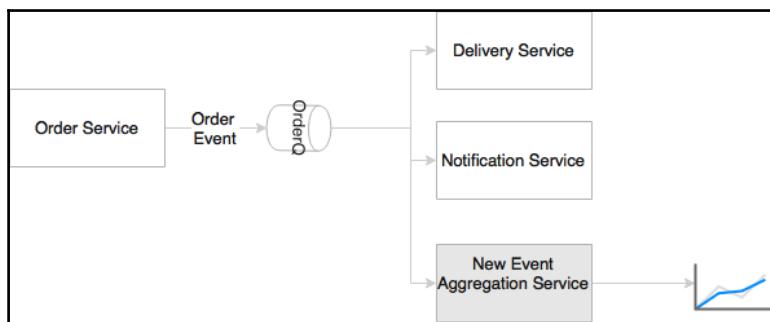
Supporting event-driven architecture

Microservices enable us to develop transparent software systems. Traditional systems communicate with each other through native protocols and hence behave like a black-box application. Business events and system events, unless published explicitly, are hard to understand and analyze. Modern applications require data for business analysis, to understand dynamic system behaviors, and analyze market trends, and they also need to respond to real-time events. Events are useful mechanisms for data extraction.

A well-architected microservice always works with events for both input and output. These events can be tapped by any services. Once extracted, events can be used for a variety of use cases.

For example, businesses want to see the velocity of orders categorized by the product type in real-time. In a monolithic system, we will need to think about how to extract these events, which may impose changes in the system.

The following diagram shows the addition of **New Event Aggregation Service** without impacting existing services:



In the microservices world, **Order Event** is already published whenever an order is created. This means that it is just a matter of adding a new service to subscribe to the same topic, extract the event, perform the requested aggregations, and push another event for the dashboard to consume.

Enables DevOps

Microservices are one of the key enablers of DevOps. DevOps is widely adopted as a practice in many enterprises, primarily to increase the speed of delivery and agility. Successful adoption of DevOps requires cultural changes and process changes, as well as architectural changes. It advocates to have agile development, high velocity release cycles, automatic testing, automatic infrastructure provisioning, and automated deployment. Automating all these processes is extremely hard to achieve with traditional monolithic applications. Microservices are not the ultimate answer, but microservices are at the center stage in many DevOps implementations. Many DevOps tools and techniques are also evolving around the use of microservices.

Considering a monolithic application takes hours to complete a full build and twenty to thirty minutes to start the application, one can see that this kind of application is not ideal for DevOps automation. It is hard to automate continuous integration on every commit. Since large monolithic applications are not automation friendly, continuous testing and deployments are also hard to achieve.

On the other hand, small footprint microservices are more automation-friendly, and, therefore, they can more easily support these requirements.

Microservices also enables smaller, focused agile teams for development. Teams will be organized based on the boundaries of microservices.

Summary

In this chapter, we learned about the fundamentals of microservices with the help of a few examples.

We explored the evolution of microservices from traditional monolithic applications. We examined some of the principles and mind shift required for modern application architectures. We also looked at some of the common characteristics repeatedly seen in most of the successful microservices implementations. Finally, we also learned the benefits of microservices.

In the next chapter, we will analyze the link between microservices and a few other architecture styles. We will also examine common microservice use cases.

15

Related Architecture Styles and Use Cases

Microservices are on top of the hype at this point. At the same time, there are noises around certain other architecture styles, for instance, serverless architecture. Which one is good? Are they competing against each other? What are the appropriate scenarios and the best ways to leverage microservices? These are the obvious questions raised by many developers.

In this chapter, we will analyze various other architecture styles and establish the similarities and relationships between microservices and other buzz words such as **Service Oriented Architecture (SOA)**, **Twelve-Factor Apps**, **serverless computing**, **Lambda architectures**, **DevOps**, **Cloud**, **Containers**, and **Reactive Microservices**.

Twelve-Factor Apps defines a set of software engineering principles to develop applications targeting cloud. We will also analyze typical use cases of microservices and review some of the popular frameworks available for the rapid development of microservices.

By the end of this chapter, you will have learned about the following:

- Relationship of microservices with SOA and Twelve-Factor Apps
- Link to serverless computing and Lambda architecture style used in the context of Big Data, **cognitive computing**, and the **Internet of Things (IoT)**
- Supporting architecture elements such as Cloud, Containers, and DevOps
- Reactive Microservices
- Typical use cases of microservices architecture
- A few popular microservices frameworks

Service-Oriented Architecture (SOA)

SOA and microservices follow similar concepts. In the Chapter 1, *Demystifying Microservices*, we saw that microservices are evolved from SOA and many service characteristics are common in both approaches.

However, are they the same or are they different?

Since microservices are evolved from SOA, many characteristics of microservices are similar to SOA. Let's first examine the definition of SOA.

The Open Group definition of SOA (<http://www.opengroup.org/soa/source-book/soa/p1.htm>) is as follows:

SOA is an architectural style that supports service-orientation. Service-orientation is a way of thinking in terms of services and service-based development and the outcomes of services.



A service:

- Is a logical representation of a repeatable business activity that has a specified outcome (e.g., check customer credit, provide weather data, consolidate drilling reports)
- Is self-contained
- May be composed of other services
- Is a "black box" to consumers of the service

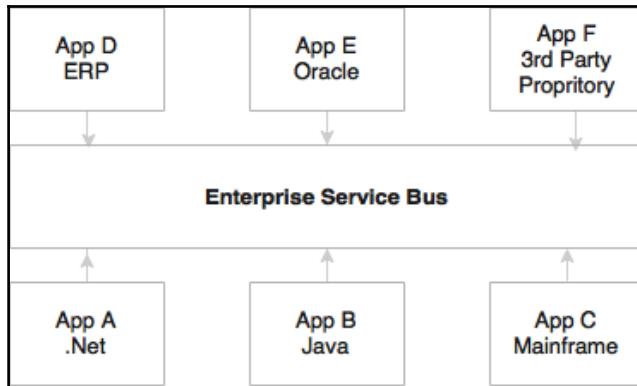
We have learned similar aspects in microservices as well. So, *in what way are microservices different?* The answer is--it depends.

The answer to the previous question could be yes or no, depending on the organization and its adoption of SOA. SOA is a broader term, and different organizations approached SOA differently to solve different organizational problems. The difference between microservices and SOA is based on the way an organization approaches SOA.

In order to get clarity, a few scenarios will be examined in the following section.

Service-oriented integration

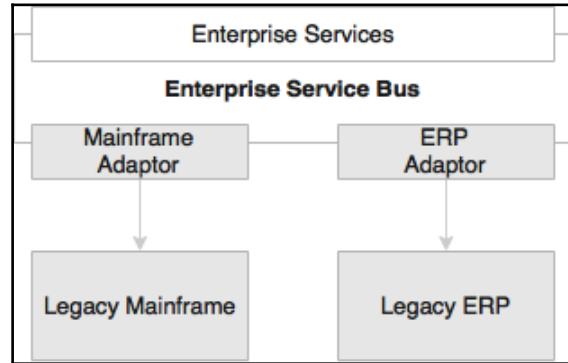
Service-oriented integration refers to a service-based integration approach used by many organizations:



Many organizations would have used SOA primarily to solve their integration complexities, also known as integration spaghetti. Generally, this is termed as **Service-Oriented Integration (SOI)**. In such cases, applications communicate with each other through a common integration layer using standard protocols and message formats, such as SOAP/XML based web services over HTTP or **Java Message Service (JMS)**. These types of organizations focus on **Enterprise Integration Patterns (EIP)** to model their integration requirements. This approach strongly relies on heavyweight **Enterprise Service Bus (ESB)**, such as TIBCO BusinessWorks, WebSphere ESB, Oracle ESB, and the likes. Most of the ESB vendors also packed a set of related products, such as Rules Engines, Business Process Management Engines, and so on, as an SOA suite. Such organization's integrations are deeply rooted into these products. They either write heavy orchestration logic in the ESB layer or business logic itself in the service bus. In both cases, all enterprise services are deployed and accessed via the ESB. These services are managed through an enterprise governance model. For such organizations, microservices is altogether different from SOA.

Legacy modernization

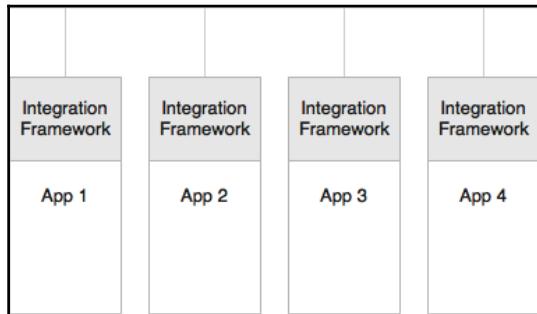
SOA is also used to build service layers on top of legacy applications:



Another category of organizations would have used SOA in transformation projects or legacy modernization projects. In such cases, the services are built and deployed in the ESB connecting to backend systems using ESB adapters. For these organizations, microservices are different from SOA.

Service-oriented application

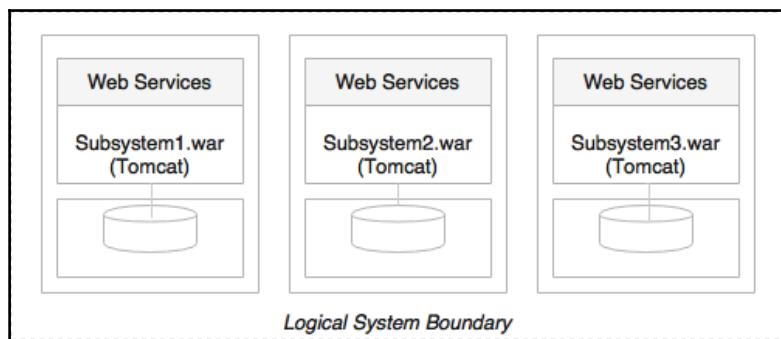
Some organizations would have adopted SOA at an application level:



In this approach, lightweight integration frameworks, such as **Apache Camel** or **Spring Integration**, are embedded within applications to handle service-related cross-cutting capabilities, such as protocol mediation, parallel execution, orchestration, and service integration. Since some of the lightweight integration frameworks had native Java object support, such applications would have even used native **Plain Old Java Objects (POJO)** services for integration and data exchange between services. As a result, all services have to be packaged as one monolithic web archive. Such organizations could see microservices as the next logical step of their SOA.

Monolithic migration using SOA

The following diagram shows a monolithic application, broken down into three micro applications:



The last possibility is transforming a monolithic application into smaller units after hitting the breaking point with the monolithic system. They would have broken the application into smaller, physically deployable subsystems, similar to the **Y-axis scaling** approach explained earlier, and deployed them as web archives on web servers, or as jars deployed on some homegrown containers. These subsystems as service would have used web services or other lightweight protocols to exchange data between services. They would have also used SOA and service-design principles to achieve this. For such organizations, they may tend to think that microservices are the same old wine in a new bottle.

Twelve-Factor Apps

Cloud computing is one of the most rapidly evolving technologies. It promises many benefits, such as cost advantage, speed, agility, flexibility, and elasticity. There are many cloud providers offering different services. They are lowering the cost models to make it more attractive to the enterprises. Different cloud providers, such as AWS, Microsoft, Rackspace, IBM, Google, and so on, use different tools, technologies, and services. On the other hand, enterprises are aware of this evolving battlefield and, therefore, they are looking for options for de-risking from lockdown to a single vendor.

Many organizations do a lift and shift of their applications to cloud. In such cases, the applications may not realize all benefits promised by the cloud platforms. Some applications need to undergo an overhaul, whereas some may need minor tweaking before moving to the cloud. This is, by and large, depends upon how the application is architected and developed.

For example, if the application has its production database server URLs hardcoded as a part of the applications war, this needs to be modified before moving the application to cloud. In the cloud, the infrastructure is transparent to the application and, especially, the physical IP addresses cannot be assumed.

How do we ensure that an application, or even microservices, can run seamlessly across multiple cloud providers and take advantages of cloud services such as elasticity?

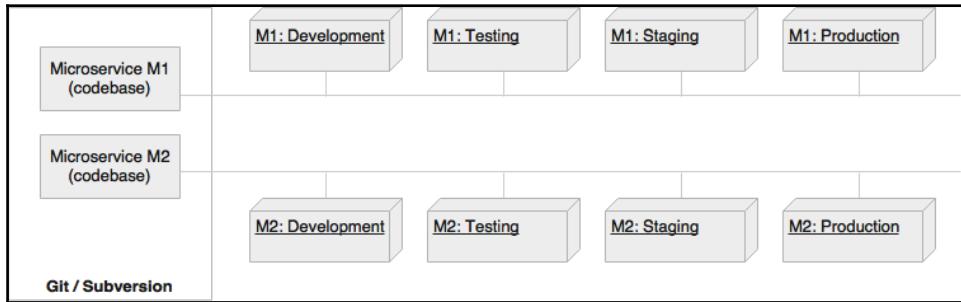
It is important to follow certain principles while developing cloud-native applications.

Cloud native is a term used to develop applications that can work efficiently in a cloud environment, and understand and utilize cloud behaviors, such as elasticity, utilization-based charging, fail aware, and so on.

Twelve-Factor App, forwarded by Heroku, is a methodology describing characteristics expected from a modern cloud-ready application. These twelve factors are equally applicable for microservices as well. Hence, it is important to understand the Twelve-Factors.

Single code base

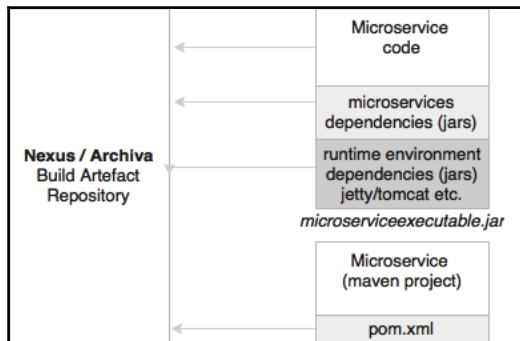
The code base principle advises that each application should have a single code base. There can be multiple instances of deployment of the same code base, such as development, testing, or production. The code is typically managed in a source-control system such as Git, Subversion, and so on:



Extending the same philosophy for microservices, each microservice should have its own code base, and this code base is not shared with any other microservice. It also means that one microservice will have exactly one code base.

Bundle dependencies

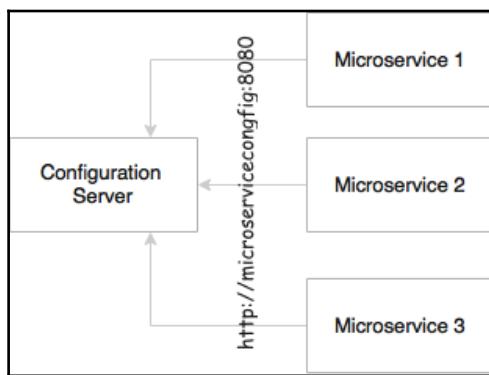
As per this principle, all applications should bundle their dependencies along with the application bundle. With build tools such as **Maven** and **Gradle**, we explicitly manage dependencies in a **Project Object Model (POM)** or gradle file, and link them using a central build artifact repository such as **Nexus** or **Archiva**. This will ensure that the versions are managed correctly. The final executables will be packaged as a war file or an executable jar file embedding all dependencies:



In the context of microservices, this is one of the fundamental principles to be followed. Each microservices should bundle all required dependencies and execution libraries, such as HTTP listener and more, in the final executable bundle.

Externalizing configurations

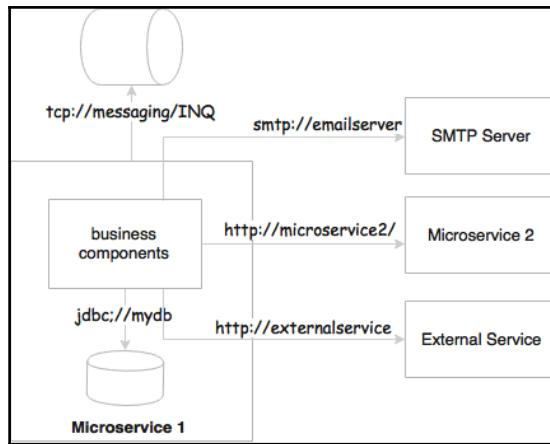
The Externalize configurations principle gives you an advice to externalize all configuration parameters from the code. An application's configuration parameters vary between environments such as support email IDs or URL of an external system, username, passwords, queue name, and more. These will be different for development, testing, and production. All service configurations should be externalized:



The same principle is obvious for microservices as well. Microservices configuration parameters should be loaded from an external source. This will also help you automate the release and deployment process as the only change between these environments are the configuration parameters.

Backing services are addressable

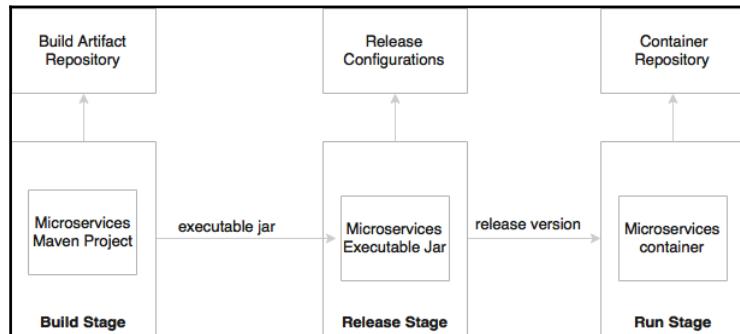
All backing services should be accessible through an addressable URL. All services need to talk to some external resources during the life cycle of their execution. For example, they could be listening to or sending messages to a messaging system, sending an email, or persisting data to a database. All these services should be reachable through a URL without complex communication requirements:



In the microservices world, microservices can either talk to a messaging system to send or receive messages, or they can accept or send messages to another service API. In a regular case, these are either HTTP endpoints using REST and JSON or TCP or HTTP-based messaging endpoints.

Isolation between build, release, and run

This principle advocates strong isolation between the build stage, the release stage, and the run stage. The build stage refers to compiling and producing binaries by including all assets required. The release stage refers to combining binaries with environment-specific configuration parameters. The run stage refers to running applications on a specific execution environment. The pipeline is unidirectional. Hence, it is not possible to propagate changes from run stages back to the build stage. Essentially, it also means that it is not recommended to do specific builds for production; rather, it has to go through the pipeline:



In microservices, the build will create executable jar files, including the service runtime, such as the HTTP listener. During the release phase, these executables will be combined with release configurations, such as production URLs, and so on, and create a release version, most probably as a container like **Docker**. In the run stage, these containers will be deployed on production via a container scheduler.

Stateless, shared nothing processes

This principle suggests that processes should be stateless and share nothing. If the application is stateless, then it is fault tolerant, and can be scaled out easily.

All microservices should be designed as stateless functions. If there is any requirement to store a state, it should be done with a backing database or in an in-memory cache.

Expose services through port bindings

A Twelve-Factor App is expected to be self-contained or standalone. Traditionally, applications are deployed into a server--a web server or an application server, such as Apache Tomcat or JBoss, respectively. A Twelve-Factor App ideally does not rely on an external web server. A HTTP listener, such as Tomcat, **Jetty**, and more, has to be embedded in the service or application itself.

Port binding is one of the fundamental requirements for microservices to be autonomous and self-contained. Microservice embeds the service listeners as a part of the service itself.

Concurrency for scale out

The concurrency for scale out principle states that processes should be designed to scale out by replicating the processes. This is in addition to the use of threads within the process.

In the microservices world, services are designed for a scale out rather than scale up. The X-axis scaling technique is primarily used for scaling a service by spinning up another identical service instance. The services can be elastically scaled or shrunk, based on the traffic flow. Furthermore, microservices may make use of parallel processing and concurrency frameworks to further speed up or scale up the transaction processing.

Disposability, with minimal overhead

The disposability with minimal overhead principle advocates to build applications with minimal startup and shutdown times, and with a graceful shutdown support. In an automated deployment environment, we should be able to bring up or bring down instances as quickly as possible. If the application's startup or shutdown takes considerable time, it will have an adverse effect on automation. The startup time is proportionally related to the size of the application. In a cloud environment targeting auto scaling, we should be able to spin up a new instance quickly. This is also applicable when promoting new versions of services.

In the microservices context, in order to achieve full automation, it is extremely important to keep the size of the application as thin as possible, with minimal startup and shutdown times. Microservices should also consider lazy loading of objects and data.

Development, production parity

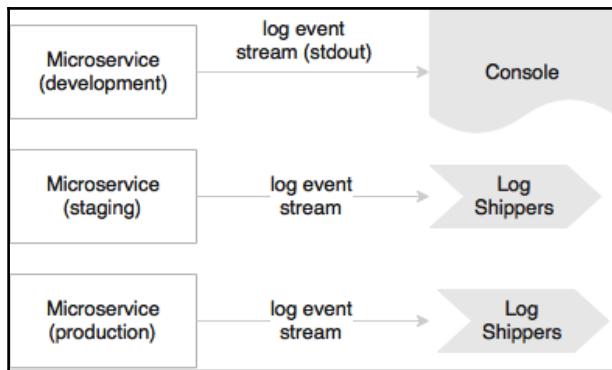
The development, production parity principle states the importance of keeping the development and production environments as identical as possible. For example, let's consider an application with multiple services or processes, such as a job scheduler service, cache services, or one or more application services. In a development environment, we tend to run all of them on a single machine. Whereas, in production, we will facilitate independent machines to run each of these processes. This is primarily to manage cost of the infrastructure. The downside is that, if production fails, there is no identical environment to reproduce and fix the issues.

This principle is not only valid for microservices, but it is also applicable to any application development.

Externalizing logs

A Twelve-Factor App never attempts to store or ship log files. In a cloud, it is better to avoid local I/Os or file systems. If the I/Os are not fast enough in a given infrastructure, they could create a bottleneck. The solution to this is to use a centralized logging framework. **Splunk**, **greylog**, **Logstash**, **Logplex**, **Loggly** are some examples of log shipping and analysis tools. The recommended approach is to ship logs to a central repository by tapping the logback appenders and write to one of the shipper's endpoints.

In a microservices ecosystem, this is very important, as we are breaking a system into a number of smaller services, which could result in decentralized logging. If they store logs in a local storage, it would be extremely difficult to correlate logs between services:



In development, microservice may direct the log stream to **stdout**, whereas, in production, these streams will be captured by the log shippers and sent to a central log service for storage and analysis.

Package admin processes

Apart from application requests, most of the applications provision for admin tasks. This principle advises you to target the same release and an identical environment as the long running processes runs to perform these activities. Admin code should also be packaged along with the application code.

This principle is not only valid for microservices, but it is also applicable to any application development.

Serverless computing

Serverless computing architecture or Functions as a Service (FaaS) has gained quite a bit of popularity these days. In serverless computing, developers need not worry about application servers, virtual machines, Containers, infrastructure, scalability, and other quality of services. Instead, developers write functions and drop those functions into an already running computing infrastructure. Serverless computing improves faster software deliveries as it eliminates the provisioning and management

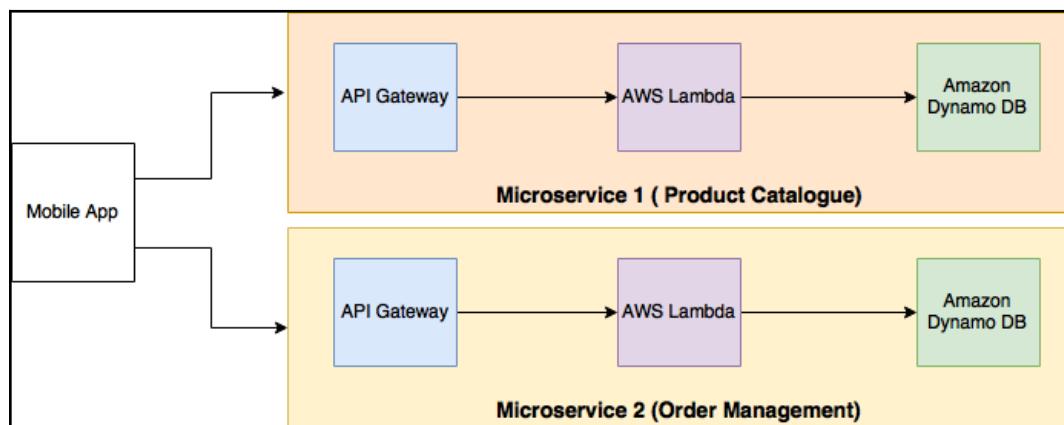
part of the infrastructure required by microservices. Sometimes, this is even referred to as **NoOps**.

FaaS platforms support multiple language runtimes, such as Java, Python, Go, and so on. There are many serverless computing platforms and frameworks available. Moreover, this space is still evolving. **AWS Lambda**, **IBM OpenWhisk**, **Azure Functions**, **Google Cloud Functions** are some of the popular managed infrastructures for serverless computing. **Red Hat Funktion** is another serverless computing platform on top of **Kubernetes**, which can be deployed on any cloud or on premise. **IronFunctions** is one of the recent entrants--a cloud-agnostic, serverless computing platform. There are other serverless computing platforms, such as Webtask for web-related functions. **BrightWork** is another serverless computing platform for JavaScript applications that offers minimal vendor locking.

There are many other frameworks intended to simplify AWS Lambda development and deployment supporting different languages. **Apex**, **Serverless**, **Lambda Framework for Java**, **Chalice for Python**, **Claudia for Node JS**, **Sparta for Go**, and **Gordon** are some of the popular frameworks in this category.

Serverless computing is closely related to microservices. In other words, microservices are the basis for serverless computing. There are a number of characteristics shared by both, serverless computing as well as microservices. Similar to microservices, functions generally perform one task at a time, are isolated in nature, and communicate through designated APIs that are either event-based or HTTP. Also, functions have smaller footprints like microservices. It is safe to say that functions follow microservices-based architecture.

The following diagram shows a serverless computing scenario based on AWS Lambda:



In this scenario, each microservice will be developed as a separate AWS Lambda function and independently wired through an API gateway for HTTP-based communication. In this case, each microservice holds its own data in an Amazon DynamoDB.

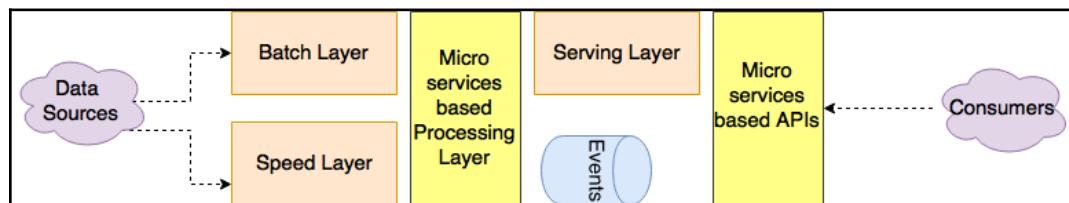
Generally, the FaaS billing model is truly based on the **pay as you use** model, as opposed to pay for what is reserved model followed in case of virtual machines or EC2 instances. Moreover, developers need not worry about passivating images when they are no longer used. When there are only a few transactions, just enough computing power will be charged. As the load increases, more resources will be automatically allocated. This makes serverless computing an attractive model for many enterprises.

The new style of microservices use cases such as big data, cognitive computing, IoT, and bots, which are perfect candidates for serverless computing. More about these use cases is explained in the next section.

It is also important to note that the disadvantage of serverless computing is its strong vendor locking. This space is still getting matured, and, perhaps, we will see more tooling in this space to reduce these gaps. In the future, we will also see a large number of services available in the market place that can be utilized by microservice developers when developing on serverless computing platforms. Serverless computing, together with microservices architecture, is definitely a promising choice for developers.

Lambda architecture

There are new styles of microservices use cases in the context of big data, cognitive computing, bots, and IoT:



The preceding diagram shows a simplified **Lambda architecture** commonly used in the context of big data, cognitive, and IoTs. As you can see in the diagram, microservices play a critical role in the architecture. The batch layer process data, and store typically in a **Hadoop Distributed File System (HDFS)** file system. Microservices are written on top of this batch layer process data and build serving layer. Since microservices are independent, when they encounter new demands, it is easy to add those implementations as microservices.

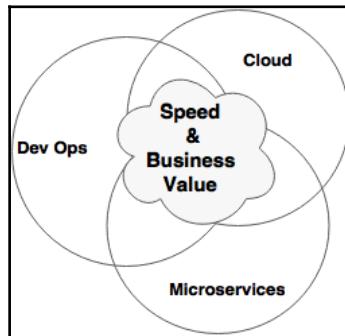
Speed-layer microservices are primarily reactive microservices for stream processing. These microservices accept a stream of data, apply logic, and then respond with another set of events. Similarly, microservices are also used for exposing data services on top of the serving layer.

The following are different variations of the preceding architecture:

- **Cognitive computing** scenarios, such as integrating an optimization service, forecasting service, intelligent price calculation service, prediction service, offer service, recommendation service, and more, are good candidates for microservices. These are independent stateless computing units that accepts certain data, applies algorithms, and returns the results. These are cognitive computing microservices run on top of either speed layer or batch layer. Platforms such as Algorithmia uses microservices-based architecture.
- **Big Data** processing services that run on top of big data platforms to provide answer sets is another popular use case. These services connect to the big data platform's read-relevant data, process those records, and provide necessary answers. These services typically run on top of the batch layer. Platforms such as **MapR** embrace microservices.
- **Bots** that are conversational in nature use the microservices architecture. Each service is independent and executes one function. This can be treated as either API service on top of the serving layer or stream processing services on top of the speed layer. Bots platforms, such as the Azure bot service, leverages the microservices architecture.
- **IoT** scenarios such as machine or sensor data stream processing utilize microservices to process data. These kinds of services run on top of the speed layer. Industrial internet platforms such as Predix are based on the microservices philosophy.

DevOps, Cloud, and Containers

The trios Cloud (more specifically, Containers), Microservices and **DevOps**, are targeting a set of common objectives--speed of delivery, business value, and cost benefits. All three can stay evolved independently, but they complement each other to achieve the desired common goals. Organizations embarking on any of these naturally tend to consider the others as they are closely linked together:



Many organizations start their journey with DevOps as an organizational practice to achieve high velocity release cycles, but eventually move to microservices architecture and cloud. However, it is not mandatory to have microservices and Cloud to support DevOps. However, automating release cycles of large monolithic applications does not make much sense, and, in many cases, it would be impossible to achieve. In such scenarios, microservices architecture and Cloud will be handy when implementing DevOps.

If we flip the coin, Cloud does not need a microservices architecture to achieve its benefits. However, to effectively implement microservices, both Cloud and DevOps are essential.

In summary, if the objective of an organization is to achieve speed of delivery and quality in a cost-effective way, the trio together can bring tremendous success.

DevOps as the practice and process for microservices

Microservice is an architecture style that enables quick delivery. However, microservices cannot provide the desired benefits by itself. A microservices-based project with a delivery cycle of six months does not give the targeted speed of delivery or business agility. Microservices need a set of supporting delivery practices and processes to effectively achieve their goal.

DevOps is the ideal candidate as the underpinning process and practices for microservice delivery. It processes and practices gel well with the microservices architecture philosophies.

Cloud and Containers as the self-service infrastructure for microservices

The main driver for cloud is to improve agility and reduce cost. By reducing the time to provision the infrastructure, the speed of delivery can be increased. By optimally utilizing the infrastructure, one can bring down the cost. Therefore, cloud directly helps you achieve both speed of delivery and cost.

Without having a cloud infrastructure with the cluster management software, it would be hard to control the infrastructure cost when deploying microservices. Hence, the cloud with self-service capabilities is essential for microservices to achieve their full potential benefits. In a microservices context, the cloud not only helps you abstract the physical infrastructure, but also provides software APIs for dynamic provisioning and automatic deployments. This is referred to as **infrastructure as code** or **software defined infrastructure**.

Containers further provide benefits when dealing with DevOps and microservices. They provide better manageability and a cost-effective way of handling large volumes of deployments. Furthermore, container services and container orchestration tools helped you manage infrastructures.

Reactive microservices

The reactive programming paradigm is an effective way to build scalable, fault-tolerant applications. The reactive manifesto defines basic philosophy of reactive programming.



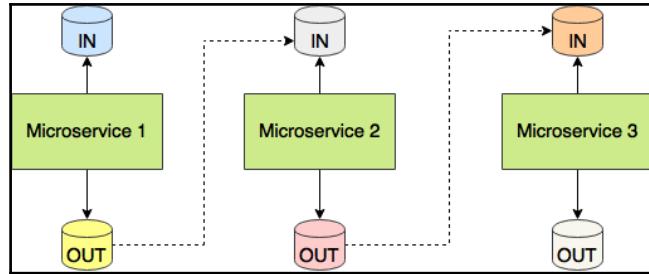
Read more about the reactive manifesto here:
<http://www.reactivemanifesto.org>

By combining the reactive programming principles together with the microservices architecture, developers can build low latency high throughput scalable applications.

Microservices are typically designed around business capabilities. Ideally, well-designed microservices will exhibit minimal dependencies between microservices. However, in reality, when microservices are expected to deliver same capabilities delivered by monolithic applications, many microservices have to work in collaboration. Dividing services based on business capabilities will not solve all issues. Isolation and communication between services are also equally important. Even though microservices are often designed around business capabilities, wiring them with synchronous calls can form hard dependencies between them. As a result, organizations may not realize the full benefits of microservices. Distributed systems with strong dependencies between them have its own overheads and are hard to manage. For example, if an upgrade is required for one of the microservices, it can seriously impact other dependent services. Hence, adopting reactive style is important for successful microservices implementations.

Let's examine reactive microservices a bit more. There are four pillars when dealing with reactive programming. These attributes are **resilient, responsive, message based, and elastic**. Resilient and responsive are closely linked to isolation. Isolation is the foundation for both reactive programming as well as microservices. Each microservice is autonomous and is the building block of a larger system. By and large, these microservices are isolated from the rest of its functional components by drawing boundaries around business functions. By doing so, failures of one service can be well isolated from the others. In case of failures, it should not cause issues to its downstream services. Either a fallback service or a replica of the same service will takeover its responsibility temporarily. By introducing isolation, each isolated components can be scaled, managed, and monitored independently.

Even though isolation is in place, if the communication or dependencies between services are modeled as synchronous blocking RPC calls, then failures cannot be fully isolated. Hence, it is extremely important to design communications between services in a reactive style by using asynchronous non-blocking calls:



As shown in the preceding diagram, in a reactive system, each microservice will listen to an event. The service reacts upon receiving an input event. The service then starts processing the event and sends out another response event. The microservice itself is not aware of any other services running in the ecosystem. In this example, **Microservice 1** does not have any knowledge of **Microservice 2** and **Microservice 3**. Choreography can be achieved by connecting an output queue of one service to the input queue of another service, as shown in the diagram.

Based on the velocity of events, services can automatically scale up by spinning up replicas of instances. For example, in a reactive style order management system, as soon as an order is placed, the system will send out an **Order Event**. There could be many microservices listening to an **Order Event**. Upon consuming this event, they perform various things. This design also allows developers to keep adding more and more reactive routines as need arises.

The flow control or choreography in case of reactive microservices will be taken care of automatically, as shown earlier in the diagram. There is no central command and control. Instead, messages and input and output contracts of microservices itself will establish flow. Changing the message flow is easy to achieve by rewiring input and output queues.

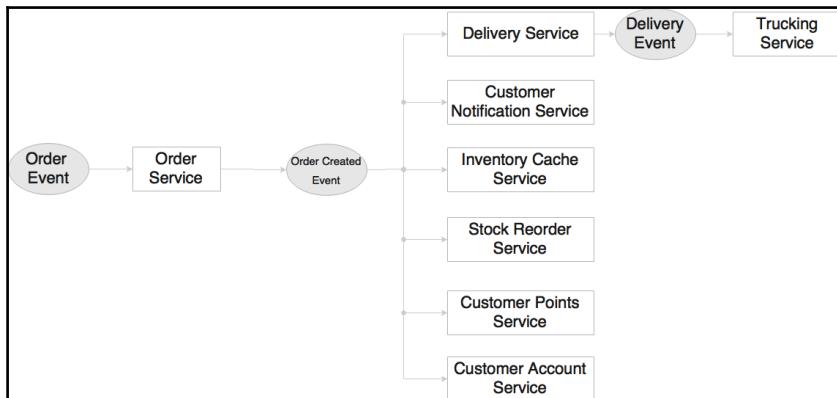


The **Promise theory** proposed by Mark Burgess in 2004 has so much relevance in this situation. The Promise theory defines an autonomous collaboration model for systems or entities to interact in a distributed environment using voluntary cooperation. The theory claims that promise-based agents can reproduce the same behavior exhibited by traditional command and control systems that follows the obligation model. Reactive microservices are inline with the Promise theory, in which, each service is independent and can collaborate in a completely autonomous manner. **Swarm Intelligence** is one of these formal architecture mechanisms, increasingly applied in the modern artificial intelligent systems for building highly scalable intelligence routines.

A highly scalable and reliable message system is the single most important component in a reactive microservices ecosystem. **QBit**, **Spring Reactive**, **RxJava**, and **RxJS** are some of the frameworks and libraries that help you build reactive microservices. Spring 5 has inbuilt support to develop reactive web applications. **Spring Cloud Streams** are good choice to build truly reactive microservices using Spring Framework.

A reactive microservice-based order management system

Let's examine another microservices example: an online retail web site. In this case, we will focus more on the backend services, such as the order event generated when the customer places an order through the website:



This microservices system is completely designed based on reactive programming practices.

When an event is published, a number of microservices are ready to kick start upon receiving the event. Each one of them is independent and is not relying on other microservices. The advantage of this model is that we can keep adding or replacing microservices to achieve specific needs.

In the preceding diagram, there are eight microservices shown. The following activities take place on arrival of an **OrderEvent**:

- **OrderService** kicks off when an **OrderEvent** is received. **OrderService** creates an order and saves details to its own database.
- If the order is successfully saved, an **OrderSuccessfulEvent** is created by **OrderService** and published.
- A series of actions will take place when **OrderSuccessfulEvent** arrives.
- **DeliveryService** accepts the event and places a **DeliveryRecord** to deliver the order to the customer. This in turn generates **DeliveryEvent** and publishes the event.
- The **TruckingService** picks up the **DeliveryEvent** and processes it. For instance, **TruckingService** creates a trucking plan.
- **CustomerNotificationService** sends a notification to the customer informing the customer that an order is placed.
- **InventoryCacheService** updates the inventory cache with the available product count.
- **StockReorderService** checks whether the stock limits are adequate and generates **ReplenishEvent** if required.
- **CustomerPointsService** recalculates the customer's loyalty points based on this purchase.
- **CustomerAccountService** updates the order history in the customer account.

In this approach, each service is responsible for only one function. Services accept and generate events. Each service is independent and is not aware of its neighborhoods. Hence, the neighborhood can organically grow as mentioned in the **honeycomb analogy**. New services could be added as and when necessary. Adding a new service does not impact any of the existing service.

Microservice use cases

Microservice is not a silver bullet, and it will not solve all the architectural challenges of today's world. There is no hard and fast rule, or a rigid guideline on when to use microservices.

Microservices may not fit in each and every use case. The success of microservices largely depends on the selection of use cases. The first and the foremost activity is to do a litmus test of the use case against the microservices benefits. The litmus test must cover all microservices benefits we had discussed earlier in this chapter. For a given use case, if there are no quantifiable benefits, or if the cost is outweighing the benefits, then the use case may not be the right choice for microservices.

Let's discuss some commonly used scenarios that are suitable candidates for a microservice architecture:

- Migrating a monolithic application due to improvements required in scalability, manageability, agility, or speed of delivery. Another similar scenario is rewriting an end-of-life heavily-used legacy application. In both cases, microservices presents an opportunity. Using a microservices architecture, it is possible to replatform the legacy application by slowly transforming functions to microservices. There are benefits in this approach. There is no humongous upfront investment required, no major disruption to business, and there are no severe business risks. Since the service dependencies are known, the microservices dependencies can be well managed.
- In many cases, we build headless business applications or business services that are autonomous in nature. For instance, payment service, login service, flight search service, customer profile service, notification service, and more. These are normally reused across multiple channels, and, hence, are good candidates for building them as microservices.
- There could be micro or macro applications that serve a single purpose and performs a single responsibility. A simple time-tracking application is an example of this category. All it does is capture time, duration, and the task performed. Common use enterprise applications are also candidates for microservices.

- Backend services of a well architected, responsive, client side MVC web application (**Backend as a Service (BaaS)**). In most of these scenarios, data could be coming from multiple logically different data sources, as described in the *Fly By Points* example mentioned in Chapter 1, *Demystifying Microservices*.
- Highly agile applications, applications demanding speed of delivery or time to market, innovation pilots, applications selected for DevOps, System of Innovation type of applications, and so on, could also be considered as potential candidates for microservices architecture.
- Applications that we could anticipate getting benefits from microservices, such as polyglot requirements; applications that require **Command Query Responsibility Segregation (CQRS)**; and so on, are also potential candidates of microservices architecture.
- Independent technical services and utility services, such as communication service, encryption service, authentication services and so on, are also good candidates for microservices.

If the use case falls into any of these categories, they are potential candidates for microservices architecture. There are a few scenarios that we should consider avoiding in microservices:

- If the organization policies are forced to use centrally-managed heavyweight components, such as ESBs, to host the business logic, or if the organization has any other policies that are hindering the fundamental principles of microservices, then microservices are not the right solution, unless the organizational process is relaxed.
- If the organization's culture, processes, and more are based on traditional waterfall delivery model, lengthy release cycles, matrix teams, manual deployments and cumbersome release processes, no infrastructure provisioning, and more, then microservices may not be the right fit. This is underpinned by the **Conway's law**. The Conway's law states that there is a strong link between organizational structure and the software they create.



Read more about the Conway's law here:

http://www.melconway.com/Home/Conways_Law.html

Microservices early adopters - Is there a common theme?

Many organizations had already successfully embarked on their journey to the microservices world. In this section, we will examine some of the front runners on the microservices space to analyze why they did what they did, and how they did it? The following is a curated list of organizations adopted microservices, based on the information available on the internet:

- **Netflix** (www.netflix.com): Netflix, an international, on-demand, media streaming company, is a pioneer in the microservices space. Netflix transformed their large pool of developers developing traditional monolithic code to smaller development teams producing microservices. These microservices work together to stream digital media to millions of Netflix customers. At Netflix, engineers started with monolithic architecture, went through the pain, and then broke the application into smaller units that are loosely coupled and aligned to business capability.
- **Uber** (www.uber.com): Uber, an international transportation network company, was started in 2008 with a monolithic architecture with single codebase. All services were embedded into the monolithic application. When Uber expanded their business from one city to multiple cities, challenges started. Uber then moved to SOA-based architecture by breaking the system into smaller independent units. Each module was given to different teams, empowered to choose their language, framework, and database. Uber has many microservices deployed in their ecosystem using RPC and REST.
- **Airbnb** (www.airbnb.com): Airbnb, a world leader providing trusted marketplace for accommodations, started with a monolithic application that performed all required functions of the business. It faced scalability issues with increased traffic. A single codebase became too complicated to manage, resulting in poor separation of concerns, and running into performance issues. Airbnb broke their monolithic application into smaller pieces, with separate codebases, running on separate machines, with separate deployment cycles. Airbnb has developed their own microservices or SOA ecosystem around these services.

- **Orbitz** (www.orbitz.com): Orbitz, an online travel portal, started with a monolithic architecture in the 2000s with a web layer, a business layer, and a database layer. As Orbitz expanded their business, they faced manageability and scalability issues with the monolithic-tiered architecture. Orbitz had then gone through continuous architecture changes. Later, Orbitz broke down their monolithic application into many smaller applications.
- **eBay** (www.ebay.com): eBay, one of the largest online retailers, started in the late 90s with a monolithic Perl application with **FreeBSD** as its database. eBay went through scaling issues as the business grew. It was consistently invested in improving architecture. In the mid 2000s, eBay moved to smaller decomposed systems based on Java and web services. They employed database partitions and functional segregation to meet the required scalability.
- **Amazon** (www.amazon.com): Amazon, one of the largest online retailer's website in 2001 was running on a big monolith application written on C++. The well-architected monolithic application was based on tiered architecture with many modular components. However, all these components were tightly coupled. As a result, Amazon was not able to speed up their development cycle by splitting teams into smaller groups. Amazon then separated out code as independent functional services wrapped with web services, and eventually advanced to microservices.
- **Gilt** (www.gilt.com): Gilt, an online shopping website, started in 2007 with a tiered monolithic Rails application with Postgres database at the back. Just like many other applications, as traffic volumes increased, this web application was not able to provide the required resiliency. Gilt went through an architecture overhaul by introducing Java and polyglot persistence. Later, Gilt moved to many smaller applications using microservices concept.
- **Twitter** (www.twitter.com): Twitter, one of the largest social websites, started with a three-tiered monolithic rails application in the mid 2000s. Later, when Twitter experienced growth in the user base, it went through an architecture refactoring cycle. With that refactoring, Twitter moved away from a typical web application to an API-based event-driven core. Twitter is using Scala and Java to develop microservices with polyglot persistence.

- **Nike** (www.nike.com): Nike, the world leader in apparels and footwear, transformed their monolithic applications to microservices. Like many other organizations, Nike was running with age old legacy applications that are hardly stable. In their journey, Nike moved to heavyweight commercial products with an objective to stabilize legacy applications, but ended up in monolithic applications that are expensive to scale, have long release cycles, and require too much manual work to deploy and manage applications. Later, Nike moved to microservices-based architecture, which brought down the development cycle considerably.

Monolithic migration as the common use case

When we analyze the preceding enterprises, there is one common theme. All these enterprises started with monolithic applications and transitioned to microservices architecture by applying learning and pain points from their previous editions.

Even today, many start-ups are starting with monolith as it is easy to start, conceptualize, and then slowly move them to microservices when the demand arises. Monolithic to microservices migration scenarios have an added advantage that we have all information upfront, readily available for refactoring.

Although it is a monolithic transformation for all of those enterprises, the catalysts were different for different organizations. Some of the common motivations are lack of scalability, long development cycles, process automation, manageability, and changes in business models.

While monolithic migrations are no brainers, there are opportunities to build ground-up microservices. More than building ground-up systems, look for an opportunity to build smaller services that are quick wins for the business. For example, adding a trucking service to an airline's end-to-end cargo management system, or adding a customer scoring service to a retailer's loyalty system. These could be implemented as independent microservices exchanging messages with their respective monolithic applications.

Another point is that many of the organizations are using microservices only for their business' critical and customer engagement applications, leaving the rest of their legacy monolithic applications to take its own trajectory.

Another important observation is that most of the organizations examined previously are at different levels of maturity in their microservices journey. When eBay transitioned from the monolithic application in the early 2000s, they functionally split the application into smaller, independent, deployable units. These logically divided units are wrapped with web services. While single responsibility and autonomy are their underpinning principles, the architectures are limited to the technologies and tools available at that point in time. Organizations such as Netflix and Airbnb built capabilities of their own to solve specific challenges they faced. To summarize, all of them are not truly microservices, but are small, business-aligned services following the same characteristics.

There is no state called definite or ultimate microservices. It is a journey, and it is evolving and maturing day by day. The mantra for architects and developers is the replaceability principle--build an architecture that maximizes the ability to replace its parts and minimizes the cost of replacing its parts. The bottom line is that enterprises shouldn't attempt to develop microservices by just following the hype.

Microservice frameworks

Microservices are already in the main stream. When developing microservices, there are some cross-cutting concerns that need to be implemented, such as externalized logging, tracing, embedded HTTP listener, health checks, and so on. As a result, significant efforts will go into developing these cross-cutting concerns. Microservices frameworks are emerged in this space to fill these gaps.

There are many microservices frameworks available apart from those that are mentioned specifically under the serverless computing. The capabilities vary between these microservice frameworks. Hence, it is important to choose the right framework for development.

Spring Boot, **Dropwizard**, and **Wildfly Swarm** are popular enterprise-grade HTTP/REST implementations for the development of microservices. However, these frameworks only provide minimalistic support for large-scale microservices development. Spring Boot, together with **Spring Cloud**, offers sophisticated support for microservices. **Spring Framework 5** introduced the reactive web framework. Combining Spring Boot and Spring Framework 5 reactive is a good choice for reactive style microservices. Alternatively, **Spring Streams** can also be used for the microservices development.

The following is a curated list of other microservices frameworks:

- Lightbend's **Lagom** (www.lightbend.com/lagom) is a full-fledged, sophisticated, and popular microservices framework for Java and Scala.
- WSO2 **Microservices For Java - MSF4J** (github.com/wso2/msf4j) is a lightweight, high performance microservices framework.
- **Spark** (sparkjava.com) is a micro framework to develop REST services.
- **Seneca** (senecaajs.org) is a microservices toolkit for Node JS in a fast and easy way similar to Spark.
- **Vert.x** (vertx.io) is a polyglot microservices toolkit to build reactive microservices quickly.
- **Restlet** (restlet.com) is a framework used to develop REST-based APIs in a quick and efficient way.
- **Payra-micro** (payara.fish) is used to develop web applications (war files) and run them on a standalone mode. Payra is based on Glass Fish.
- **Jooby** (jooby.org) is another micro-web framework that can be used to develop REST-based APIs.
- **Go-fasthttp** (github.com/valyala/fasthttp) is a high performance HTTP package for Go, useful to build REST services.
- **JavaLite** (javalite.io) is a framework to develop applications with HTTP endpoints.
- **Mantl** (mantl.io) is open source microservices framework come from Cisco for the microservice development and deployments.
- **Fabric8** (fabric8.io) is an integrated microservices development platform on top of Kubernetes, backed by Red Hat.
- **Hook.io** (hook.io) is another microservices deployment platform.
- **Vamp** (vamp.io) is an open source self-hosted platform to manage microservices that relies on container technologies.
- **Go Kit** (github.com/go-kit/kit) is a standard library for microservices using the Go language.
- **Micro** (github.com/micro/micro) is a microservice toolkit for Go language.
- **Lumen** (lumen.laravel.com) is a lightweight, fast micro-framework.
- **Restx** (restx.io) is a lightweight, REST development framework.
- **Gizmo** (github.com/NYTimes/gizmo) is a reactive microservices framework for Go.
- **Azure service fabric** (azure.microsoft.com/en-us/services/service-fabric/) has also emerged as a microservice development platform.

This list does not end here. There are many more in this category, such as Kontena, Gilliam, Magnetic, Eventuate, LSQ, and Stellient, which are some of the platforms supporting microservices.

The rest of this book will focus on building microservices using Spring Framework projects.

Summary

In this chapter, we learned about the relationship of the microservices architecture with a few other popular architecture styles.

We started with microservices relationships with SOA and Twelve-Factor Apps. We also examined the link between the microservices architecture and other architectures, such as serverless computing architecture and Lambda Architecture. We also learned the advantages of using microservices together with cloud and DevOps. We then analyzed examples of a few enterprises from different industries that successfully adopted microservices and their use cases. Finally, we reviewed some of the microservices frameworks evolved over a period of time.

In the next chapter, we will develop a few sample microservices to bring more clarity to our learnings in this chapter.

16

Building Microservices with Spring Boot

Developing microservices is not so tedious anymore--thanks to the powerful Spring Boot framework. Spring Boot is a framework for developing production-ready microservices in Java.

This chapter will move from the microservices theory explained in the previous chapters to hands-on practice by reviewing code samples. Here, we will introduce the Spring Boot framework, and explain how Spring Boot can help building RESTful microservices inline with the principles and characteristics discussed in the previous chapter. Finally, some of the features offered by Spring Boot for making the microservices production ready will be reviewed.

At the end of this chapter, you will have learned about the following:

- Setting up the latest Spring development environment
- Developing RESTful services using Spring Framework 5 and Spring Boot
- Building reactive microservices using Spring WebFlux and Spring Messaging
- Securing microservices using Spring Security and OAuth2
- Implementing cross-origin microservices
- Documenting Spring Boot microservices using Swagger
- Spring Boot Actuator for building production-ready microservices

Setting up a development environment

To crystalize microservices' concepts, a couple of microservices will be built. For that, it is assumed that the following components are installed:

- **JDK 1.8** (<http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>)
- **Spring Tool Suite 3.8.2 (STS)** (<https://spring.io/tools/sts/all>)
- **Maven 3.3.1** (<https://maven.apache.org/download.cgi>)

Alternately, other IDEs like **IntelliJ IDEA/NetBeans/Eclipse** could be used. Similarly, alternate build tools like **Gradle** can be used. It is assumed that Maven repository, class path, and other path variables are set properly for running STS and Maven projects.

This chapter is based on the following versions of Spring libraries:

- Spring Framework 5.0.0.RC1
- Spring Boot 2.0.0. M1



The focus of this chapter is not to explore the full features of Spring Boot, but to understand some of the essential and important features of Spring Boot, required when building microservices.

Spring Boot for building RESTful microservices

Spring Boot is a utility framework from the Spring team for bootstrapping Spring-based applications and microservices quickly and easily. The framework uses an opinionated approach over configurations for decision making, thereby reducing the effort required on writing a lot of boilerplate code and configurations. Using the 80-20 principle, developers should be able to kick start a variety of Spring applications with many default values. Spring Boot further presents opportunities to the developers for customizing applications by overriding auto-configured values.

Spring Boot not only increases the speed of development, but also provides a set of production-ready ops features such as health checks and metrics collections. Since Spring Boot masks many configuration parameters and abstracts many lower level implementations, it minimizes the chances of errors to a certain extent. Spring Boot recognizes the nature of the application based on the libraries available in the classpath, and runs the auto-configuration classes packaged in those libraries.

Often, many developers mistakenly see Spring Boot as a code generator, but, in reality, it is not. Spring Boot only auto-configures build files, for example, pom files in the case of Maven. It also sets properties, such as data source properties, based on certain opinionated defaults.

Consider the following dependencies in the file pom.xml:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>

<dependency>
    <groupId>org.hsqldb</groupId>
    <artifactId>hsqldb</artifactId>
    <scope>runtime</scope>
</dependency>
```

For instance, in the preceding case, Spring Boot understands that the project is set to use the Spring Data JPA and HSQL database. It automatically configures the driver class and other connection parameters.

One of the great outcomes of Spring Boot is that it almost eliminates the need to have traditional XML configurations. Spring Boot also enables microservices development by packaging all the required runtime dependencies into a fat executable jar.

Getting started with Spring Boot

These are the different ways in which Spring Boot-based application development can be started:

- By using **Spring Boot CLI** as a command-line tool
- By using IDEs like **Spring Tool Suite (STS)**, which provide Spring Boot, supported out of the box

- By using the **Spring Initializr** project at <http://start.spring.io>
- By using **SDKMAN! (The Software Development Kit Manager)** from <http://sdkman.io>

The first three options will be explored in this chapter, developing a variety of example services.

Developing a Spring Boot microservice

The easiest way to develop and demonstrate Spring Boot's capabilities is by using the Spring Boot CLI, a command-line tool.

The following are the steps to set up and run Spring Boot CLI:

1. Install the Spring Boot command-line tool by downloading the `spring-boot-cli-2.0.0.BUILD-M1-bin.zip` file from the following location URL:
<https://repo.spring.io/milestone/org/springframework/boot/spring-boot-cli/2.0.0.M1/>
2. Unzip the file into a directory of choice. Open a terminal window, and change the terminal prompt to the `bin` folder.



Ensure that the `/bin` folder is added to the system path so that Spring Boot can be run from any location. Otherwise, execute from the `bin` folder by using the command `./spring`.

3. Verify the installation with the following command. If successful, the Spring CLI version will be printed on the console as shown:

```
$spring --version  
Spring CLI v2.0.0.M1
```

4. As the next step, a quick REST service will be developed in groovy, which is supported out of the box in Spring Boot. To do so, copy and paste the following code using any editor of choice, and save it as `myfirstapp.groovy` into any folder:

```
@RestController  
class HelloworldController {  
    @RequestMapping("/")  
    String sayHello(){
```

```
        return "Hello World!"  
    }  
}
```

5. In order to run this groovy application, go to the folder where `myfirstapp.groovy` is saved, and execute the following command. The last few lines of the server startup log will be as shown in the following command snippet:

```
$spring run myfirstapp.groovy  
2016-12-16 13:04:33.208  INFO 29126 --- [runner-0]  
s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on  
port(s):  
8080 (http)  
2016-12-16 13:04:33.214  INFO 29126 --- [runner-0]  
o.s.boot.SpringApplication : Started application in 4.03  
seconds (JVM running for 104.276)
```

6. Open a browser window, and point the URL to `http://localhost:8080`; the browser will display the following message:

Hello World!

There is no war file created and no Tomcat server was running. Spring Boot automatically picked up Tomcat as the web server, and embedded it into the application. This is a very basic, minimal microservice. The `@RestController`, used in the previous code, will be examined in detail in the next example.

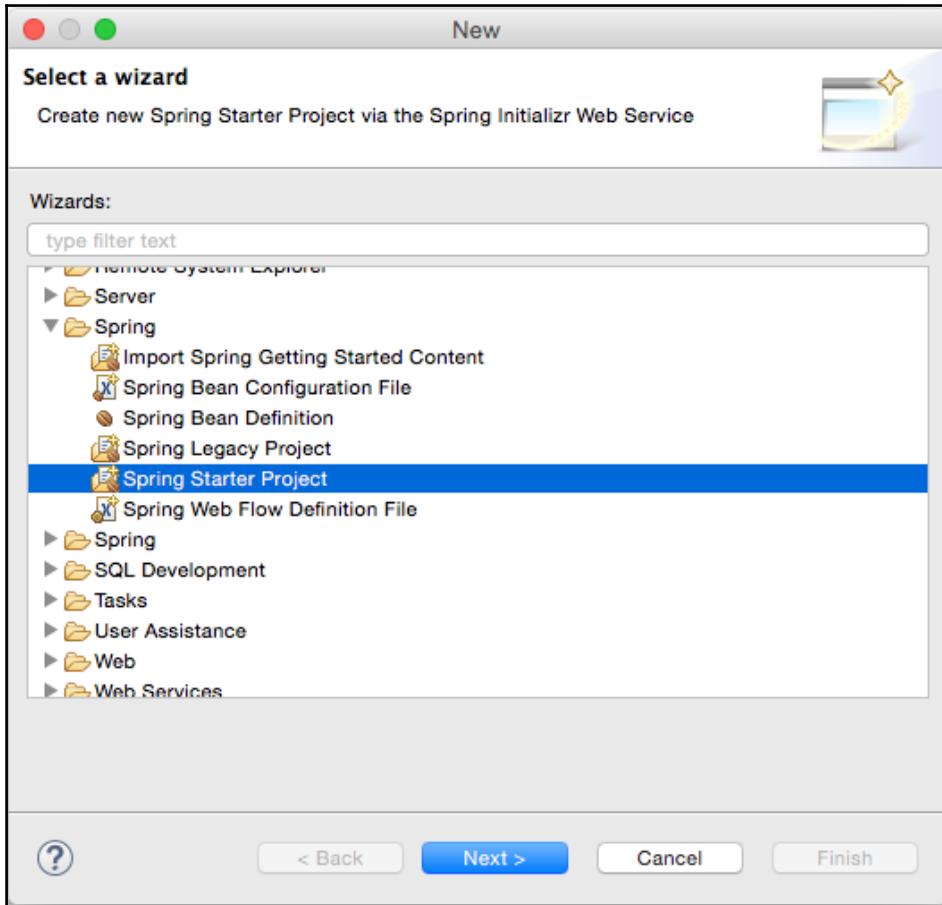
Developing our first Spring Boot microservice

In this section, we will demonstrate how to develop a Java-based REST/JSON Spring Boot service using STS.



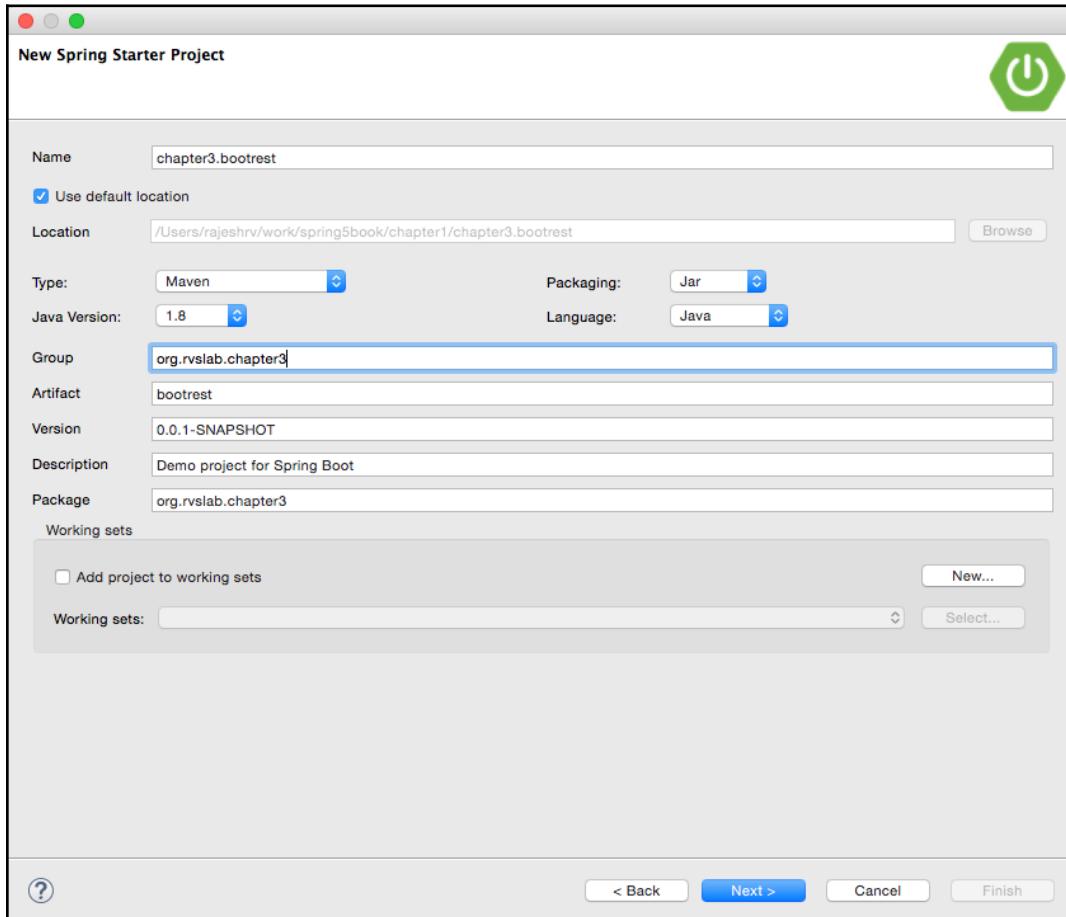
The full source code of this example is available as the `chapter3.Bootrest` project in the code files of this book under the following Git repository: <https://github.com/rajeshrv/Spring5Microservice>

1. Open STS, right-click in **Project Explorer** window, select **New Project**, then select Spring Starter Project as shown in the following screenshot. Then click on **Next**:



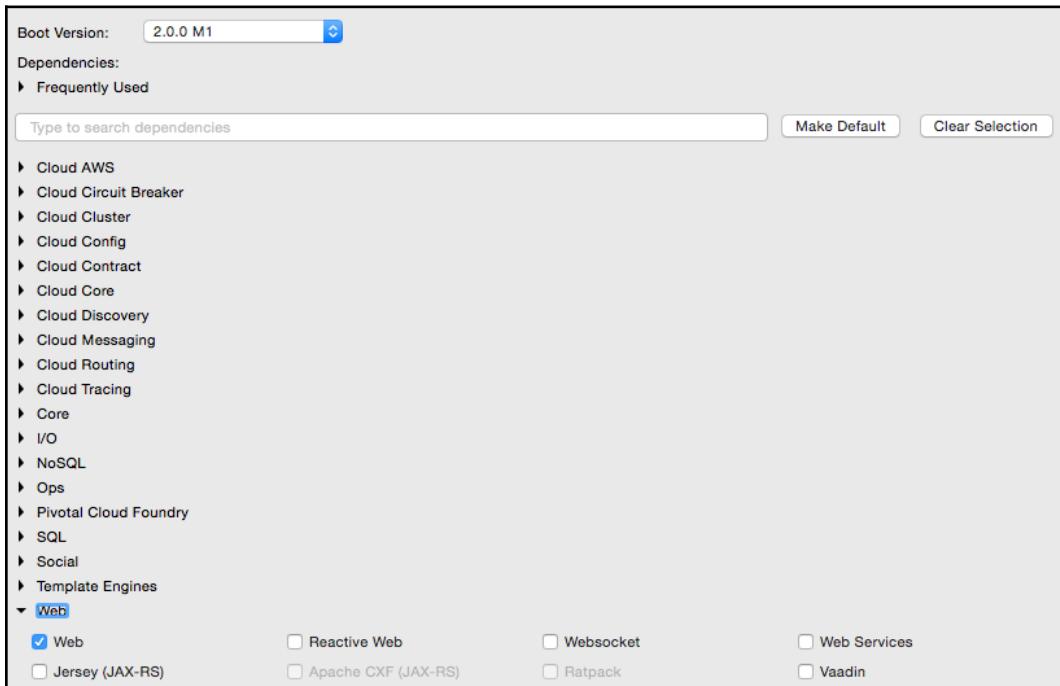
2. The Spring Starter Project is a basic template wizard, which provides a selection of a number of other starter libraries.
3. Type the project name as `chapter3.bootrest`, or any other name of your choice. It is important to choose the packaging as Jar. In traditional web applications, a war file is created, and then deployed into a servlet container, whereas, Spring Boot packages all the dependencies into a self-contained, autonomous jar with an embedded HTTP listener.

4. Select **Java Version** as **1.8**. Java 1.8 is recommended for Spring 5 applications. Change other Maven properties such as **Group**, **Artifact**, and **Package** as shown in the following screenshot:

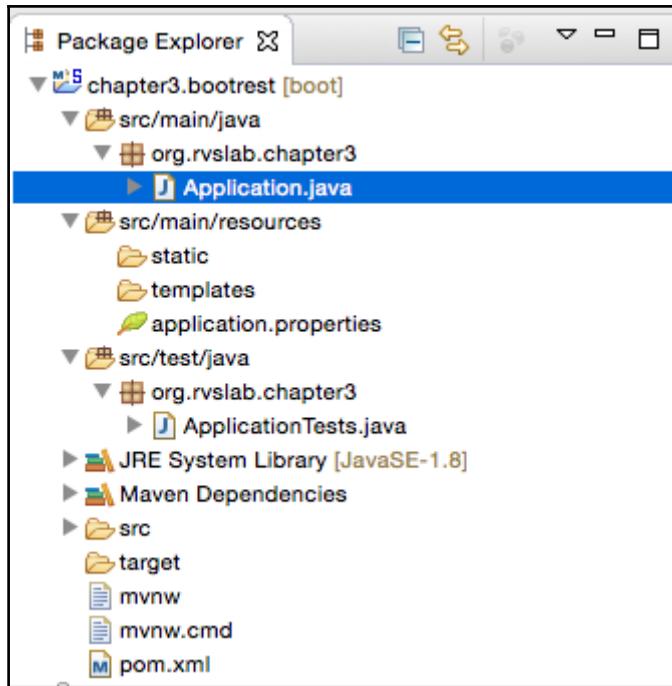


5. Once completed, click on **Next**.

6. The wizard will show the library options. In this case, since we are developing REST services, select **Web** under **Web**. This is an interesting step, which tells Spring Boot that a Spring MVC web application is being developed so that spring boot can include the necessary libraries, including Tomcat, as the HTTP listener and other configurations as required:



7. Click on **Finish**.
8. This will generate a project named `chapter3.bootrest` in **STS Project Explorer**:



- Let us examine the pom file. The parent element is one of the interesting aspects in pom.xml:

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.0.0.M1</version>
</parent>
```

Spring-boot-starter-parent is a **bill-of-materials (BOM)**, a pattern used by Maven's dependency management. The BOM is a special kind of pom used for managing different library versions required for a project. The advantage of using the spring-boot-starter-parent pom is that developers need not worry about finding the right, compatible versions of different libraries such as Spring, Jersey, Junit, Logback, Hibernate, Jackson, and more.

The starter pom has a list of Boot dependencies, sensible resource filtering, and sensible plug-in configurations required for the Maven builds.



Refer to the following link to see the different dependencies provided in the starter parent (version 2.0.0). All these dependencies can be overridden if required: <https://github.com/spring-projects/spring-boot/blob/a9503abb94b203a717527b81a94dc9d3cb4b1afa/spring-boot-dependencies/pom.xml>

The starter pom itself does not add jar dependencies to the project. Instead, it only adds library versions. Subsequently, when dependencies are added to pom.xml, they refer to the library versions from this pom.xml. Some of the properties are as shown next:

```
<activemq.version>5.14.5</activemq.version>
<commons-collections.version>3.2.2
</commons-collections.version>
<hibernate.version>5.2.10.Final</hibernate.version>
<jackson.version>2.9.0.pr3</jackson.version>
<mssql-jdbc.version>6.1.0.jre8</mssql-jdbc.version>
<spring.version>5.0.0.RC1</spring.versiontomcat.version>8.5.15</tomcat.version
```

Reviewing the dependency section of our pom.xml, one can see that this is a clean and neat pom file with only two dependencies:

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Since **Web** is selected, spring-boot-starter-web adds all dependencies required for a Spring MVC project. It also includes dependencies to Tomcat as an embedded HTTP listener. This provides an effective way to get all the dependencies required as a single bundle. Individual dependencies could be replaced with other libraries, such as replacing Tomcat with Jetty.

Similar to web, Spring Boot also comes up with a number of `spring-boot-starter-*` libraries, such as `amqp`, `aop`, `batch`, `data-jpa`, `thymeleaf`, and so on.

The last thing to be reviewed in the file `pom.xml` is the Java 8 property as shown here:

```
<java.version>1.8</java.version>
```

By default, the parent pom adds Java 6. It is recommended to override the Java version to 8 for Spring 5.

10. Let us now examine `Application.java`. Spring Boot, by default, generated a class `org.rvslab.chapter3.Application.java` under `src/main/java` for bootstrapping:

```
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

There is only a main method in the application, which will be invoked at startup, as per the Java convention. The main method bootstraps the Spring Boot application by calling the run method on `SpringApplication`.

`Application.class` is passed as a parameter to tell Spring Boot that this is the primary component.

More importantly, the magic is done by the `@SpringBootApplication` annotation. `@SpringBootApplication` is a top-level annotation, which encapsulates three other annotations, as shown in the following code snippet:

```
@Configuration
@EnableAutoConfiguration
@ComponentScan
public class Application {
```

The `@Configuration` annotation hints that the contained class declares one or more `@Bean` definitions. `@Configuration` is meta-annotated with `@Component`, therefore, they are candidates for component scanning.

`@EnableAutoConfiguration` tells Spring Boot to automatically configure the Spring application based on the dependencies available in the class path.

11. Let us examine `application.properties`--a default `application.properties` file is placed under `src/main/resources`. It is an important file for configuring any required properties for the Spring Boot application. At the moment, this file is kept empty, and will be revisited with some test cases later in this chapter.
12. Let us examine `ApplicationTests.java` under `src/test/java`. This is a placeholder for writing test cases against the Spring Boot application.
13. As the next step, add a REST endpoint. Let us edit `Application.java` under `src/main/java`, and add a RESTful service implementation. The RESTful service is exactly the same as what was done in the previous project. Append the following code at the end of the `Application.java` file:

```
@RestController
class GreetingController{
    @GetMapping("/")
    Greet greet(){
        return new Greet("Hello World!");
    }
}
class Greet{
    private String message;
    public Greet() {}
    public Greet(String message){
        this.message = message;
    }
    //add getter and setter
}
```

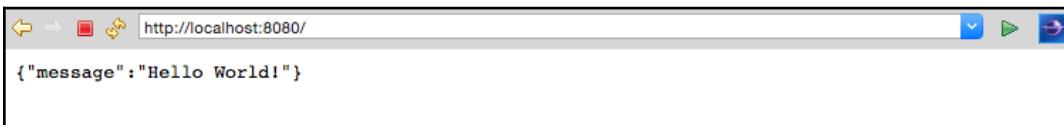
14. To run, go to **Run As | Spring Boot App**. Tomcat will be started on the 8080 port:

```
rvslab:chapter3.bootrest rajeshrv$ java -jar target/bootrest-0.0.1-SNAPSHOT.jar
 .\n /_\n( )\n \\\n\n :: Spring Boot ::   (v2.0.0.M1)

2017-06-09 12:06:59.503  INFO 3909 --- [           main] org.rvslab.chapter3.Application      : Starting Application v0.0.1-SNAPSHOT on rvslab.local with PID 3909 (/Users/rajeshrv/work/spring5bookprefinal/chapter3/chapter3.bootrest/target/bootrest-0.0.1-SNAPSHOT.jar started by rajeshrv in /Users/rajeshrv/work/spring5bookprefinal/chapter3/chapter3.bootrest)
```

15. We can notice the following things from the log:

- Spring Boot gets its own process ID (in this case, 3909)
- Spring Boot automatically starts the Tomcat server at the local host, port 8080
- Next, open a browser and point to `http://localhost:8080`. This will show the JSON response as follows:



The key difference between the legacy service and this one is that the Spring Boot service is self-contained. To make this clearer, run the Spring Boot application outside STS.

Open a terminal window, go to the project folder, and run Maven as follows:

```
$ maven install
```

This preceding command will generate a fat jar under the target folder of the project. Running the application from the command line shows the following:

```
$ java -jar target/bootrest-0.0.1-SNAPSHOT.jar
```

As one can see, `bootrest-0.0.1-SNAPSHOT.jar` is self-contained, and could be run as a standalone application. At this point, the jar is as thin as 14 MB. Even though the application is not more than just a *Hello world*, the Spring Boot service just developed practically follows the principles of microservices.

Testing Spring Boot microservice

There are multiple ways to test REST/JSON Spring Boot microservices. The easiest way is to use a web browser or a curl command pointing to the URL, like this:

```
curl localhost:8080
```

There are number of tools available for testing RESTful services such as Postman, Advanced Rest Client, SOAP UI, Paw, and more.

In this example, for testing the service, the default test class generated by Spring Boot will be used. Adding a new test case to `ApplicationTests.java` results in this:

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
public class ApplicationTests {
    @Autowired
    private TestRestTemplate restTemplate;
    @Test
    public void testSpringBootApp() throws JsonProcessingException,
    IOException {
        String body = restTemplate.getForObject("/", String.class);
        assertThat(new ObjectMapper().readTree(body)
            .get("message")
            .textValue())
            .isEqualTo("Hello World!");
    }
}
```

Note that `@SpringBootTest` is a simple annotation for testing Spring Boot applications, which enables Spring Boot features during test execution.

`webEnvironment=WebEnvironment.RANDOM_PORT` property directs the Spring Boot application to bind to a random port. This will be handy when running many Spring Boot services as part of a regression test. Also note that `TestRestTemplate` is being used for calling the RESTful service. `TestRestTemplate` is a utility class, which abstracts the lower-level details of the HTTP client, and also automatically discovers the actual port used by Spring Boot.

To test this, open a terminal window, go to the project folder, and run `mvn install`.

HATEOAS-enabled Spring Boot microservice

In the next example, **Spring Initializr** will be used to create a Spring Boot project. Spring Initializr is a drop-in replacement for the STS project wizard, and provides a web UI for configuring and generating a Spring Boot project. One of the advantages of the Spring Initializr is that it can generate a project through the website, which can then be imported into any IDE.

In this example, the concept of **HyperMedia As The Engine Of Application State (HATEOAS)** for REST-based services and the **Hypertext Application Language (HAL)** browser will be examined.

HATEOAS is useful for building conversational style microservices which exhibit strong affinity between UI and its backend services.

HATEOAS is a REST service pattern in which navigation links are provided as part of the payload metadata. The client application determines the state, and follows the transition URLs provided as part of the state. This methodology is particularly useful in responsive mobile and web applications where the client downloads additional data based on user navigation patterns.

The HAL browser is a handy API browser for hal+json data. HAL is a format based on JSON, which establishes conventions for representing hyperlinks between resources. HAL helps APIs to be more explorable and discoverable.



The full source code of this example is available as the chapter3.boothateoas project in the code files of this book under the following Git repository: <https://github.com/rajeshrv/Spring5Microservice>

Here are the concrete steps for developing a HATEOAS sample using Spring Initializr:

1. In order to use Spring Initializr, go to <https://start.spring.io>:

SPRING INITIALIZR bootstrap your application now

Generate a with and Spring Boot

Project Metadata

Artifact coordinates
Group
Artifact

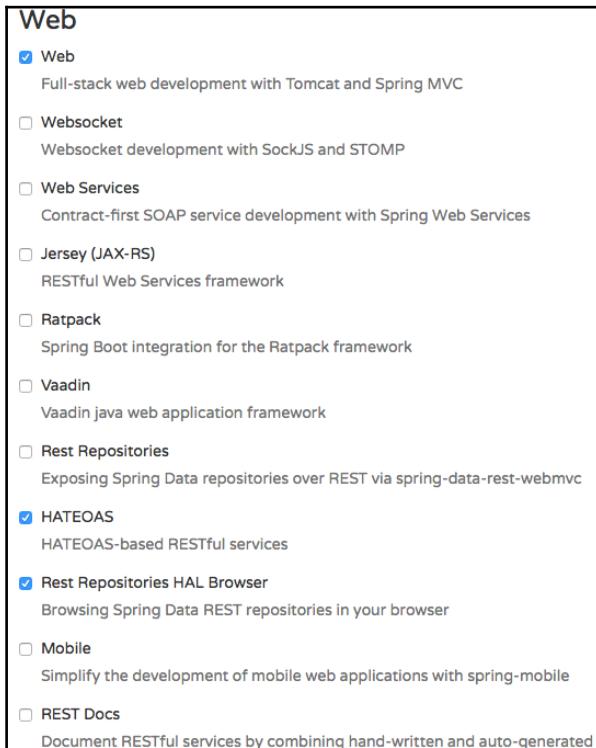
Dependencies

Add Spring Boot Starters and dependencies to your application
Search for dependencies
Selected Dependencies Web Security JPA Actuator Devtools...

Generate Project

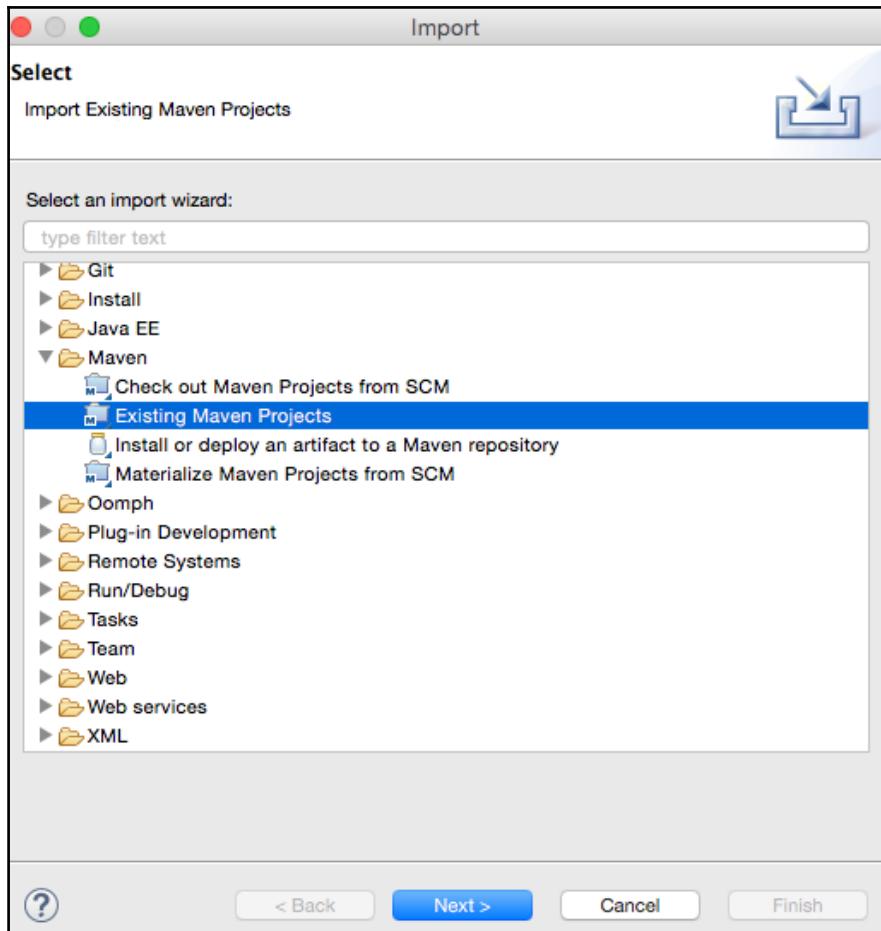
Don't know what to look for? Want more options? [Switch to the full version.](#)

2. Fill the details such as **Maven Project**, Spring Boot version, **Group**, and **Artifact**, as shown in the preceding screenshot, and click on **Switch to the full version** below the **Generate Projects** button. Select **Web**, **HATEOAS**, and **Rest Repositories HAL Browser**. Make sure the Java version is 8, and the package type is selected as Jar, as shown in the following screenshot:



3. Once selected, hit the **Generate Project** button. This will generate a Maven project, and download the project as a zip file into the download directory of the browser.
4. Unzip the file, and save it to a directory of your choice.

5. Open STS, go to the **File** menu, and click on **Import**:



6. Navigate to **Maven | Existing Maven Projects**, then click on **Next**.
7. Click on **browse** next to the root directory, and select the unzipped folder. Click on **Finish**. This will load the generated Maven project in **STS Project Explorer**.

8. Edit BoothateoasApplication.java to add a new REST endpoint as follows:

```
@RequestMapping("/greeting")
@ResponseBody
public HttpEntity<Greet> greeting(@RequestParam(value =
"name",
required = false, defaultValue = "HATEOAS") String name) {
    Greet greet = new Greet("Hello " + name);
    greet.add(linkTo(methodOn(GreetingController.class)
        .greeting(name)).withSelfRel());
    return new ResponseEntity<Greet>(greet, HttpStatus.OK);
}
```

9. Note that this is the same GreetingController class as in the previous example. But a method named greeting was added this time. In this new method, an additional optional request parameter is defined and defaulted to HATEOAS. The following code adds a link to the resulting JSON--in this case, it adds the link to the same API.
10. The following code adds a self-reference web link to the Greet object:

```
"href": "http://localhost:8080/greeting?name=HATEOAS":
```

```
greet.add(linkTo(methodOn(
    GreetingController.class).greeting(name)).withSelfRel());
```

In order to do this, we need to extend the Greet class from ResourceSupport as shown in following code. The rest of the code remains the same:

```
class Greet extends ResourceSupport{
```

11. Add is a method on ResourceSupport. The linkTo and methodOn are static methods of ControllerLinkBuilder, a utility class for creating links on controller classes. The methodOn method does a dummy method invocation, and linkTo creates a link to the controller class. In this case, we use withSelfRel to point it to self.
12. This will essentially produce a link, /greeting?name=HATEOAS, by default. A client can read the link, and initiate another call.
13. Run as Spring Boot App. Once the server startup is completed, point the browser to http://localhost:8080.

14. This will open the HAL browser window. In the **Explorer** field, type /greeting?name=World!, and click on the **Go** button. If everything is fine, the response details will be seen in the HAL browser as shown in the following screenshot:

The screenshot shows the HAL Browser interface for Spring Data REST. The top bar says "The HAL Browser (for Spring Data REST)".

Explorer: Shows the URL "/greeting?name=world" in the input field and a "Go!" button.

Custom Request Headers: An empty section.

Properties: Displays a JSON object:

```
{  
    "message": "Hello world"  
}
```

Links: A table with columns: rel, title, name / index, docs, GET, NON-GET. It contains one row for "self".

rel	title	name / index	docs	GET	NON-GET
self					

Inspector: **Response Headers** section shows:

```
200 success  
  
Date: Fri, 09 Jun 2017 09:05:45 GMT  
Transfer-Encoding: chunked  
Content-Type: application/hal+json; charset=UTF-8
```

Response Body: Displays the JSON response from the server:

```
{  
    "message": "Hello world",  
    "_links": {  
        "self": {  
            "href": "http://localhost:8080/greeting?name=world"  
        }  
    }  
}
```

As shown in the preceding screenshot, the **Response Body** has the result with a link, href, which points back to the same service. This is because we pointed the reference to itself. Also, review the **Links** section. The little green box against self is the navigable link.

It does not make much sense in this simple example, but this could be handy in larger applications with many related entities. Using the links provided, the client can easily navigate back and forth between these entities with ease.

Reactive Spring Boot microservices

Reactive microservices mentioned in [Chapter 2, Related Architecture Styles and Use cases](#), basically highlight the need of asynchronously integrating microservices in an ecosystem. Even though external service calls primarily get benefits from reactive style programming, reactive principles are useful in any software development, as it improves resource efficiency and scalability characteristics. Therefore, it is important build microservices using reactive programming principles.

There are two ways we can implement reactive microservices. The first approach is to use the Spring **WebFlux** in the Spring Framework 5. This approach uses reactive style web server for microservices. The second approach is to use a messaging server such as RabbitMQ for asynchronous interaction between microservices. In this chapter, we will explore both the options mentioned here.

Reactive microservices using Spring WebFlux

Reactive programming in Java is based on the **Reactive Streams** specification. Reactive stream specification defines the semantics for asynchronous stream processing or flow of events between disparate components in a non-blocking style.

Unlike the standard observer pattern, Reactive Streams allow to maintain sequence, notification on completion, and notification when there is an error with full backpressure support. With backpressure support, a receiver can set terms such as how much data it wants from the publisher. Also, the receiver can start receiving data only when data is ready to be processed. Reactive Streams are particularly useful for handling different thread pools for different components, or in the case of integrating slow and fast components.



Reactive streams specification is now adopted as part of Java 9 `java.util.concurrent.Flow`. Its semantics are similar to `CompletableFuture` in Java 8 with lambda expressions for collecting results.

Spring Framework 5 integrates reactive programming principles at its core as WebFlux. Spring 5 WebFlux is based on Reactive Streams specification. Under the hood, Spring's Web Reactive Framework uses the Reactor project (<https://projectreactor.io>) for implementing reactive programming. Reactor is an implementation of Reactive Streams specification. With Spring Framework, developers can also choose to use RxJava instead of Reactor.

In this section, we will see how to build Reactive Spring Boot microservices using Spring 5 WebFlux libraries. These libraries help developers to create asynchronous, non-blocking HTTP servers with full backpressure support, without coding callback methods. Note that it is not a one-size-fits solution in all cases and, if not used properly, this can backfire on the quality of services. Also, developers need to be sure that the downstream components support full reactive programming.

In order to get full power of reactive programming, reactive constructs should be used end-to-end from client, endpoint, and to the repository. That means, if a slow client accesses a reactive server, then the data read operations in the repository can slow down to match the slowness of the client.



At the time of writing this book, Spring Data Kay M1 supports reactive drivers for Mongo DB, Apache Cassandra, and Redis. The reactive CRUD repository, `ReactiveCrudRepository`, is a handy interface for implementing reactive repositories.

Spring WebFlux supports two options for implementing Spring Boot applications. The first approach is annotation based with `@Controller` and the other annotations generally used with Spring Boot. The second approach is functional programming Java 8 lambda style coding.

Let us build a sample using the annotation style reactive programming using WebFlux.

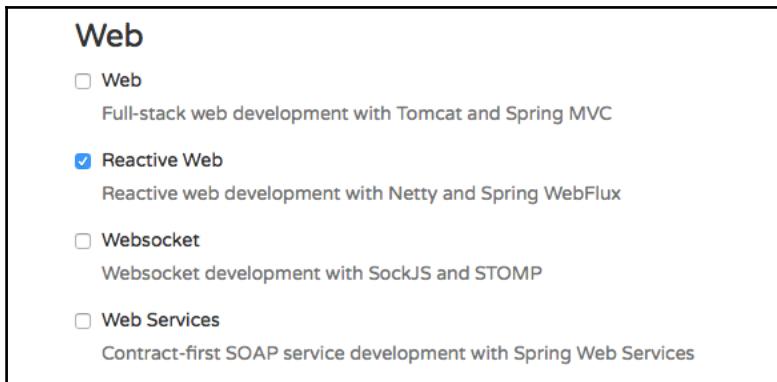


The full source code of this example is available as the `chapter3.webflux` project in the code files of this book under the following Git repository: <https://github.com/rajeshrv/Spring5Microservice>

Follow these steps to build a reactive Spring Boot application:

1. Go to <https://start.spring.io>, and generate a new Spring Boot project.

2. Select **Reactive Web** under the **Web** section:



3. Generate project, and import the newly generated project into STS.
4. Examine `pom.xml`; there is only one difference. Instead of `spring-boot-starter-web`, this project uses `spring-boot-starter-webflux` in the dependency section. Following is the dependency to Spring Webflux:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
```

5. Add the `GreetingController` and `Greet` classes from `chapter3.bootrest` to the `Application.java` class.
6. Run this project, and test with a browser by pointing to `http://localhost:8080`. You will see the same response.
7. Let us add some Reactive APIs to enable reactive programming to the Boot application. Let us modify `RestController`. Let us add a construct, `Mono`, as follows:

```
@RequestMapping("/")
Mono<Greet> greet() {
    return Mono.just(new Greet("Hello World!"));
}
```

In this case, the response body uses `Mono`, which means that the `Greet` object will be serialized only when `Mono` is completed in an asynchronous non-blocking mode. Since we have used `Mono`, this method just creates a single definitive item.

In this case, Mono is used to declare a logic which will get executed as soon as the object is deserialized. You may consider Mono as a placeholder (deferred) for zero or one object with a set of callback methods.

In case of Mono as a parameter to a controller method, the method may be executed even before the serialization gets over. The code in the controller will decide what we want to do with the Mono object. Alternately, we can use **Flux**. Both these constructs will be explained in detail in the next section.

Let us now change the client. Spring 5 reactive introduced `WebClient` and `WebTestClient` as an alternate to `RestTemplate`. `WebClient` is fully support reactive under the hood.

The client-side code is as follows:

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.DEFINED_PORT)
public class ApplicationTests {
    WebTestClient webClient;
    @Before
    public void setup() {
        webClient = WebTestClient.bindToServer()
            .baseUrl("http://localhost:8080").build();
    }

    @Test
    public void testWebFluxEndpoint() throws Exception {
        webClient.get().uri("/")
            .accept(MediaType.APPLICATION_JSON)
            .exchange()
            .expectStatus().isOk()
            .expectBody(Greet.class).returnResult()
            .getResponseBody().getMessage().equals("Hello World!");
    }
}
```

`WebTestClient` is a purpose build class for testing `WebFlux` server. `WebClient`, another client class similar to `RestTemplate` is more handy when invoking `WebFlux` from a non-testing client. The preceding test code first creates a `WebTestClient` with the server URL. Then it executes a `get` method on the / URL, and verifies it against the existing result.

8. Run the test from the command prompt using `mvn install`. You will not notice any difference in functionality, but the execution model has changed under the hood.

Understanding Reactive Streams

Let us understand the Reactive Streams specification. Reactive Streams has just four interfaces, which are explained as follows:

Publisher

A Publisher holds the source of data, and then publishes data elements as per the request from a subscriber. A Subscriber can then attach a subscription on the Publisher. Note that the `subscribe` method is just a registration method, and will not return any result:

```
public interface Publisher<T> {  
    public void subscribe(Subscriber<? super T> s);  
}
```

Subscriber

Subscriber subscribes to a Publisher for consuming streams of data. It defines a set of callback methods, which will be called upon those events. Complete is when everything is done and success. Note, that all these are callback registrations, and the methods themselves do not respond with any data:

```
public interface Subscriber<T> {  
    public void onSubscribe(Subscription s);  
    public void onNext(T t);  
    public void onError(Throwable t);  
    public void onComplete();  
}
```

Subscription

A Subscription is shared by exactly one Publisher and one Subscriber for the purpose of mediating data exchange between this pair. Data exchange happens when the subscriber calls `request`. `cancel` is used basically to stop the subscription as seen in this example:

```
public interface Subscription {
    public void request(long n);
    public void cancel();
}
```

Processor

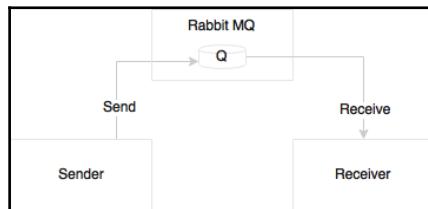
A Processor represents a processing stage--which is both a Subscriber and a Publisher, and **MUST** obey the contracts of both. A of can be chained by connecting a Publisher and Subscriber:

```
public interface Processor<T, R> extends Subscriber<T>,
    Publisher<R> {
}
```

Reactor has two implementations for Publisher--**Flux** and **Mono**. Flux can emit 0...N events, whereas, Mono is for a single event (0...1). Flux is required when many data elements or a list of values is transmitted as Streams.

Reactive microservices using Spring Boot and RabbitMQ

In an ideal case, all microservice interactions are expected to happen asynchronously using publish subscribe semantics. Spring Boot provides a hassle-free mechanism to configure messaging solutions:



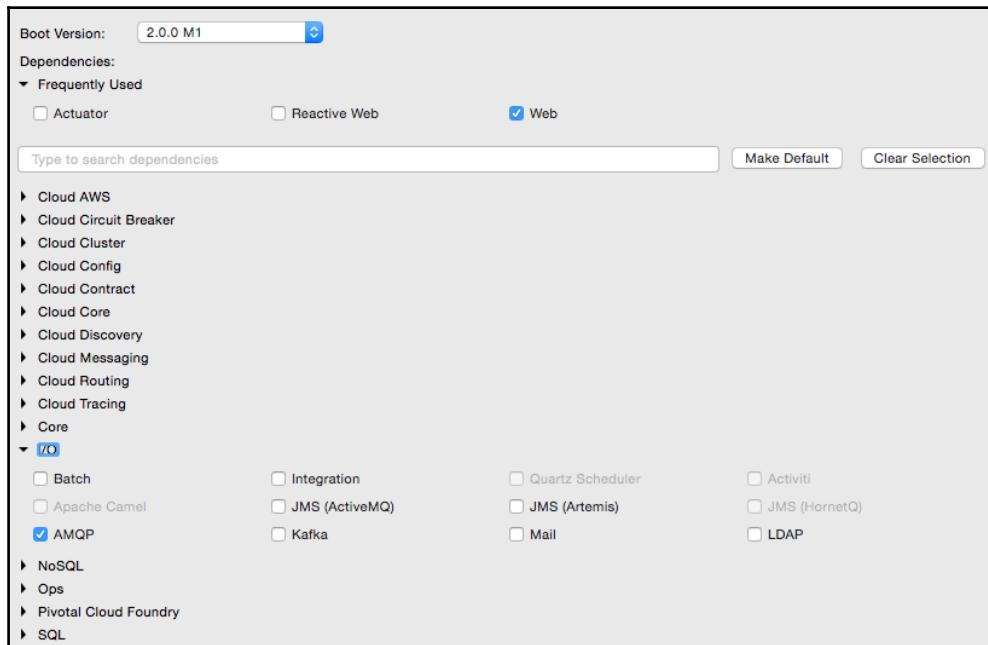
In this next example, we will create a Spring Boot application with a sender and receiver, both connected through an external queue.



The full source code of this example is available as the chapter3.bootmessaging project in the code files of this book under the following Git repository: <https://github.com/rajeshrv/Spring5Microservice>

Let us follow these steps to create a Spring Boot reactive microservice using RabbitMQ:

1. Create a new project using STS to demonstrate this capability. In this example, instead of selecting **Web**, select **AMQP** under **I/O**:



2. RabbitMQ will also be needed for this example. Download and install the latest version of RabbitMQ from <https://www.rabbitmq.com/download.html>. RabbitMQ 3.5.6 is used in this book.
3. Follow the installation steps documented on the site. Once ready, start the RabbitMQ server like this:

```
$ ./rabbitmq-server
```

4. Make the configuration changes to the `application.properties` file to reflect the RabbitMQ configuration. The following configuration uses the default port, username, and password of RabbitMQ:

```
spring.rabbitmq.host=localhost  
spring.rabbitmq.port=5672  
spring.rabbitmq.username=guest  
spring.rabbitmq.password=guest
```

5. Add a message sender component and a queue named `TestQ` of the type `org.springframework.amqp.core.Queue` to `Application.java` under `src/main/java`. The `RabbitMessagingTemplate` template is a convenient way to send messages, and abstracts all the messaging semantics. Spring Boot provides all boilerplate configurations for sending messages:

```
@Component
class Sender {
    @Autowired
    RabbitMessagingTemplate template;

    @Bean
    Queue queue() {
        return new Queue("TestQ", false);
    }
    public void send(String message) {
        template.convertAndSend("TestQ", message);
    }
}
```

6. For receiving a message, all that is needed is a `@RabbitListener` annotation. Spring Boot auto-configures all required boilerplate configurations:

```
@Component
class Receiver {
    @RabbitListener(queues = "TestQ")
    public void processMessage(String content) {
        System.out.println(content);
    }
}
```

7. The last piece of this exercise is to wire the sender to our main application, and implement the `CommandLineRunner` interface's `run` method to initiate the message sending. When the application is initialized, it invokes the `run` method of the `CommandLineRunner`:

```
@SpringBootApplication
public class Application implements CommandLineRunner{
    @Autowired
    Sender sender;
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
    @Override
```

```
public void run(String... args) throws Exception {  
    sender.send("Hello Messaging..!!!");  
}  
}
```

8. Run the application as a Spring Boot application, and verify the output. The following message will be printed on the console:

Hello Messaging..!!!

Implementing security

It is important to secure the microservices. This will be more significant when there are many microservices communicating with each other. Each service needs to be secured, but at the same time, security shouldn't surface as an overhead. In this section, we will learn some basic measures to secure microservices.



The full source code of this example is available as the `chapter3.security` project in the code files of this book under the following Git repository: <https://github.com/rajeshrv/Spring5Microservice>

Perform the following steps for building this example:

- Create a new Spring Starter project, and select **Web** and **Security** (under core)
- Name the project as `chapter3.security`
- Copy rest endpoint from `chapter3.bootrest`

Securing a microservice with basic security

Adding basic authentication to Spring Boot is pretty simple. The `pom.xml` file will have the following dependency. This will include the necessary Spring security library files:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

This will, by default, assume that basic security is required for this project. Run the application, and test it with a browser. The browser will ask for the login username and password.

The default basic authentication assumes the `.`. The default password will be printed on the console at startup:

```
Using default security password: a7d08e07-ef5f-4623-b86c-
63054d25baed
```

Alternately, the username and password can be added in `application.properties` as shown next:

```
security.user.name=guest
security.user.password=guest123
```

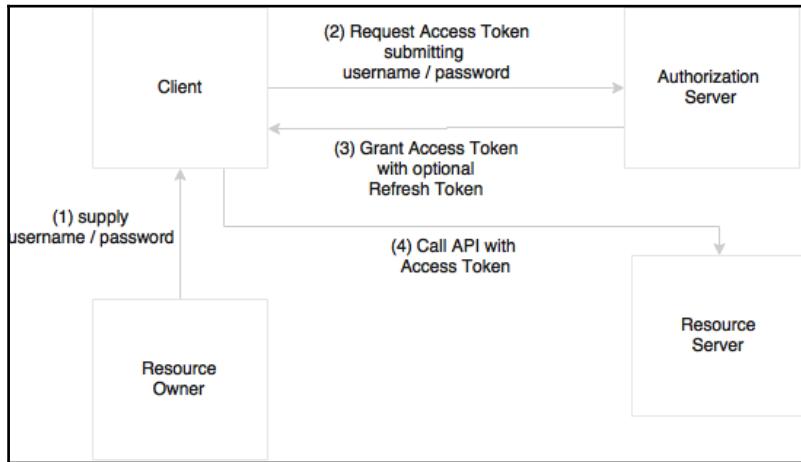
Securing microservice with OAuth2

In this section, we will see the basic Spring Boot configuration for OAuth2. When a client application requires access to a protected resource, the client sends a request to an authorization server. The authorization server validates the request, and provides an access token. This access token will be validated for every client-to-server request. The request and response sent back and forth depends on the grant type.



Read more about OAuth and grant types at this link: <http://oauth.net>

The resource owner password credentials grant approach will be used in the following example:



In this case, as shown in the preceding diagram, the resource owner provides the client with a username and password. The client then sends a token request to the authorization server by providing the credentials. The authorization server authorizes the client, and returns an access token. On every subsequent request, the server validates the client token.

To implement OAuth2 in our example, follow these steps:

1. As the first step, update `pom.xml` with `oauth2` dependency as follows:

```
<dependency>
    <groupId>org.springframework.security.oauth</groupId>
    <artifactId>spring-security-oauth2</artifactId>
</dependency>

<!-- below dependency is explicitly required when
     testing OAuth2 with Spring Boot 2.0.0.M1 -->
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-crypto</artifactId>
    <version>4.2.2.RELEASE</version>
</dependency>
```

2. Next, add two new annotations-- `@EnableAuthorizationServer` --and `@EnableResourceServer` to `Application.java`. The `@EnableAuthorizationServer` annotation creates an authorization server with an in-memory repository to store client tokens and to provide clients with a username, password, client ID, and secret. `@EnableResourceServer` is used to access the tokens. This enables a spring security filter that authenticates via an incoming OAuth2 token.



In our example, both, authorization server and resource server, are the same. But in practice, these two will be running.

```
@EnableResourceServer  
@EnableAuthorizationServer  
@SpringBootApplication  
public class Application {
```

3. Add the following properties to the `application.properties` file:

```
security.user.name=guest  
security.user.password=guest123  
security.oauth2.client.client-id: trustedclient  
security.oauth2.client.client-secret: trustedclient123  
security.oauth2.client.authorized-grant-types:  
authorization_code,refresh_token,password
```

4. Add another test case to test OAuth2 as follows:

```
@Test  
public void testOAuthService() {  
    ResourceOwnerPasswordResourceDetails resource =  
        new ResourceOwnerPasswordResourceDetails();  
    resource.setUsername("guest");  
    resource.setPassword("guest123");  
    resource.setAccessTokenUri("http://localhost:8080/oauth  
        /token");  
    resource.setClientId("trustedclient");  
    resource.setClientSecret("trustedclient123");  
    resource.setGrantType("password");  
    resource.setScope(Arrays.asList(new String[]  
        {"read", "write", "trust"}));  
    DefaultOAuth2ClientContext clientContext =  
        new DefaultOAuth2ClientContext();  
    OAuth2RestTemplate restTemplate =  
        new OAuth2RestTemplate(resource, clientContext);
```

```
Greet greet = restTemplate  
    .getForObject("http://localhost:8080", Greet.class);  
Assert.assertEquals("Hello World!", greet.getMessage());  
}
```

As shown in the preceding code, a special rest template, `OAuth2RestTemplate`, is created by passing the resource details encapsulated in a resource details object. This rest template handles the OAuth2 processes underneath. The access token URI is the endpoint for the token access.

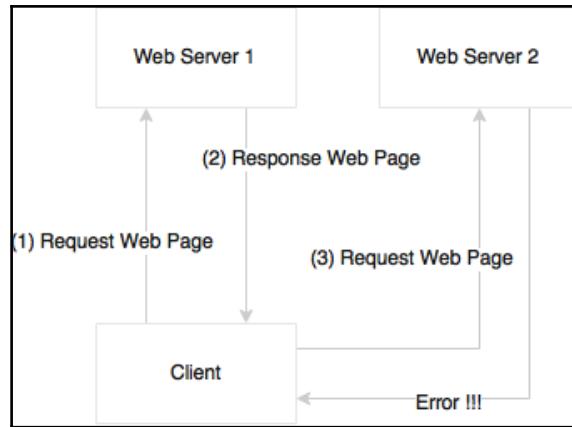
5. Rerun the application using maven install. The first two test cases will fail, and the new one will succeed. This is because the server accepts only OAuth2-enabled requests.

These are quick configurations provided by Spring Boot out of the box, but are not good enough to be production grade. We may need to customize `ResourceServerConfigurer` and `AuthorizationServerConfigurer` to make them production ready. Regardless, the approach remains the same.

Enabling cross origin for microservices interactions

Browsers are generally restricted when client-side web applications running from one origin request data from another origin. Enabling cross origin access is generally termed as **CORS (Cross Origin Resource Sharing)**.

This is particularly important when dealing with microservices, such as when the microservices run on separate domains, and the browser tries to access these microservices from one browser after another:



The preceding example showcases how to enable cross origin requests. With microservices, since each service runs with its own origin, it will easily get into the issue of a client-side web application, which consumes data from multiple origins. For instance, a scenario where a browser client accesses customers from the customer microservice, and order history from the order microservices is very common in microservices world.

Spring Boot provides a simple declarative approach for enabling cross origin requests.

The following code example shows how to use a microservice to enable cross origin:

```
@RestController
class GreetingController{
    @CrossOrigin
    @RequestMapping("/")
    Greet greet(){
        return new Greet("Hello World!");
    }
}
```

By default, all origins and headers are accepted. We can further customize the cross origin annotations by giving access to a specific origin as follows. The `@CrossOrigin` annotation enables a method or a class to accept cross origin requests:

```
@CrossOrigin("http://mytrustedorigin.com")
```

Global CORS could be enabled by using the `WebMvcConfigurer` bean, and customizing the `addCorsMappings(CorsRegistry registry)` method.

Spring Boot actuators for microservices instrumentation

The previous sections explored most of the Spring Boot features required for developing a microservices. In this section, we will explore some of the production-ready operational aspects of Spring Boot.

Spring Boot actuators provide an excellent out-of-the-box mechanism for monitoring and managing Spring Boot microservices in production.



The full source code of this example is available as the `chapter3.bootactuator` project in the code files of this book under the following Git repository: <https://github.com/rajeshrv/Spring5Microservice>

Create another Spring starter project, and name it as `chapter3.bootactuator.application`; this time, select the **Web, HAL browser, hateoas, and Actuator** dependencies. Similar to `chapter3.bootrest`, add a `GreeterController` endpoint with the `greet` method. Add `management.security.enabled=false` to the `application.properties` file to grant access to all endpoints.

Do the following to execute the application:

1. Start the application as Spring Boot App.
2. Point the browser to `localhost:8080/application`. This will open the HAL browser. Review the **Links** section.

A number of links are available under the **Links** section. These are automatically exposed by the Spring Boot actuator:

rel	title	name / index	docs	GET	NON-GET
self				→	!
health				→	!
trace				→	!
dump				→	!
loggers				→	!
configprops				→	!
beans				→	!
info				→	!
autoconfig				→	!
env				→	!
metrics				→	!
mappings				→	!
auditevents				→	!
heapdump				→	!

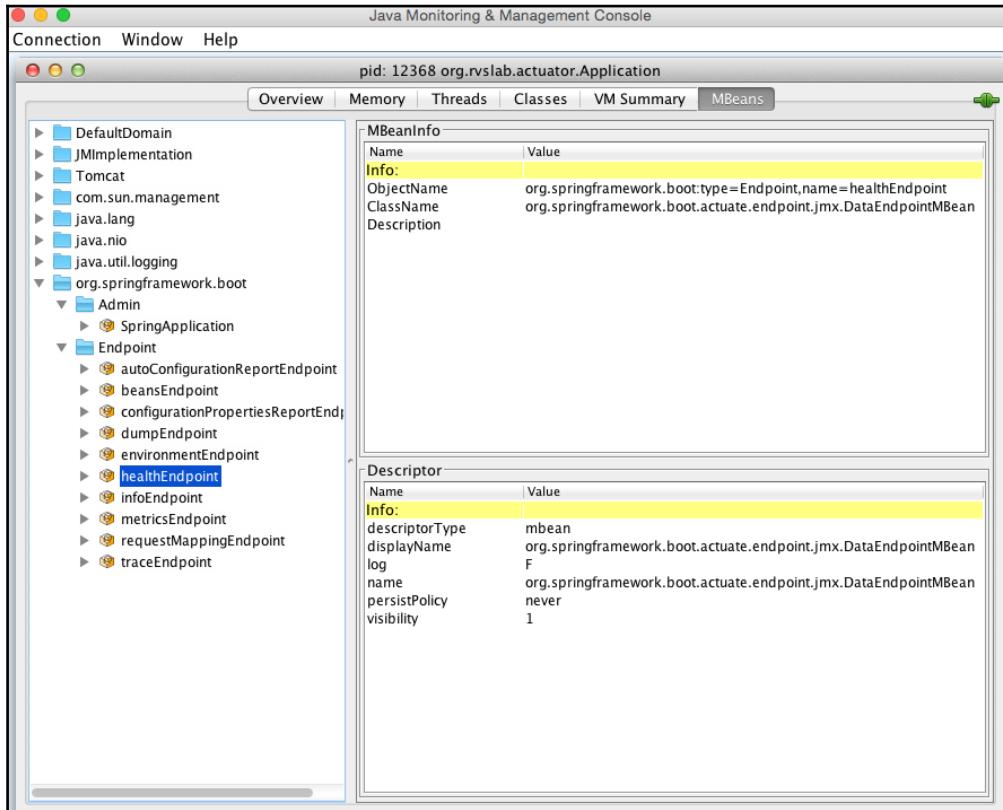
Some of the important links are listed as follows:

- **dump**: Performs a thread dump and displays the result
- **mappings**: Displays a list of all the http request mappings
- **info**: Displays information about the application
- **health**: Displays the health condition of the application
- **autoconfig**: Displays the auto configuration report
- **metrics**: Shows different metrics collected from the application

From the browser, individual endpoints are accessible using `/application/<endpoint_name>`. For example, to access the `/health` endpoint, point the browser to `localhost:8080/application/health`.

Monitoring using JConsole

Alternately, we can use the JMX console to see Spring Boot information. Connect to the remote Spring Boot instance from a jconsole. The Boot information will be shown as follows:



Monitoring using ssh

Spring Boot provides remote access to the Boot application using ssh. The following command connects to the Spring Boot application from a terminal window. The password can be customized by adding the `shell.auth.simple.user.password` property in the `application.properties` file. The updated `application.properties` will look as follows:

```
shell.auth.simple.user.password=admin
```

Use a terminal window to connect to the remote boot application using the following code:

```
$ ssh -p 2000 user@localhost
```

When connected with the preceding command, similar actuator information can be accessed. The following is an example of the metrics information accessed through the CLI:

- **help**: lists all the options available
- **dashboard**: dashboard is one interesting feature, which shows a lot of system-level information

Adding a custom health module

Adding a new custom module to the Spring Boot application is not so complex. For demonstrating this feature, assume that if a service gets more than two transactions in a minute, then the server status will be set as Out of Service.

In order to customize this, we have to implement the `HealthIndicator` interface, and override the `health` method. The following code is a quick and dirty implementation to do the job:

```
class TPSCounter {  
    LongAdder count;  
    int threshold = 2;  
    Calendar expiry = null;  
  
    TPSCounter(){  
        this.count = new LongAdder();  
        this.expiry = Calendar.getInstance();  
        this.expiry.add(Calendar.MINUTE, 1);  
    }  
    boolean isExpired(){  
        return Calendar.getInstance().after(expiry);  
    }  
  
    boolean isWeak(){  
        return (count.intValue() > threshold);  
    }  
  
    void increment(){  
        count.increment();  
    }  
}
```

The preceding code is a simple **Plain Old Java Object (POJO)** class, which maintains the transaction counts window. The `isWeak` method checks whether the transaction in a particular window has reached its threshold. `isExpired` checks whether the current window is expired or not. The `increment` method simply increases the counter value.

As the next step, implement our custom health indicator class, `TPSHealth`. This is done by extending `HealthIndicator` as follows:

```
@Component
class TPSHealth implements HealthIndicator {
    TPSCounter counter;
    @Override
    public Health health() {
        boolean health = counter.isWeak();
        if (health) {
            return Health.outOfService()
                .withDetail("Too many requests", "OutofService")
                .build();
        }
        return Health.up().build();
    }

    void updateTx(){
        if(counter == null || counter.isExpired()){
            counter = new TPSCounter();
        }
        counter.increment();
    }
}
```

The `health` method checks whether the counter `isWeak` or not. If it is weak, it marks the instance as out of service.

Finally, we autowire `TPSHealth` into the `GreetingController` class, and then call `health.updateTx()` in the `greet` method, like this:

```
Greet greet() {  
    logger.info("Serving Request....!!!!");  
    health.updateTx();  
    return new Greet("Hello World!");  
}
```

Go to the `/application/health` endpoint in the HAL browser, and see the status of the server.

Now open another browser, point to `http://localhost:8080`, and fire the service twice or thrice. Go back to the `/application/health` endpoint, and refresh to see the status. It would have been changed to out of service.

In this example, since there is no action taken other than collecting the health status, new service calls will still go through even though the status is out of service. But in the real world, a program should read the `/application/health` endpoint, and block further requests going to that instance.

Building custom metrics

Just like health, customization of the metrics is also possible. The following example shows how to add a counter service and gauge service, just for demonstration purpose:

```
@Autowired  
CounterService counterService;  
  
@Autowired  
GaugeService gaugeService;
```

And add the following methods to the `greet` method:

```
this.counterService.increment("greet.txnCount");  
this.gaugeService.submit("greet.customgauge", 1.0);
```

Restart the server, and go to `/application/metrics` to see the new gauge and counter added already reflected there.

Documenting microservices

The traditional approach of API documentation is either to write service specification documents or to use static service registries. With a large number of microservices, it would be hard to keep the documentation of APIs in sync.

Microservices can be documented in many ways. This section will explore how microservices can be documented using the popular Swagger framework. In the following examples, we will use `Springfox` libraries for generating REST API documentation. `Springfox` is a set of Java Spring-friendly library.

Create a new Spring Starter project, and choose **Web** in the library selection window. Name the project as `chapter3.swagger`.



The full source code of this example is available as the `chapter3.swagger` project in the code files of this book under the following Git repository:

<https://github.com/rajeshrv/Spring5Microservice>

Since `Springfox` libraries are not part of the Spring suite, edit the `pom.xml` file, and add the `springfox-swagger` library dependencies. Add the following dependencies to the project:

```
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>2.6.1</version>
</dependency>

<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>2.6.1</version>
</dependency>
```

Create a REST service similar to the services created earlier, but also add the `@EnableSwagger2` annotation as shown next:

```
@SpringBootApplication
@EnableSwagger2
public class Application {
```

This is all that is required for a basic swagger documentation. Start the application, and point the browser to `http://localhost:8080/swagger-ui.html`. This will open the **swagger** API documentation page, as seen in this screenshot:

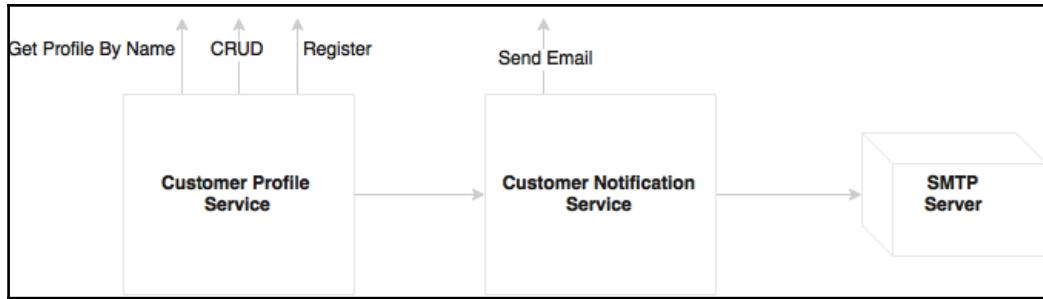
The screenshot shows the Swagger UI interface. At the top, there's a green header bar with the title 'swagger' and a 'default (/v2/api-docs)' link. Below the header, the title 'Api Documentation' is displayed. Underneath it, 'Apache 2.0' is mentioned. The main content area shows two sections: 'basic-error-controller : Basic Error Controller' and 'greeting-controller : Greeting Controller'. The 'greeting-controller' section is expanded, showing a 'GET / greet' operation. The response model is defined as a JSON object with a single key 'message' of type 'string'. Below this, there's a 'Response Content Type' dropdown set to 'application/json'. A 'Response Messages' table lists HTTP status codes 401 (Unauthorized) and 403 (Forbidden). The table has columns for 'HTTP Status Code', 'Reason', 'Response Model', and 'Headers'.

As shown in the preceding screenshot, the Swagger lists out the possible operations on the `GreetingController` class. Click on the `GET` operation. This expands the `GET` row, which provides an option to try out the operation.

Putting it all together - Developing a customer registration microservice example

So far, the examples we have seen are not more than just a simple *hello world*. Putting our learnings together, this section demonstrates an end-to-end customer profile microservice implementation. The customer profile microservices will demonstrate the interaction between different microservices. It will also demonstrate microservices with business logic and primitive data stores.

In this example, two microservices--**Customer Profile Service** and **Customer Notification Service**-- will be developed:



As shown in the preceding diagram, the customer profile microservice exposes methods to create, read, update, and delete a customer, and a registration service for registering a customer. The registration process applies certain business logic, saves the customer profile, and sends a message to the customer notification microservice. The customer notification microservice accepts the message sent by the registration service, and sends an e-mail message to the customer using an SMTP server. Asynchronous messaging is used for integrating customer profile with the customer notification service.

The customer microservices class domain model is shown in the following diagram:



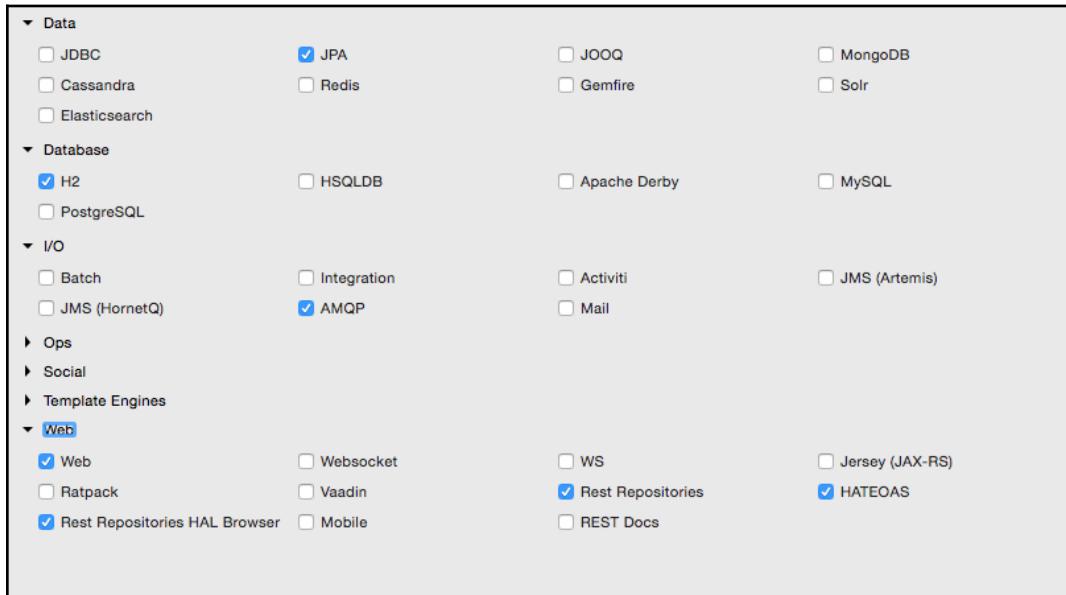
The `CustomerController` in the diagram is the REST endpoint, which invokes a component class, `CustomerComponent`. The component class/bean handles all the business logic. `CustomerRepository` is a Spring data JPA repository defined for handling persistence of the `Customer` entity.



The full source code of this example is available as the `chapter3.bootcustomer` and `chapter3.bootcustomernotification` projects in the code files of this book.

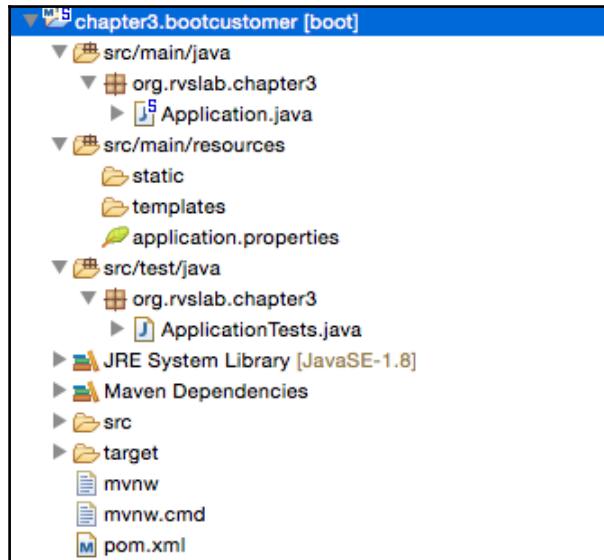
Follow the steps listed next to build this example:

1. Create a new Spring Boot project, and call it `chapter3.bootcustomer` the same way as earlier. Select the checkbox of the following options in the starter module selection screen:



This will create a web project with **JPA**, **Rest Repository**, and **H2** as database. **H2** is a tiny in-memory embedded database with easy-to-demonstrate database features. In the real world, it is recommended to use an appropriate enterprise-grade database. This example uses **JPA** for defining persistence entities and the REST repository for exposing REST-based repository services.

The project structure will look like the following screenshot:



2. Start building the application by adding an entity class named `Customer`. For simplicity, there are only three fields added to the customer entity which are auto-generated--`Id` field, `name`, and `email`:

```
@Entity
class Customer {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)

    private Long id;
    private String name;
    private String email;
```

3. Add a `Repository` class for handling persistence handling of customer. The `CustomerRepository` extends the standard `JpaRepository`. That means, all the CRUD methods and default finder methods are automatically implemented by the Spring Data JPA:

```
@RepositoryRestResource
interface CustomerRespository extends JpaRepository<Customer, Long>{
    Optional<Customer> findByName(@Param("name") String name);
}
```

In the preceding example, we added a new method, `findByName`, to the `Repository` class, which essentially searches for a customer based on customer name, and returns a `Customer` object if there is a matching name.

`@RepositoryRestResource` enables repository access through RESTful services. This also enables **HATEOAS** and **HAL** by default. Since for CRUD methods there is no additional business logic required, we will leave it as it is, without controller or component classes. Using **HATEOAS** helps us to navigate through the `CustomerRepository` methods effortlessly.



Note that there is no configuration added anywhere to point to any database. Since **H2** libraries are in the class path, all configurations are done by default by Spring Boot, based on **H2** auto-configuration.

4. Update `Application.java` by adding `CommandLineRunner` to initialize the repository with some customer records, as follows:

```
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
    @Bean
    CommandLineRunner init(CustomerRepository repo) {
        return (evt) -> {
            repo.save(new Customer("Adam", "adam@boot.com"));
            repo.save(new Customer("John", "john@boot.com"));
            repo.save(new Customer("Smith", "smith@boot.com"));
            repo.save(new Customer("Edgar", "edgar@boot.com"));
            repo.save(new Customer("Martin", "martin@boot.com"));
            repo.save(new Customer("Tom", "tom@boot.com"));
            repo.save(new Customer("Sean", "sean@boot.com"))
        };
    }
}
```

`CommandLineRunner`, defined as a bean, indicates that it should run when it is contained in a `SpringApplication`. This will insert six sample customer records to the database at startup.

5. At this point, run the application as Spring Boot App. Open the HAL browser by pointing the browser URL to `http://localhost:8080`.

6. In the **Explorer**, point to `http://localhost:8080/customers` and click on **Go**. This will list all customers in the **Response Body** section of the HAL browser.
7. In the **Explorer** section, enter the following URL, and click on **Go**. This will automatically execute paging and sorting on the repository, and return the result--`http://localhost:8080/customers?size=2&page=1&sort=name`

Since the page size is set as two, and first page is requested, it will come back with two records in a sorted order.

8. Review the **Links** section. As shown in the following diagram, it will facilitate navigating first, next, previous, and last. These are done using the HATEOAS links automatically generated by the repository browser:

rel	title	name / index	docs	GET	NON-GET
first					
prev					
self					
next					
last					
profile					
search					

9. Also, one can explore the details of a customer by selecting the appropriate link, <http://localhost:8080/customers/2>.
10. As the next step, add a controller class, `CustomerController`, for handling the service endpoints. There is only one endpoint in this class, `/register`, which is used for registering a customer. If successful, it returns the customer object as the response:

```
@RestController
class CustomerController{
    @Autowired
    CustomerRegistrar customerRegistrar;

    @RequestMapping( path="/register", method =
        RequestMethod.POST)
    Customer register(@RequestBody Customer customer){
        return customerRegistrar.register(customer);
    }
}
```

The `CustomerRegistrar` component is added for handling business logic. In this case, there is only minimal business logic added to the component. In this component class, while registering a customer, we will just check whether the customer name already exists in the database or not. If it does not exist, then a new record should be inserted, otherwise, an error message should be returned:

```
@Component
class CustomerRegistrar {
    CustomerRespository customerRespository;
    @Autowired
    CustomerRegistrar(CustomerRespository customerRespository) {
        this.customerRespository = customerRespository;
    }
    // ideally repository will return a Mono object
    public Mono<Customer> register(Customer customer) {
        if(customerRespository
            .findByName(customer.getName())
            .isPresent())
```

```
        System.out.println("Duplicate Customer.  
        No Action required");  
    else {  
        customerRepository.save(customer);  
    }  
    return Mono.just(customer);  
}  
}
```

11. Restart the Boot application, and test using the HAL browser using the following URL: <http://localhost:8080>.
12. Point the Explorer field to <http://localhost:8080/customers>. Review the results in the **Links** section:

rel	title	name / index	docs	GET	NON-GET
self					 Perform non-GET request
profile					
search					

13. Click on **NON-GET** against self. This will open a form for creating a new customer.

14. Fill the form and change the action as shown in the preceding screenshot. Click on the **Make Request** button. This will call the register service and register the customer. Try giving a duplicate name to test the negative case as follows:

The screenshot shows a modal dialog titled "Create/Update" with a sub-section titled "Customer". Inside, there are two input fields: "Name" containing "World" and "Email" containing "world@hello.com". Below these is a section labeled "Action:" with a dropdown set to "POST" and a text input field containing the URL "http://localhost:8080/register". At the bottom right of the dialog is a blue "Make Request" button.

Let us complete the last part in the example by integrating the customer notification service for notifying the customer. When registration is successful, an e-mail should be sent to the customer by asynchronously calling the customer notification microservice.

Perform the following steps to build the customer notification microservice:

1. First update `CustomerRegistrar` for calling the second service. This is done through messaging. In this case, we inject a `Sender` component for sending a notification to the customer by passing the customer's e-mail address to the sender:

```
@Component  
 @Lazy  
 class CustomerRegistrar {  
     CustomerRepository customerRespository;  
     Sender sender;  
     @Autowired  
     CustomerRegistrar(CustomerRepository customerRespository,  
     Sender sender) {
```

```
        this.customerRepository = customerRespository;
        this.sender = sender;
    }
    // ideally repository will return a Mono object
    public Mono<Customer> register(Customer customer) {
        if(customerRespository.findByName(
            customer.getName()).isPresent())
            System.out.println("Duplicate Customer.
                No Action required");
        else {
            customerRespository.save(customer);
            sender.send(customer.getEmail());
        }
        return Mono.just(customer); //HARD CODED BECOSE THERE
            IS NO REACTIVE REPOSITORY.
    }
}
```

The sender component will be based on RabbitMQ and AMQP. In this example, RabbitMessagingTemplate is used as explored in the last messaging example:

```
@Component
@Lazy
class Sender {
    @Autowired
    RabbitMessagingTemplate template;
    @Bean
    Queue queue() {
        return new Queue("CustomerQ", false);
    }
    public void send(String message){
        template.convertAndSend("CustomerQ", message);
    }
}
```



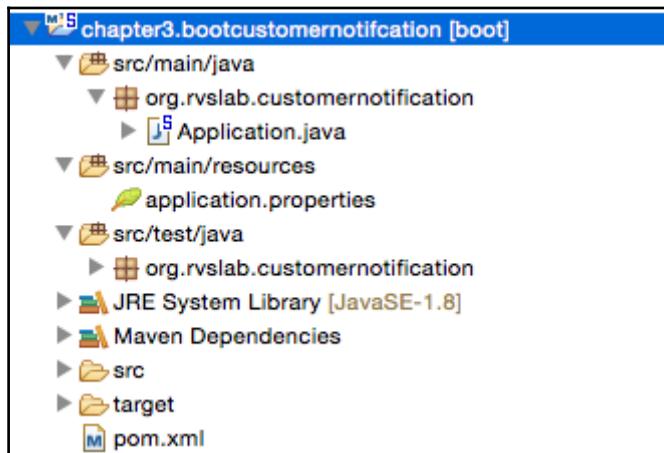
@Lazy is a useful annotation, which helps to increase the boot startup time. Those beans will be initialized only when the need arises.

2. We will also update the Application.property file to include RabbitMQ-related properties:

```
spring.rabbitmq.host=localhost  
spring.rabbitmq.port=5672  
spring.rabbitmq.username=guest  
spring.rabbitmq.password=guest
```

3. We are ready to send the message. For consuming the message and sending emails, we will create a notification service. For this, let us create another Spring Boot service, chapter3.bootcustomernotification. Make sure that the AMQP and Mail starter libraries are selected when creating the Spring Boot service. Both AMQP and Mail are under I/O.

The package structure of the chapter3.bootcustomernotification project is as shown in following screenshot:



4. Add a Receiver class. The Receiver class waits for a message on the customer. This will receive a message sent by the Customer Profile service. On the arrival of a message, it sends an email:

```
@Component  
class Receiver {  
    @Autowired
```

```
Mailer mailer;

@Bean
Queue queue() {
    return new Queue("CustomerQ", false);
}

@RabbitListener(queues = "CustomerQ")
public void processMessage(String email) {
    System.out.println(email);
    mailer.sendMail(email);
}
}
```

5. Add another component for sending an email to the customer. We use `JavaMailSender` to send the email:

```
@Component
class Mailer {
    @Autowired
    private JavaMailSender javaMailService;

    public void sendMail(String email) {
        SimpleMailMessage mailMessage=new SimpleMailMessage();
        mailMessage.setTo(email);
        mailMessage.setSubject("Registration");
        mailMessage.setText("Successfully Registered");
        javaMailService.send(mailMessage);
    }
}
```

Behind the scene, Spring Boot automatically configures all the parameters required for `JavaMailSender`.

Perform the following steps to test the application:

1. To test SMTP, a test setup for SMTP is required to ensure that the mails are going out. In this example, Fake SMTP will be used.
2. Download the Fake SMTP server from <http://nilhcem.github.io/FakeSMTP>.
3. Once you have downloaded `fakeSMTP-2.0.jar`, run the SMTP server by executing the following command:

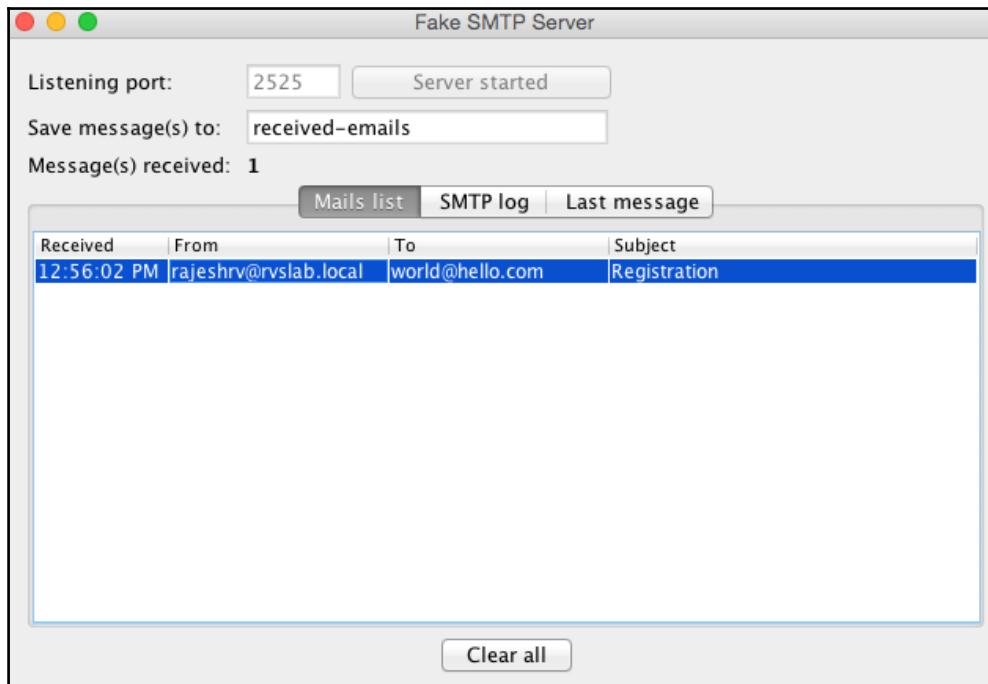
```
$ java -jar fakeSMTP-2.0.jar
```

4. This will open a GUI for monitoring email messages. Click on the **Start Server** button next to the listening port text box.

5. Update `Application.properties` with the following configuration parameters to connect to RabbitMQ as well as to the mail server:

```
spring.rabbitmq.host=localhost  
spring.rabbitmq.port=5672  
spring.rabbitmq.username=guest  
spring.rabbitmq.password=guest  
spring.mail.host=localhost  
spring.mail.port=2525
```

6. We are ready to test our microservices end to end. Start both Spring Boot Apps. Open the browser, and repeat the customer creation steps through the HAL browser. In this case, immediately after submitting the request, we will be able to see the e-mail in the SMTP GUI.
7. Internally, the Customer Profile service asynchronously calls the Customer Notification service, which, in turn, sends the email message to the SMTP server:



Summary

In this chapter, we learned about Spring Boot, and its key features for building production-ready applications.

We explored the previous generation web applications, and then compared how Spring Boot makes developers' lives easier in developing fully qualified microservices. We also saw HTTP-based and message-based asynchronous reactive microservices. Further, we explored how to achieve some of the key capabilities required for microservices such as security, HATEOAS, cross-origin, and so on, with practical examples. We also saw how the Spring Boot actuators help the operations teams, and also how to customize our needs. Then, documenting microservices APIs was also discussed. We closed the chapter with a complete example that puts together all our learnings.

In the next chapter, we will take a step back, and perform a pragmatic analysis of microservices with the help of practical day-to-day scenarios.

17

Microservices Capability Model

In the previous chapter, [Chapter 4, Applying Microservices Concepts](#), you learned some of the practical design aspects of the microservices development. In this chapter, we will take a step back and put together our learnings into a capability model.

What is the importance of a microservices capability model? Microservices are not as simple as developing web applications with UI, business logic, and databases. This is good enough for simple services, or when dealing with fewer microservices.

Developers need to think beyond service implementation when working with large-scale microservices. There are a number of ecosystem capabilities required for the successful delivery of microservices. It is important to ensure that those required capabilities are in place as a precondition. Unfortunately, there is no standard reference model available for microservice implementations.

Even though the capabilities required for microservice implementation may vary based on the solution context, this book attempts to build a generic microservice capability model instead of building a low-level reference architecture. At the end of this chapter, we will also examine a maturity model for microservice adoption.

In the chapter, you will learn about the following:

- A capability model for the microservices ecosystem
- An overview of each capability and its importance in a microservice ecosystem
- An overview of some tools and technologies available supporting these capabilities
- A maturity model for microservices

Microservices capability model

In the service-oriented world, there are reference architectures available, which can be used as a basis for SOA implementations. For example, a comprehensive SOA Reference Architecture is defined by The Open Group, which is available for public reference.

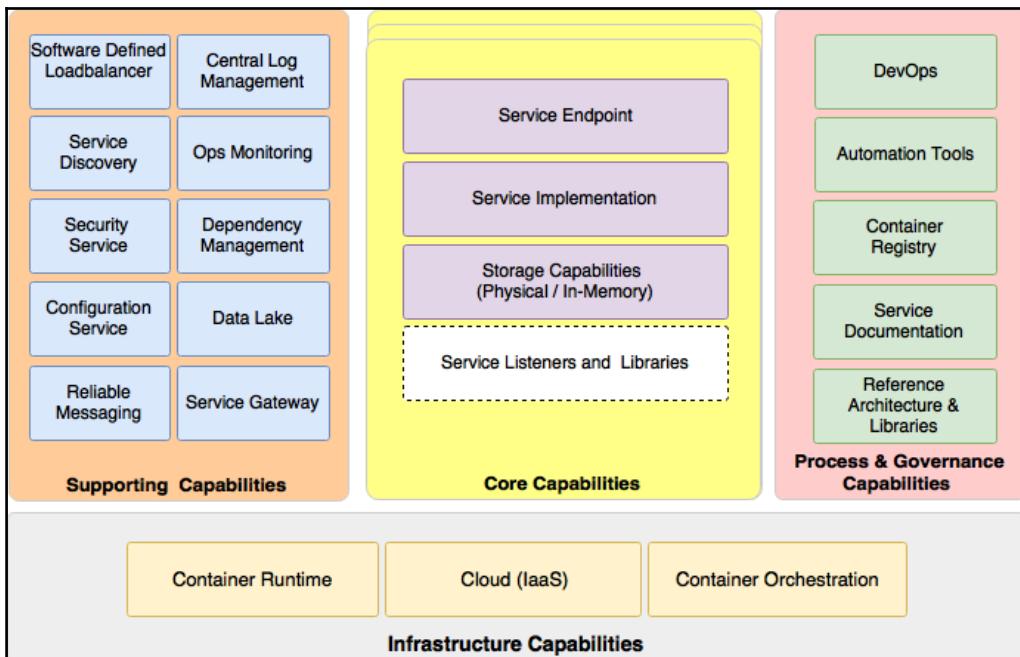


The Open Group SOA Reference Architecture is available at http://www.opengroup.org/soa/source-book/soa_refarch/index.htm.

However, there is no standard or reference architecture for microservices. It is still an evolving space. Many of the architectures available publicly today are from tool vendors, and they seem to be inclined towards their own tools stack.

The microservices capability model defined in this chapter is based on design guidelines, common patterns, and best practice solutions for designing and developing microservices.

The following diagram depicts the microservices capability model, which will be used as a reference model for the rest of the chapters in this book:



The capability model is broadly classified into four areas:

- Core capabilities, which are embedded in each microservices
- Supporting capabilities
- Infrastructure capabilities
- Process and governance capabilities

Core capabilities

Core capabilities are those components generally packaged inside a single microservice. For example, let's take an Order microservice. The Order microservice will have two key deployable parts, `order.jar` developed using Spring Boot and its own database—**Order DB**. The `order.jar` will encapsulate service listeners, libraries required for execution, service implementation code, and service APIs or endpoints, and Order DB stores all data required for Order service. Smaller microservices only require these core capabilities.

Gartner named this as **inner architecture** and the capabilities outside of this core as **outer architecture**.

In Chapter 3, *Building Microservices with Spring Boot*, we discussed the implementation of core capabilities using Spring Boot.

The core capabilities depicted in the capability model are explained in the following section.

Service listeners and libraries

Service listeners are endpoint listeners for accepting service requests coming to the microservice. HTTP and message listeners, such as AMQP, JMS, and more are used as service listeners in most of the cases. If microservices are enabled for HTTP endpoint, then the HTTP listener will be embedded within the microservices, thereby eliminating the need to have any external application server. This HTTP listener will be started at the time of the application startup. An example of embedded HTTP-based service implementations are Spring Boot services.

If the microservice is based on asynchronous communication, then, instead of an HTTP listener, a message listener will be started. This needs a reliable messaging system capable of handling large-scale messages, such as Kafka, Rabbit MQ, and more. Messaging endpoints can also be used for request-response scenarios using one of the reactive client libraries, such as RxJava.

Optionally, other protocols can also be considered for specific scenarios. There may not be any listeners if the microservice is a scheduled service.

Storage capability

The microservices will have some kind of storage mechanism to store state or transactional data pertaining to the business capability. If there is a storage required, then it will be dedicated to a microservice. However, storage is optional for microservices. There may be scenarios in which microservice is just a stateless compute service.

Depending on the capabilities that are implemented, the storage could be either a physical storage RDBMS such as MySQL, NoSQL such as Hadoop, Cassandra, Neo4J, Elasticsearch, and more; an in-memory store cache or in-memory data grid such as Ehcache, Hazelcast, Infinispan, and more; or even in-memory databases such as solidDB, TimesTen, and more.

Service implementation

This is the core of microservices, where the business logic is implemented. This could be implemented in any applicable language, such as Java, Scala, Conjure, Erlang, and so on. All required business logic to fulfill the function will be embedded within the microservice itself. Just like with normal application designs, a general modular, layered architecture is recommended for service implementations. This service implementation will provide concrete service endpoint interfaces.

As a best practice, microservice implementations may send out state changes to the external world without really worrying about the targeted consumers of these events. They could be consumed by other microservices, supporting services such as audit by replication or external applications. This will allow other microservices and applications to respond to state changes.

Service endpoint

Service endpoint refers to the APIs exposed by microservices for external consumption. These could be synchronous or asynchronous endpoints.

Synchronous endpoints are generally REST/JSON, but any other protocols, such as Avro, Thrift, or protocol buffers can also be used. Asynchronous endpoints will be through message listeners, such as Spring AMQP, Spring Cloud Streams, and more; backed by a reliable messaging solution, such as Rabbit MQ; or any other messaging servers or other messaging style implementations, such as Zero MQ.

Infrastructure capabilities

Certain infrastructure capabilities are required for a successful deployment and managing large-scale microservices. When deploying microservices at scale, not having proper infrastructure capabilities can be challenging and lead to failures.

In some cases, **Platform as a Service (PaaS)** vendors such as Red Hat OpenShift offer all these capabilities out of the box.

We will discuss infrastructure capabilities in Chapter 9, *Containerizing Microservices with Docker*.

Cloud

Microservices implementation will be difficult in a traditional data center environment with long lead time to provision infrastructures. Even a large number of infrastructure dedicated per microservice may not be very cost effective. Managing them internally in a data center may increase the cost of ownership and operations. An **Infrastructure as a Service (IaaS)** cloud-like infrastructure will be better for microservice deployment.

Microservices require a supporting elastic cloud-like infrastructure that can automatically provision VMs or containers. The automation will also take care of elastically scaling up by adding containers or VMs on demand and scale down when the load falls below threshold.

AWS, Azure, IBM Bluemix, or private cloud (off-premise or on-premise) are candidates for microservice deployments.

Container runtime

When there are many microservices, deploying them on large physical machines is not cost effective, and is also hard to manage. With physical machines, it is also hard to handle automatic fault tolerance of microservices.

Virtualization is adopted by many organizations because of its ability to provide optimal use of physical resources, and it provides resource isolation. It also reduces the overheads in managing large physical infrastructure components. VMWare, Citrix, and more provide virtual machine technologies.

Virtual machines are still heavy and resource intensive, and consume start-up time. Containers are a next wave of virtual machines that provide further cost saving and isolation that is required for smaller footprint microservices. Docker, Rocket, and LXD are some of the containerized technologies.

Above the host operating system, we need a software that can manage containers such as the starting and stopping containers. In this book, this is referred to as container runtime. An example of this is Docker installed on a Linux environment.

Container orchestration

Once we have a large number of containers or virtual machines, it is hard to manage and maintain them automatically. Container orchestration tools provide a uniform operating environment on top of the container runtime, and share available capacity across multiple containers. Apache Mesos, Rancher, CoreOS, and Kubernetes are examples of popular container orchestration tools. These tools are also called **container scheduler** and, sometimes, **container as service**.

One of the other challenges is manual provisioning and deployments. If a deployment has manual elements, the deployer or operational administrators should know the running topology, manually reroute traffic, and then deploy the application one by one until all services are upgraded. With many server instances running, this could lead to significant operational overheads. Moreover, chances of errors will be high in this manual approach.

Container orchestration tools generally helps us to automatically deploy applications, adjust traffic flows, replicate a new version to all instances, and gracefully fade out older versions. In a large deployment environment with many microservices, we need container orchestration tools to avoid overheads.

Container orchestration tools also helps us manage application life cycle activities, including application availability and constraints-based deployment, such as deploying across data centers, ensuring minimum number of instances to be up and running, and so on. Kubernetes supports this capability out of the box, while Mesos requires frameworks such as Marathon to address this capability. **Data Centre Operating System (DCOS)** from Mesosphere combines both Mesos and Marathon out of the box.

Supporting capabilities

Supporting capabilities are not directly linked to microservices, but these are essential for large-scale microservices development. These services will have a dependency on the production runtime of microservices.

Service gateway

The Service gateway or API gateway provides a level of indirection by either proxying service endpoints or composing multiple service endpoints. The API gateway is also useful for policy enforcements and routing. The API gateway can also be used for real-time load balancing in some cases.

There are many API gateways available in the market. Spring Cloud Zuul, Mashery, Apigee, Kong, WSO2, and 3scale are some examples of the API gateway providers.

We will discuss the API gateway using Spring Cloud in [Chapter 7, Scale Microservices with Spring Cloud Components](#).

Software-defined load balancer

The load balancer should be smart enough to understand the changes in the deployment topology and respond accordingly. This moves away from the traditional approach of configuring static IP addresses, domain aliases, or cluster addresses in the load balancer. When new servers are added to the environment, it should automatically detect this and include them in the logical cluster by avoiding any manual interactions. Similarly, if the service instance is unavailable, it should take it out from the load balancer.

A combination of Ribbon, Eureka, and Zuul provide this capability in Spring Cloud Netflix. Alternately, container orchestration tools have an out-of-the-box capability to do load balancers. DCOS has **Marathon Load Balancer (marathon-lb)** for this purpose.

We will discuss Software-Defined Load balancer using Spring Cloud in [Chapter 7, Scale Microservices with Spring Cloud Components](#).

Central log management

Log files are a good piece of information for analysis and debugging. Since each microservice is deployed independently, they emit separate logs, maybe to a local disk. This will result in fragmented logs. When we scale services across multiple machines, each service instance could produce separate log files. This makes it extremely difficult to debug and understand the behavior of the services through log mining.

If we consider Order, Delivery, and Notification as three different microservices, we will find no way to correlate a customer transaction that runs across order processing, delivery, and notification.

When implementing microservices, we need a capability to ship logs from each services to a centrally-managed log repository. With this approach, services are not relaying on local disks or local I/Os. A second advantage is that the log files are centrally managed and are available for all sorts of analysis, such as historical, real time, and trending. By introducing a correlation ID, end-to-end transactions can be easily tracked.

We also need a capability to centralize all logs emitted by service instances with correlation IDs so that we can stitch and track end-to-end transitions based on this correlation ID.

We will discuss logging solutions in Chapter 8, *Logging and Monitoring Microservices*.

Service discovery

With many services running on a cloud-like environment, static service resolution is almost impossible. Therefore, large-scale microservices need an auto discovery mechanism to identify where the services are running.

A service registry provides a runtime environment for services to automatically publish their availability at runtime. A registry will be a good source of information to understand the services topology at any point so that consumers can look up for service discovery.

Eureka from Spring Cloud, Zookeeper, and Etcd are some of the service registry tools available. Alternately, container discovery services are available out of the box from container orchestration tools. For example, Mesos-DNS comes out of the box as part of the DCOS distribution.

We will discuss service discovery using Spring Cloud in Chapter 7, *Scale Microservices with Spring Cloud Components*.

Security service

In monolithic applications, application security was part of the application itself. Hence, it was easy to manage. With microservices, security became a more predominant issue, as there are many services, and just one service cannot hold the security master data. Distributed microservices ecosystem require a central server to manage service security. This includes service authentication and token services.

Spring Security and Spring Security OAuth are good candidates to build this capability. Single sign-on solutions provided by Microsoft, Ping, and Okta are other enterprise-grade security solutions that can be integrated well with microservices.

In Chapter 3, *Building Microservices with Spring Boot*, we discussed this capability using Spring Boot.

Service configuration

When many services, especially those running on different servers, are deployed using automated tools, it is hard to keep the application configuration static, as we practiced in the monolithic application development.

With microservices, all service configurations should be externalized, as proposed in the Twelve-Factor App. A central service for all configurations could be a good choice. Spring Cloud Config server and Archaius are out-of-the-box configuration servers. Alternately, Spring Boot profiles can be used to manage configurations for small-scale microservices if the configuration changes are not dynamic in nature.

We will discuss service configuration using Spring Cloud in Chapter 7, *Scale Microservices with Spring Cloud Components*.

Ops monitoring

With a large number of microservices, and with multiple versions and service instances, it would be difficult to find out which service is running on which server, what the health of these services is, the service dependencies, and so on. This was much easier with the monolithic applications that are tagged against a specific or a fixed set of servers.

Apart from understanding the deployment topology and health, it also poses a challenge in identifying service behaviors, debugging, and identifying hotspots. Strong monitoring capabilities are required to manage such an infrastructure.

Spring Cloud Netflix Turbine, Hystrix dashboard, and more provide service-level information. End-to-end monitoring tools, such as AppDynamic, NewRelic, and Dynatrace, and other tools, such as Statd, Sensu, and Simian Viz, could add value in microservices monitoring. Tools such as Datadog help us manage infrastructure efficiently.

We will discuss monitoring solutions in Chapter 8, *Logging and Monitoring Microservices*.

Dependency management

Dependency management is one of the key issues in large microservices deployments. How do we identify and reduce the impact of a change? How do we know whether all dependent services are up and running? How will the service behave if one of the dependent services is not available?

Too many dependencies could raise challenges in microservices. Four important design aspects are stated as follows:

- Reduce the dependencies by properly designing service boundaries.
- Reduce impacts by designing dependencies that are as loosely coupled as possible. Also, design service interactions through asynchronous communication styles.
- Dependency issues need to be tackled using patterns such as circuit breakers.
- Monitor dependencies using visual dependency graphs.

With many microservices, the number of **Configurable Items (CIs)** will be too high, and the number of servers in which these CIs are deployed might also be unpredictable. This will make it extremely difficult to manage data in a traditional **Configuration Management Database (CMDB)**. In many cases, it is more useful to dynamically discover the current running topology than a statically configured CMDB style deployment topology. A graph-based CMDB is more obvious to manage these scenarios.

There are many tools available in the market, but a combination of tools can address the dependency issue. Some of the Ops monitoring or APM tools such as AppDynamic are useful. Cloud Craft, Light Mesh, and Simian Viz are some of the other tools.

We will discuss dependency management solutions in [Chapter 8, Logging and Monitoring Microservices](#).

Data lake

Microservices abstract their own local transactional store, which is used for their own transactional purposes. The type of store and the data structure will be optimized for the services offered by the microservice.

For example, if we want to develop a customer relationship graph, we may use a graph database such as Neo4J, OrientDB, and more. A predictive text search to find out customers based on any related information, such as passport number, address, email, phone number, and more, could be best realized using an indexed search database, such as Elasticsearch or Solr.

This will place us into a unique situation of fragmenting data into heterogeneous data islands. For example, Customer, Loyalty Points, Reservations, and more are different microservices, and, hence, use different databases. What if we want to do a near real-time analysis of all high-value customers by combining data from all three data stores? With a monolithic application, this was easy because all the data was present in a single database.

In order to satisfy this requirement, a data warehouse or a data lake is required. Traditional data warehouses, such as Oracle, Teradata, and more, are used primarily for batch reporting. However, with NoSQL databases, such as Hadoop, and micro-batching techniques, near real-time analytics is possible with the concept of data lakes. Unlike the traditional warehouses that are purposely built for batch reporting, data lakes store raw data without assuming how the data will be used. Now, the question really is how to port data from microservices into data lakes.

Data porting from microservices to a data lake or a data warehouse can be done in many ways. Traditional ETL could be one of the options. Since we are allowing backdoor entry with ETL and breaking the abstraction, this is not considered as an effective way for data movement. A better approach is to send events from microservices as and when they occur. For example, customer registration, customer update event, and so on. Data ingestion tools will consume these events and propagate the state change to the data lake appropriately. The data ingestion tools are highly-scalable platforms, such as Spring Cloud Data Flow, Kafka, Flink, Flume, and so on.

We will discuss monitoring solutions in Chapter 8, *Logging and Monitoring Microservices*.

Reliable messaging

It is recommended to use a reactive style for microservices development. This will help in decoupling microservices, and, therefore, create better scalability. In a reactive system, we need a reliable, high-available messaging infrastructure service.

Messaging servers such as RabbitMQ, ActiveMQ, and Kafka are popular choices. IBM MQ and TIBCO EMS are other commercially viable enterprise-scale messaging platforms. Cloud messaging or messaging as a service such as [Iron.io](#) is a popular choice for internet-scale messaging.

Process and governance capabilities

The last piece in the puzzle is the process and governance capabilities that are required for microservices. These are processes, tools, and guidelines around microservices implementations.

DevOps

One of the biggest challenges in microservices implementation is the organization culture. To harness the speed of delivery of microservices, the organization should adopt agile development processes, continuous integration, automated QA checks, automated delivery pipelines, automated deployments, and automatic infrastructure provisioning.

Organizations following a waterfall development or heavyweight release management processes with infrequent release cycles are a challenge for microservices development.

DevOps is a key in successful implementations. It complements microservices development by supporting agile development, high-velocity delivery, automation, and better change management.

We will discuss these capabilities in Chapter 11, *Microservice Development Life Cycle*.

Automation tools

Automation tools used in agile development, continuous integration, continuous delivery, and continuous deployment are essential for the successful delivery of microservices. Without automation, managing delivery of many smaller microservices will be a nightmare for any enterprise. Since the velocity of changes in each microservices is different, it may be worth considering different pipelines for each microservices.

Test automation is extremely important in the microservices delivery. In general, microservices also poses a challenge for the testability of services. In order to achieve a full-service functionality, one service may rely on another service, and that in turn, may rely on another service--either synchronously or asynchronously. The issue is how do we test an end-to-end service to evaluate its behavior. The dependent services may or may not be available at the time of testing.

Service virtualization and **service mocking** is one of the techniques used to test services without actual dependencies. In testing environments, when the services are not available, mock services can simulate behavior of the actual service. The microservices ecosystem needs service virtualization capabilities. However, this may not give us full confidence, as there may be many corner cases that mock services do not simulate, especially when there are deep dependencies.

Another approach is to use a consumer-driven contract. The translated integration test cases can cover more or less all corner cases of the service invocation.

A lot of emphasis is required in automated functional, real-user testing, synthetic testing, integration, release, and performance testing. Test automation and continuous delivery approaches such as AB Testing, Future Flags, Canary Testing, Blue-Green deployments, and Red-Black deployments, all reduce the risks of production releases.

In-production destructive testing is also a useful technique in microservice development. Netflix uses Simian Army for anti-fragile testing. Matured services need consistent challenges to see the reliability of the services and how good fallback mechanisms are. Simian Army components create various error scenarios to explore the behavior of the system under the failure scenarios.

We will discuss these capabilities in Chapter 11, *Microservice Development Life Cycle*.

Container registry

A microservices registry is where the versioned binaries of microservices are placed. These could be a simple artifactory repository or truly container repository, such as a Docker registry. Generally, a Docker registry stores base images as well as application images built using those base images. Automation tools will be integrated with the Docker registry as part of the development and delivery pipeline to upload and download images.

It is arguable that container registry is part of the supporting capability or process and governance capability. In this reference model, it is added as a part of the process and governance mainly because it is more of a part of automation tools and doesn't play any role beyond deployments. Registry is also fundamental for the governance. Many organizations set policies around base container images to avoid security-related issues.

Docker Hub, Google Container Repository, Core OS Quay, and Amazon EC2 container registry are some of the examples. Registry can be either private or public, based on the organizations the cyber security policies.

We will discuss Docker registry in Chapter 9, *Containerizing Microservices with Docker*.

Microservice documentation

Microservices impose decentralized governance, and this is quite in contrast with the traditional SOA governance. Organizations may find it hard to come up with this change, and that could negatively impact the microservices development.

There are a number of challenges that come with a decentralized governance model. How do we understand who is consuming a service? How do we ensure service reuse? How do we define which services are available in the organization? How do we ensure enforcement of enterprise policies?

One of the most important considerations is to have a place where all stakeholders can see all the services, their documentations, contracts, and service-level agreements. Especially in a distributed agile development environment where scrum teams are empowered to do their designs, it is important to understand what endpoints are available, what they offer, and how to access them, hosted in a central repository.

A good API repository should do the following:

- Expose repository via a web browser
- Provide easy ways to navigate APIs
- Be nicely organized
- Have the ability to invoke and test those endpoints with samples

Swagger, RAML, or API blueprint are helpful in achieving good microservices documentation. Swagger is popular and commonly used by many enterprises.



Spotify's service documentation in their public developer portal is a good example of microservices documentation:
<https://developer.spotify.com/web-api/endpoint-reference/>

We discussed the microservice documentation using Swagger in Chapter 3, *Building Microservice with Spring Boot*.

Reference architecture and libraries

Decentralized governance also poses challenges in developing microservices in silos with different patterns, tools, and technologies. This will limit organizations that are extended to deploy cost-effective solutions and also reusing services.

The first and foremost factor to consider in a decentralized governance model is to have a set of standards, reference models, best practices, and guidelines on how to implement better services. These should be available to the organization in the form of standard libraries, tools, and techniques. This ensures that the services developed are top quality, and that they are developed in a consistent manner.

The reference architecture provides a blueprint at the organization level to ensure that services are developed according to certain standards and guidelines in a consistent manner. Many of these could then be translated to a number of reusable libraries that enforce service development philosophies.

Standardization on tools helps enterprises to avoid different flavors of microservices implementations which are not interoperable, for example, different teams using different tools for microservice documentation, different image registries, or different container orchestration tools.

This reference architecture and libraries, together with a strong commitment to documentation, is essential when dealing with decentralized microservices governance.

We will discuss these capabilities in Chapter 11, *Microservice Development Life Cycle*.

Microservices maturity model

Microservice adoption needs some careful thoughts. A quick maturity assessment will be helpful to understand the maturity of the organization and some of the challenges the organization can expect.

The maturity model in the following diagram is derived from the capability model discussed earlier in this chapter:

	Level 0 Traditional	Level 1 Basic	Level 2 Intermediate	Level 3 Advanced
Application	Monolithic	Service Oriented Integrations	Service Oriented Applications	API Centric
Database	One Size Fit All Enterprise DB	Enterprise DB + No SQLs and Light databases	Polyglot, DBaaS	Matured Data Lake / Near Realtime Analytics
Infrastructure	Physical Machines	Virtualization	Cloud	Containers
Monitoring	Infrastrucure	App & Infra Monitoring	APMs	APM & Central Log Management
Process	Waterfall	Agile and CI	CI & CD	DevOps

The 4*5 maturity model is simple enough for a quick self-evaluation. Four levels of maturities are mapped against five characteristics of application development-- **Application, Database, Infrastructure, Monitoring, and Processes**.

Level 0 - Traditional

Characteristics of the Traditional maturity level are explained as follows:

- Organizations still develop applications in a monolithic approach. There may be internal modularizations using subsystem designs, but packaged as a monolithic WAR. Use of proprietary service interfaces instead of RESTful service.
- Organizations use the *one size fits all* database model based on an enterprise standard and license model, irrespective of application types or application size.
- The infrastructure will be primarily based on physical machines with no virtualization implemented.
- The infrastructure monitoring may exist, but only as a limited application-level monitoring, for example, the application's URL monitoring.
- These organizations will be primarily based on waterfall development methods with long and fat release cycles.

These characteristics are not fully suitable for microservice development. Such organizations may struggle when attempting large-scale microservice developments. The recommended approach is to start small and adopt in small scale. A big stride needs careful planning and adoption of all related capabilities.

Level 1 - Basic

The characteristics of the Basic maturity level are explained as follows:

- Organizations still develop applications in a monolithic model, but uses Service-Oriented Integration for application-to-application communications
- While predominantly these organizations use one size fit for all database models, there may be some footprints of NoSQL and other lightweight databases
- The infrastructure will be primarily based on virtual machines, with no cloud adoption yet
- The infrastructure monitoring may be sophisticated, which includes somewhat matured application layer monitoring
- These organizations use agile development methods with automated tools for continuous integration

These characteristics are still not fully suitable for microservice development. Such organizations will face issues around optimal infrastructure usage and faster application delivery. The recommended approach for organizations at this level is to identify candidates for microservices and carefully approach them with some tactical decisions on infrastructure to start with, such as sharing a single instance of database. The risk level is less compared to Level 0, Traditional.

Level 2 - Intermediate

The characteristics of the Intermediate maturity level are explained as follows:

- Organizations develop SOA-based applications with a strong focus on service-based developments. For optimization purposes, they may still use proprietary protocols at application level.
- These organizations use polyglot persistence as a first class citizen. These organizations will have the culture of choosing the right databases for the right purpose without much worry about the economies of the scale.

- The infrastructure will be primarily based on cloud--either public or private.
- Organizations use both infrastructure monitoring as well as APM tools for end-to-end application monitoring.
- These organizations use agile development methods with automated tools for continuous integration and delivery.

Such organizations are more or less just one step away from full-scale microservices developments. Microservices is their natural progression as their next phase of architecture adoption. The risk is very minimal at this level.

Level 3 - Advanced

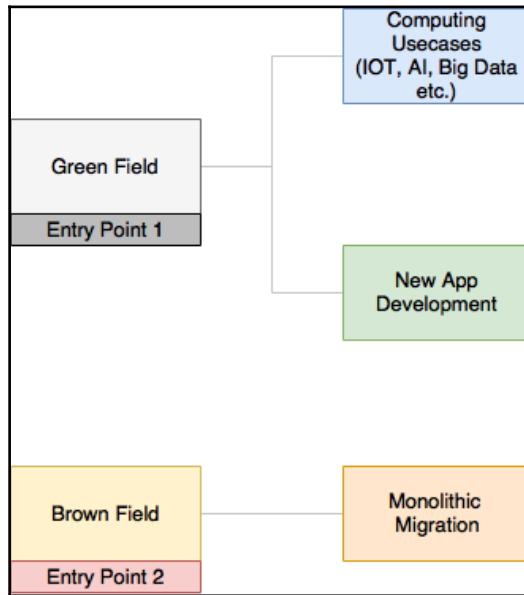
The characteristics of the Advanced maturity level are explained as follows:

- Organizations use APIs as first-class citizens for development. They use an API-first design philosophy.
- These organizations use polyglot persistence as first class citizen. Also, these organizations are matured in terms of data lake and near real-time analytics.
- The infrastructure will be primarily based on cloud, but also use containers and container-orchestration tools.
- Organizations use both infrastructure monitoring as well as APM tools for end-to-end application monitoring, including synthetic and real-user monitoring. They also use the centralized log management solutions.
- These organizations use full DevOps philosophies for application/product developments.

Such organizations may already be using some form of microservices, and they are ready to quickly move to large-scale microservice developments.

Entry points for adoption

When adopting microservices, organizations generally use one of the two entry points mentioned in the following diagram:



The first entry point is termed as the **Green Field** approach. In this approach, organizations will use microservice for developing new capabilities. There are a couple of use cases for Green Field development:

- Development of computing services, such as IoTs, AI algorithms, Big Data processing, and more
- New ground-up application developments

The second entry point is termed as the **Brown Field** approach. In this case, organizations will use microservices for monolithic migrations.

Summary

In this chapter, we established a technology and tool agnostic capability model for microservice based on best practices, common patterns, and design guidelines, inspired from successful microservices implementations across the industry. This will be useful for organizations thinking about the microservices journey to understand the different areas to be considered before attempting microservice adoption.

We expanded each of these capabilities in the capability model and learned their importance in microservices implementations. Along side, we also learned various technology solutions available to support these capabilities. Finally, we explored a maturity model for microservices adoption.

Next, we will take a real-world problem and model using the microservices architecture to see how to translate our learnings into practice.

18

Scale Microservices with Spring Cloud Components

In order to manage Internet-scale microservices, one requires more capabilities than what is offered by the Spring Boot framework. The Spring Cloud project has a suite of purpose-built components to achieve these additional capabilities effortlessly.

This chapter will provide a deep insight into the various components of the Spring Cloud project, such as Eureka, Zuul, Ribbon, and Spring Config, by positioning them against the microservices capability model discussed in [Chapter 4, Applying Microservices Concepts](#). This chapter will demonstrate how the Spring Cloud components help to scale the BrownField Airline's PSS microservices system, developed in the previous chapter.

At the end of this chapter, we will have learned about the following:

- The **Spring Config Server** for externalizing configuration
- The Eureka Server for service registration and discovery
- Understanding the relevance of Zuul as a service proxy and gateway
- The implementation of automatic microservice registration and service discovery
- Spring Cloud messaging for asynchronous reactive microservice composition

What is Spring Cloud?

Add a link to Netflix OSS. The Spring Cloud project is an umbrella project from the Spring team, which implements a set of common patterns required by distributed systems as a set of easy-to-use Java Spring libraries. Despite its name, Spring Cloud by itself is not a cloud solution. Rather, it provides a number of capabilities, which are essential when developing applications targeting cloud deployments that adhere to the Twelve-Factor Application principles. By using Spring Cloud, developers just need to focus on building business capabilities using Spring Boot, and leverage the distributed, fault-tolerant, and self-healing capabilities available out-of-the-box from Spring Cloud.

The Spring Cloud solutions are agnostic to the deployment environment and can be developed and deployed on a desktop PC or in an elastic cloud. The cloud-ready solutions, which are developed using Spring Cloud, are also agnostic, and portable across many cloud providers, such as Cloud Foundry, AWS, Heroku, and others. When not using Spring Cloud, developers will end up using services natively provided by the cloud vendors, resulting in deep coupling with the PaaS providers. An alternate option for developers is to write quite a lot of boilerplate code to build these services. Spring Cloud also provides simple, easy-to-use Spring-friendly APIs, which abstract the cloud provider's service APIs, such as those APIs coming with the AWS Notification service.

Built on Spring's *convention over configuration* approach, Spring Cloud defaults all configurations, and helps developers to a quick start. Spring Cloud also hides complexities, and provides simple declarative configurations to build systems. The smaller footprints of the Spring Cloud components make it developer-friendly, and also make it easy to develop cloud-native applications.

Spring Cloud offers many choices of solutions for developers based on their requirements. For example, the service registry can be implemented using popular options such as Eureka, Zookeeper, or Consul. The components of Spring Cloud are fairly decoupled, hence, developers get the flexibility to pick and choose what is required.



What is the difference between Spring Cloud and Cloud Foundry? Spring Cloud is a developer kit for developing Internet-scale Spring Boot applications, whereas, Cloud Foundry is an open source Platform as a Service for building, deploying, and scaling applications.

Spring Cloud releases

The Spring Cloud project is an overarching Spring project, which includes a combination of different components. The versions of these components are defined in the *spring-cloud-starter-parent* BOM.



In this book, we are relying on the **Dalston SR1** version of the Spring Cloud. Dalston does not support Spring Boot 2.0.0 and Spring Framework 5. Spring Cloud Finchley, planned for the end of this year, is expected to support Spring Boot 2.0.0. Hence, examples in the previous chapters need to downgrade to the Spring Boot 1.5.2.RELEASE version.

Add the following dependency in `pom.xml` to use Spring Cloud Dalston dependency:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-dependencies</artifactId>
    <version>Dalston.SR1</version>
    <type>pom</type>
    <scope>import</scope>
</dependency>
```

Setting up the environment for the BrownField PSS

In this chapter, we will amend the BrownField's PSS microservices developed in Chapter 6, *Microservices Evolution - A Case Study*, using the Spring Cloud capabilities. We will also examine how to make these services enterprise grade using the Spring Cloud components.

In order to prepare the environment for this chapter, import and rename (`chapter6.*` to `chapter7.*`) projects into a new STS workspace.



The full source code of this chapter is available under the `chapter7` project in the code files under <https://github.com/rajeshrv/Spring5Microservice>.

Spring Cloud Config

Simplify this section. The Spring Cloud Config Server is an externalized configuration server in which applications and services can deposit, access, and manage all runtime configuration properties. The Spring Config Server also supports version control of the configuration properties.

In earlier examples with Spring Boot, all configuration parameters were read from a property file packaged inside the project, either `application.properties` or `application.yaml`. This approach is good, since all properties are moved out of code to a property file. However, when microservices are moved from one environment to another, these properties need to undergo changes, which require application rebuild. This is in violation of one of the Twelve-Factor Application principles, which advocates one time build and moving the binaries across environments.

A better approach is to use the concept of profiles. Profiles, as discussed in Chapter 3, *Building Microservices with Spring Boot*, is used to partition different properties for different environments. The profile specific configuration will be named as `application-{profile}.properties`. For example, `application-development.properties` represents a property file targeted for the development environment.

However, the disadvantage of this approach is that the configurations are statically packaged along with the application. Any changes in the configuration properties require the application to be rebuilt.

There are alternate ways to externalize configuration properties from the application deployment package. Configurable properties can also be read from an external source in a number of ways, which are as follows:

- From an external JNDI server using the JNDI namespace (`java:comp/env`)
- Using the Java system properties (`System.getProperties()`), or by using the `-D` command-line option
- Using the `PropertySource` configuration

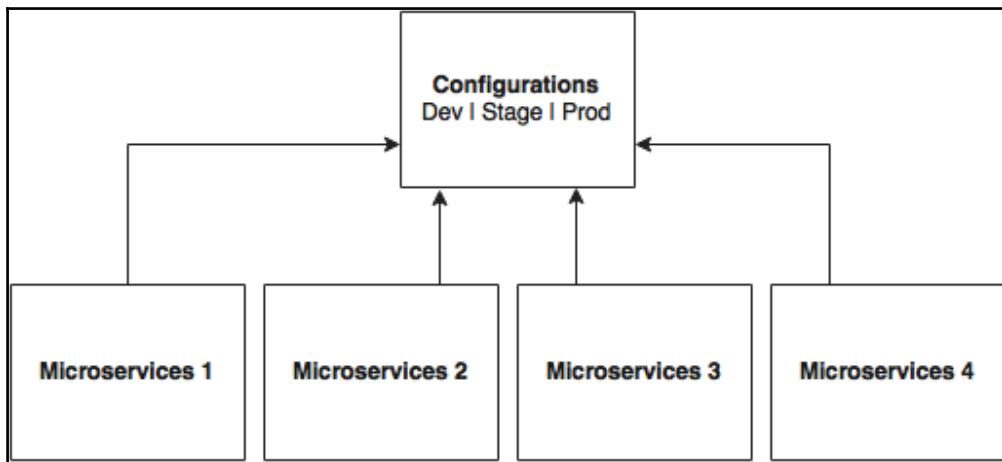
```
@PropertySource("file:${CONF_DIR}/application.properties")
public class ApplicationConfig {
```

- Using a command-line parameter pointing a file to an external location

```
java -jar myproject.jar --spring.config.location=<file location>
```

JNDI operations are expensive, lack flexibility, have difficulties in replication, and are not version controlled. `System.properties` is not flexible enough for large-scale deployments. The last two options rely on a local or a shared filesystem mounted on the server.

For large-scale deployments, a simple, yet powerful, centralized configuration management solution is required.



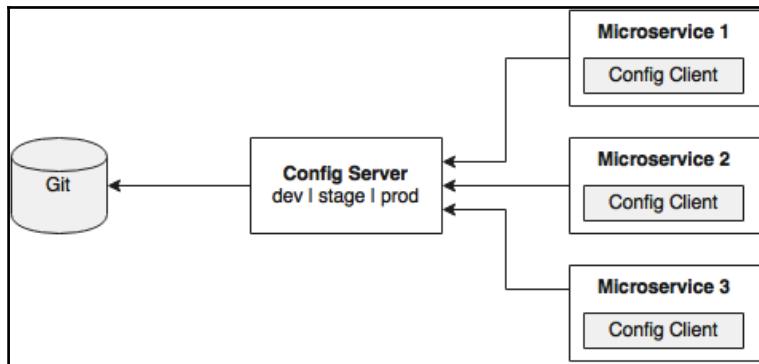
As shown in the preceding diagram, all microservices point to a central server to get the required configuration parameters. The microservices then locally cache these parameters to improve performance. The Config Server propagates the configuration state changes to all subscribed microservices so that the local cache's state can be updated with the latest changes. The Config Server also uses profiles to resolve values specific to an environment.

As shown in the next diagram, there are multiple options available under the Spring Cloud project for building the configuration server. Config Server, Zookeeper Configuration, and Consul Configuration are available as options. However, this chapter will only focus on the Spring Config Server implementation:



The Spring Config Server stores properties in a version-controlled repository such as Git or SVN. The Git repository can be local or remote. A highly available remote Git server is preferred for large-scale distributed microservices deployments.

The Spring Cloud Config Server architecture is shown in the following diagram:

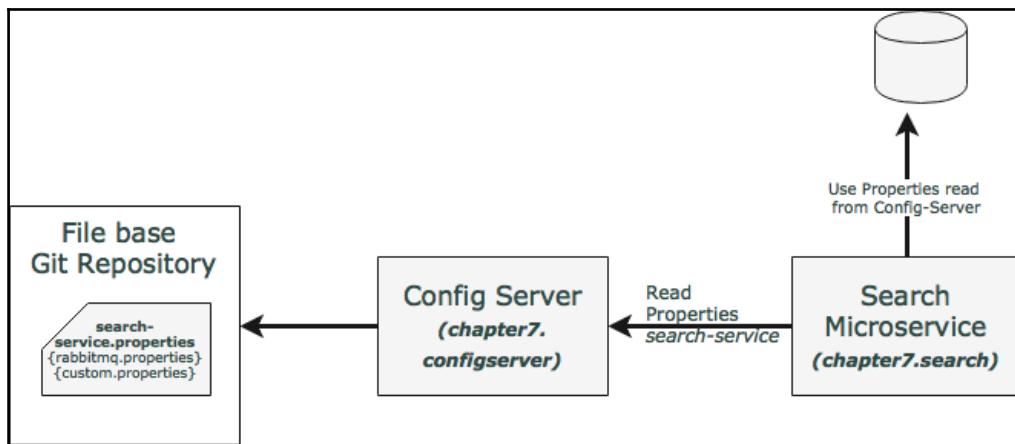


As shown in the diagram, the **Config Client** embedded in the Spring Boot microservices does a configuration lookup from a central configuration server using simple declarative mechanisms and stores properties into the Spring environment. The configuration properties can be application-level configurations such as trade limit per day, or infrastructure-related configurations such as server URLs, credential and so on.

Unlike Spring Boot, Spring Cloud uses a bootstrap context, which is a parent context of the main application. Bootstrap context is responsible for loading configuration properties from the **Config Server**. The bootstrap context looks for `bootstrap.yaml` or `bootstrap.properties` for loading the initial configuration properties. To make this work in a Spring Boot application, rename the `application.*` file as `bootstrap.*`.

Building microservices with Config Server

The next few sections demonstrate how to use the Config Server in a real-world scenario. In order to do this, we will modify our search microservice (`chapter7.search`) to use the Config Server. The following diagram depicts the scenario:

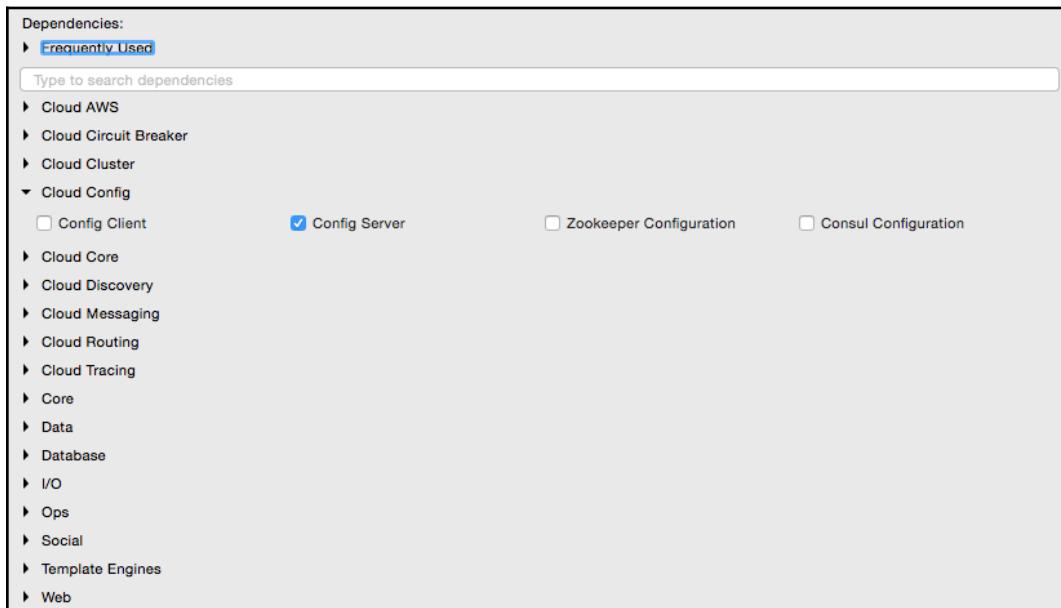


In this example, the search service will read the **Config Server** at startup by passing the service name. In this case, the service name of the search service will be **search-service**. The properties configured for the **search-service** include the RabbitMQ properties as well as a custom property.

Setting up the Config Server

The following steps need to be followed to create a new Config Server using STS:

1. Create a new Spring Starter project and select **Config Server** and **Actuator**, as shown in this screenshot:



2. Set up a Git repository. This could be done by pointing to a remote Git configuration repository, such as <https://github.com/spring-cloud-samples/config-repo>. This URL is an indicative one, a Git repository used by the Spring Cloud examples. We will have to use our own Git repository instead.
3. Alternatively, a local-file-system-based Git repository can be used. In a real production scenario, an external Git is recommended. The Config Server in this chapter will use a local-file-system-based Git repository for demonstration purposes.
4. Use the following set of commands to set up a local Git repository:

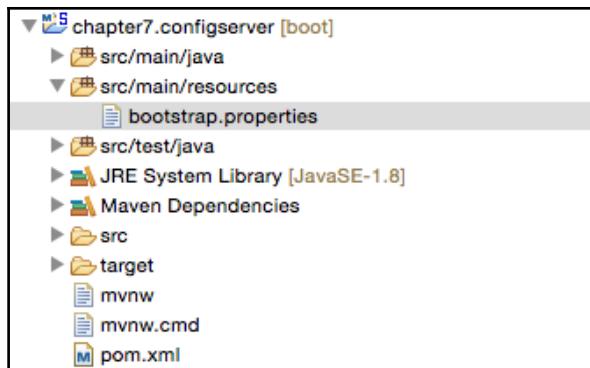
```
$ cd $HOME  
$ mkdir config-repo  
$ cd config-repo  
$ git init .  
$ echo message : helloworld > application.properties  
$ git add -A .  
$ git commit -m "Added sample application.properties"
```

This will create a new Git repository on the local filesystem. A property file, named `application.properties`, with a property message and value `helloworld` is also created.



`application.properties` is created for demonstration purposes. We will change this in subsequent sections.

5. The next step is to change the configuration in the Config Server to use the Git repository created in the previous step. In order to do this, rename `application.properties` as `bootstrap.properties`:



6. Edit the contents of the new `bootstrap.properties` to match the following:

```
server.port=8888
spring.cloud.config.server.git.uri:
  file://${user.home}/config-repo
```



Port 8888 is the default port for the Config Server. Even without configuring the `server.port`, the Config Server should bind to 8888. In a Windows environment, an extra / is required in the file URL.

7. Also add `management.security.enabled=false` to disable security validation.
8. Optionally, rename the default package of the autogenerated `Application.java` from `com.example` to `com.brownfield.configserver`. Add `@EnableConfigServer` in `Application.java`.

```
@EnableConfigServer
@SpringBootApplication
public class ConfigserverApplication {
```

9. Run the Config Server by right-clicking the project, and run as Spring Boot App.
10. Check `curl http://localhost:8888/env` to see whether the server is running. If everything is fine, this will list all environment configurations. Note that `/env` is an Actuator endpoint.
11. Check `http://localhost:8888/application/default/master` to see the properties specific to `application.properties`, which was added in the earlier step. The browser will display properties configured in the `application.properties`. The browser should display content similar to this:

```
{"name": "application", "profiles": ["default"], "label": "master",
  "version": "6046fd2ff4fa09d3843767660d963866ffcc7d28",
  "propertySources": [{"name": "file:///Users/rvlabs
    /config-repo /application.properties", "source":
  {"message": "helloworld"}}]}}
```

Understanding the Config Server URL

In the previous section, we used

`http://localhost:8888/application/default/master` to explore the properties. How do we interpret the given URL?

The first element in the URL is the application name. In the last example, the application name should be `application`. The application name is a logical name given to the application, using the `spring.application.name` property in the `bootstrap.properties` of the Spring Boot application. Each application must have a unique name. The Config Server will use the name to resolve and pick up appropriate properties from the Config Server repository. The application name is also sometimes referred to as service ID. If there is an application with the name `myapp`, then there should a `myapp.properties` in the configuration repository to store all properties related to that application.

The second part of the URL represents the profile. There can be more than one profile configured within the repository for an application. The profiles can be used in various scenarios. The two common scenarios are to segregate different environments such as Dev, Test, Stage, Prod, and so on, or to segregate server configurations such as primary, secondary, and others. The first one represents different environments of an application, whereas the second one represents different servers where an application is deployed.

The profile names are logical names, which will be used for matching the filename in the repository. The default profile is named as `default`. To configure properties for different environments, we have to configure different files, as shown next. In this example, the first file is for the development environment, whereas the second is for the production environment:

```
application-development.properties  
application-production.properties
```

These are accessible using the following URLs:

`http://localhost:8888/application/development`

`http://localhost:8888/application/production`

The last part of the URL is the label, and is named as `master` by default. The label is an optional Git label, which can be used if required.

In short, the URL is based on this pattern:

```
http://localhost:8888/{name}/{profile}/{label}
```

The configuration can also be accessed by ignoring the profile. In the given example, all the following three URLs point to the same configuration:

```
http://localhost:8888/application/default
```

```
http://localhost:8888/application/master
```

```
http://localhost:8888/application/default/master
```

There is an option to have different Git repositories for different profiles. This makes sense for production systems, since the access to different repositories could be different.

Accessing the Config Server from clients

In the previous section, a Config Server is set up and accessed using a web browser. In this section, the Search microservice will be modified to use the Config Server. The Search microservice will act as a Config Client.

Follow the steps listed next to use the Config Server instead of reading properties from the application.properties file:

1. Add the Spring Cloud Config dependency and the Actuator (if Actuator is not already in place) to pom.xml. The Actuator is mandatory for refreshing the configuration properties:

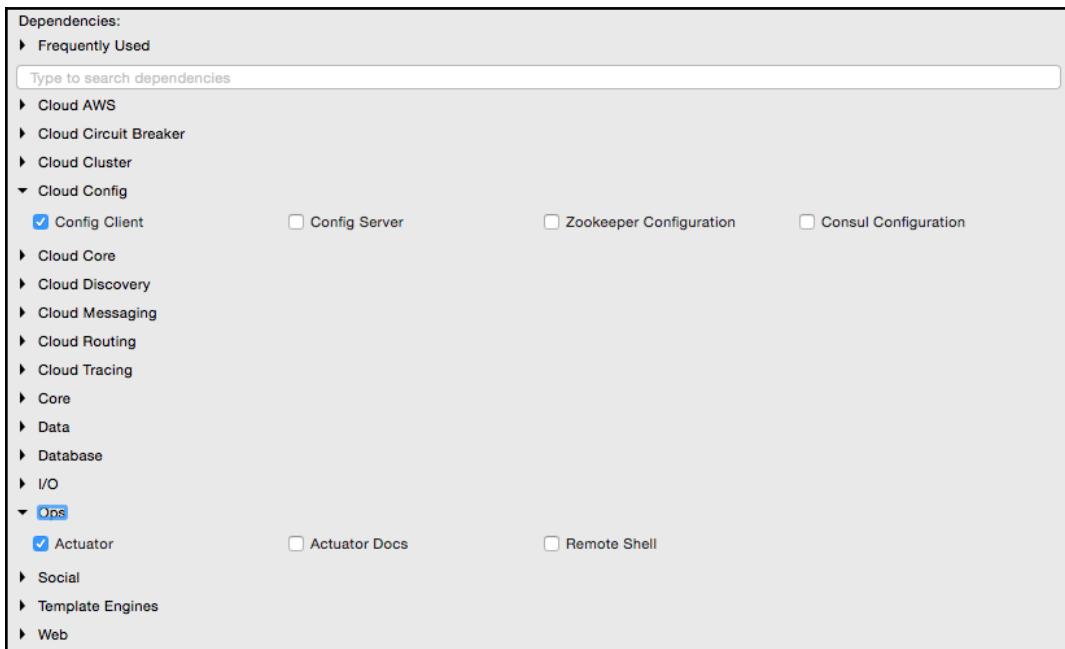
```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
```

2. Since we are modifying the Spring Boot Search microservice from the earlier chapter, we will have to add the following to include the Spring Cloud dependencies. This is not required if the project is created from scratch:

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
```

```
<version>Dalston.BUILD-SNAPSHOT</version>
<type>pom</type>
<scope>import</scope>
</dependency>
</dependencies>
</dependencyManagement>
```

3. The following screenshot shows the Cloud starter library selection screen. If the application is built from the ground up, select the libraries as shown here:



4. Rename the `application.properties` to `bootstrap.properties`, and add an application name and a configuration server URL. The configuration server URL is not mandatory if the Config Server is running on the default port (8888) on the local host.

The new `bootstrap.properties` file will look like this:

```
spring.application.name=search-service
spring.cloud.config.uri=http://localhost:8888
server.port=8090
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
```

```
spring.rabbitmq.username=guest  
spring.rabbitmq.password=guest
```

search-service is a logical name given to the Search microservices. This will be treated as the service ID. The Config Server will look for `search-service.properties` in the repository to resolve the properties.

5. Create a new configuration file for `search-service`. Create a new `search-service.properties` under the `config-repo` folder where the Git repository is created. Note that `search-service` is the service ID given to the Search microservice in the `bootstrap.properties` file. Move service-specific properties from `bootstrap.properties` to the new `search-service.properties`. The following properties will be removed from `bootstrap.properties` and added to `search-service.properties`:

```
spring.rabbitmq.host=localhost  
spring.rabbitmq.port=5672  
spring.rabbitmq.username=guest  
spring.rabbitmq.password=guest
```

6. In order to demonstrate the centralized configuration of properties and propagation of changes, add a new application-specific property to the property file. We will add `originairports.shutdown` to temporarily take out an airport from the search. Users will not get any flights when searching with an airport mentioned in the shutdown list:

```
originairports.shutdown=SEA
```

In this preceding example, we will not return any flights when searching with SEA as origin.

7. Commit this new file into the Git repository by executing the following commands:

```
git add -A .  
git commit -m "adding new configuration"
```

The final `search-service.properties` file should look like this:

```
spring.rabbitmq.host=localhost  
spring.rabbitmq.port=5672  
spring.rabbitmq.username=guest  
spring.rabbitmq.password=guest  
originairports.shutdown:SEA
```

The chapter7.search project's bootstrap.properties should look like the following:

```
spring.application.name=search-service
server.port=8090
spring.cloud.config.uri=http://localhost:8888
```

8. Modify the Search microservice code to use the configured parameter, originairports.shutdown. A RefreshScope annotation has to be added at the class level to allow properties to be refreshed when there is a change. In this case, we are adding a refresh scope to the SearchRestController class.

```
@RefreshScope
```

9. Add the following instance variable as a placeholder for the new property that is just added in the Config Server. The property names in the search-service.properties must match:

```
@Value("${originairports.shutdown}")
private String originAirportShutdownList;
```

10. Change the application code to use this property. This is done by modifying the search method as follows:

```
@RequestMapping(value="/get", method = RequestMethod.POST)
List<Flight> search(@RequestBody SearchQuery query){
    logger.info("Input : "+ query);
    if(Arrays.asList(originAirportShutdownList.split(","))
        .contains(query.getOrigin())){
        logger.info("The origin airport is in shutdown state");
        return new ArrayList<Flight>();
    }
    return searchComponent.search(query);
}
```

The search method is modified to read the parameter originAirportShutdownList and see whether the requested origin is in the shutdown list. If there is a match, instead of proceeding with the actual search, the search method will return an empty flight list.

11. Start the Config Server. Then start the Search microservice. Make sure that the Rabbit MQ server is running.

12. Modify the `chapter7.website` project to match the `bootstrap.properties` content, as follows, to utilize the Config Server:

```
spring.application.name=test-client
server.port=8001
spring.cloud.config.uri=http://localhost:8888
```

13. Change the `CommandLineRunner`'s run method in `Application.java` to query SEA as origin airport.

```
SearchQuery searchQuery = new SearchQuery(
    "SEA", "SFO", "22-JAN-18");
```

14. Run the `chapter7.website` project. `CommandLineRunner` will now return an empty flight list. The following message will be printed in the server:

```
The origin airport is in shutdown state
```

Handling configuration changes

The `/refresh` endpoint will refresh the locally cached configuration properties and reload fresh values from the Config Server.

In order to force reloading the configuration properties, call the `/refresh` endpoint of the microservice. This is actually the Actuator's `refresh` endpoint. The following command will send an empty POST to the `/refresh` endpoint:

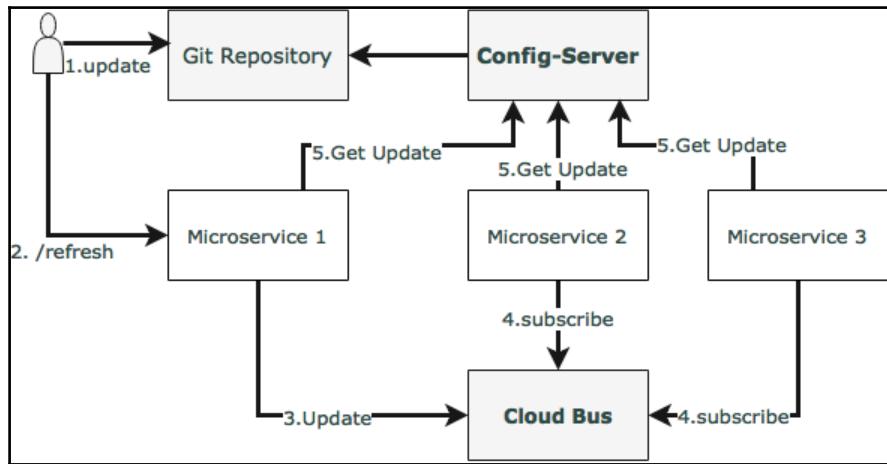
```
curl -d {} localhost:8090/refresh
```

Spring Cloud Bus for propagating configuration changes

With the preceding approach, configuration parameters can be changed without restarting the microservices. This is good when there are only one or two instances of the services running. What happens if there are many instances?

For example, if there are five instances, then we have to hit `/refresh` against each service instance. This is definitely a cumbersome activity.

The following diagram shows the solution using Spring Cloud Bus:



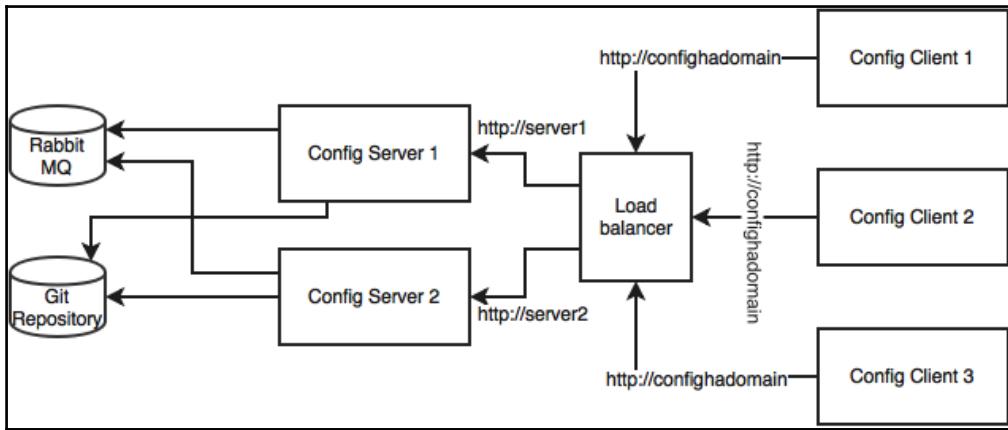
The Spring Cloud Bus provides a mechanism to refresh configurations across multiple instances without knowing how many instances there are, nor their locations. This is particularly handy when there are many service instances of a microservice running, or when there are many microservices of different types running. This is done by connecting all service instances through a single message broker. Each instance subscribes for change events and refreshes its local configuration when required. This refresh is triggered by making a call to any one instance by hitting the `/bus/refresh` endpoint, which then propagates the changes through the cloud bus and the common message broker.

Setting up high availability for the Config Server

The previous sections explored how to setup the Config Server, allowing real-time refresh of configuration properties. However, the Config Server is a single point of failure in this architecture.

There are three single points of failure in the default architecture established in the previous section. One of them is the availability of the Config Server itself, the second one is the Git repository, and the third one is the RabbitMQ server.

The following diagram shows a high-availability architecture for the Config Server:



The architecture mechanisms and rationale are explained as follows:

The Config Server requires high availability, since the services won't be able to bootstrap if the Config Server is not available. Hence, redundant Config servers are required for high availability. However, the applications can continue to run if the Config Server is unavailable after the services are bootstrapped. In this case, services will run with the last-known configuration state. Hence, the Config Server availability is not at the same critical level as the microservices availability.

In order to make the Config Server highly available, we need multiple instances of the Config servers. Since the Config Server is a stateless HTTP service, multiple instances of configuration servers can be run in parallel. Based on the load on the configuration server, a number of instances have to be adjusted. The `bootstrap.properties` is not capable of handling more than one server address. Hence, multiple configuration servers should be configured to run behind a load balancer or behind a local DNS with failover and fallback capabilities. The load balancer or DNS server URL will be configured in the microservices `bootstrap.properties`. This is with the assumption that the DNS or the load balancer is highly available and capable of handling failovers.

In a production scenario, it is not recommended to use a local-file-based Git repository. The configuration server should be typically backed with a highly available Git service. This is possible by either using an external high available Git service or a highly available internal Git service. SVN can also be considered.

Having said that, an already bootstrapped Config Server is always capable of working with a local copy of the configuration. Hence, we need a highly available Git only when the Config Server needs to be scaled. Therefore, this too is not as critical as the microservices availability or the Config Server availability.



The GitLab example for setting up high availability is available at the following link: <https://about.gitlab.com/high-availability/>

RabbitMQ also has to be configured for high availability. The high availability for RabbitMQ is needed only to push configuration changes dynamically to all instances. Since this is more of an offline, controlled activity, it does not really require the same high availability as required by the components.

Rabbit MQ high availability can be achieved by either using a cloud service or a locally configured highly available Rabbit MQ service.



Setting up high availability for Rabbit MQ is documented at the link <https://www.rabbitmq.com/ha.html>.

Monitoring Config Server health

The Config Server is nothing but a Spring Boot application, and is, by default, configured with an Actuator. Hence, all Actuator endpoints are applicable for the Cloud Server. The health of the server can be monitored using the following Actuator URL:

`http://localhost:8888/health`

Config Server for configuration files

We may run into scenarios where we need a complete configuration file to be externalized, such as `logback.xml`. The Config Server provides a mechanism to configure and store such files. This is achievable by using the URL format as follows:

`/{{name}}/{{profile}}/{{label}}/{{path}}`

The name, profile, and label has the same meaning as explained earlier. The path indicates the filename, such as `logback.xml`.

Completing changes to use Config Server

In order to build this capability to the complete BrownField Airline's PSS, we have to make use of the configuration server for all services.



All microservices in our `chapter7.*` examples need to make similar changes to look to the Config Server for getting the configuration parameters.

We are not externalizing the queue names used in the Search, Booking, and Check-in services at the moment. Later in this chapter, these will be changed to use Spring Cloud Streams.

Eureka for registration and discovery

So far, we have achieved externalizing configuration parameters, as well as load balancing across many service instances.

Ribbon-based load balancing is sufficient for most of the microservices requirements. However, this approach falls short in these scenarios:

- If there is a large number of microservices, and if we want to optimize infrastructure utilization, we will have to dynamically change the number of service instances and associated servers. It is not easy to predict and preconfigure the server URLs in a configuration file.
- When targeting cloud deployments for highly scalable microservices, static registration and discovery is not a good solution considering the elastic nature of the cloud environment.
- In cloud deployment scenarios, IP addresses are not predictable and will be difficult to statically configure in a file. We will have to update the configuration file every time there is a change in address.

The Ribbon approach partially addresses this issue. With Ribbon, we can dynamically change the service instances, but whenever we add new service instances or shut down instances, we will have to manually go and update the Config Server. Though the configuration changes will be automatically propagated to all required instances, manual configuration changes will not work with large-scale deployments. When managing large deployments, automation, wherever possible, is paramount.

To fix this gap, the microservices should self-manage their life cycle by dynamically registering service availability and provision-automated discovery for consumers.

Understanding dynamic service registration and discovery

Dynamic registration is primarily from the service provider's point of view. With dynamic registration, when a new service is started, it automatically enlists its availability in a central service registry. Similarly, when a service goes out of service, it is automatically delisted from the service registry. The registry always keeps up-to-date information of the services available, as well as their metadata.

Dynamic discovery is applicable from the service consumer's point of view. Dynamic discovery is where clients look for the service registry to get the current state of the services topology and then invoke the services accordingly. In this approach, instead of statically configuring the service URLs, the URLs are picked up from the service registry.

The clients may keep a local cache of the registry data for faster access. Some registry implementations allow clients to keep a watch on the items they are interested in. In this approach, the state changes in the registry server will be propagated to the interested parties to avoid using stale data.

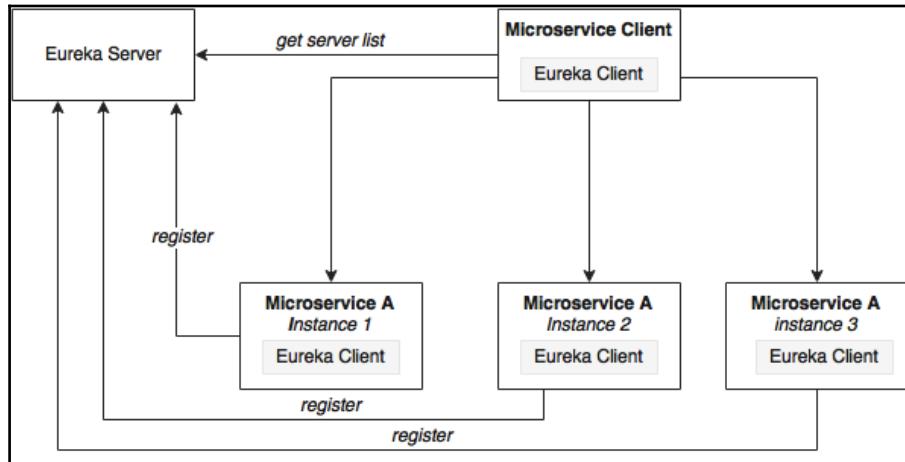
There are a number of options available for dynamic service registration and discovery. Netflix Eureka, Zookeeper, and Consul are available as part of Spring Cloud, as shown in the `start.spring.io` screenshot that follows. Etcd is another service registry available outside of Spring Cloud to achieve dynamic service registration and discovery. In this chapter, we will focus on the Eureka implementation:

Cloud Discovery

- Eureka Discovery
Service discovery using spring-cloud-netflix and Eureka
- Eureka Server
spring-cloud-netflix Eureka Server
- Zookeeper Discovery
Service discovery with Zookeeper and spring-cloud-zookeeper-discovery
- Cloud Foundry Discovery
Service discovery with Cloud Foundry
- Consul Discovery
Service discovery with Hashicorp Consul

Understanding Eureka

Spring Cloud Eureka also comes from the Netflix OSS. The Spring Cloud project provides a Spring-friendly declarative approach for integrating Eureka with Spring-based applications. Eureka is primarily used for self-registration, dynamic discovery, and load balancing. The Eureka internally uses Ribbon for load balancing.



As shown in the preceding diagram, Eureka consists of a server component and a client-side component. The server component is the registry in which all microservices register their availability. The registration typically includes service identity and its URLs. The microservices use the **Eureka Client** for registering their availability. The consuming components will also use the **Eureka Client** for discovering the service instances.

When a microservice is bootstrapped, it reaches out to the **Eureka Server** and advertises its existence with the binding information. Once registered, the service endpoint sends ping requests to the registry every 30 seconds to renew its lease. If a service endpoint cannot renew its lease for a few times, that service endpoint is taken out of the service registry. The registry information is replicated to all Eureka clients so that the clients need to go to the remote **Eureka Server** for each and every request. Eureka clients fetch the registry information from the server and cache it locally. After that, the clients use that information to find other services. This information is updated periodically (every 30 seconds) by getting the delta updates between the last fetch cycle and the current one.

When a client wants to contact a microservice endpoint, the **Eureka Client** provides a list of currently available services based on the requested service ID. The **Eureka Server** is zone-aware. Zone information can also be supplied when registering a service.

When a client requests for a services instance, the Eureka service tries to find the service running in the same zone. The Ribbon client then load balances across these available service instances supplied by the **Eureka Client**. The communication between the **Eureka Client** and **Eureka Server** uses REST and JSON.

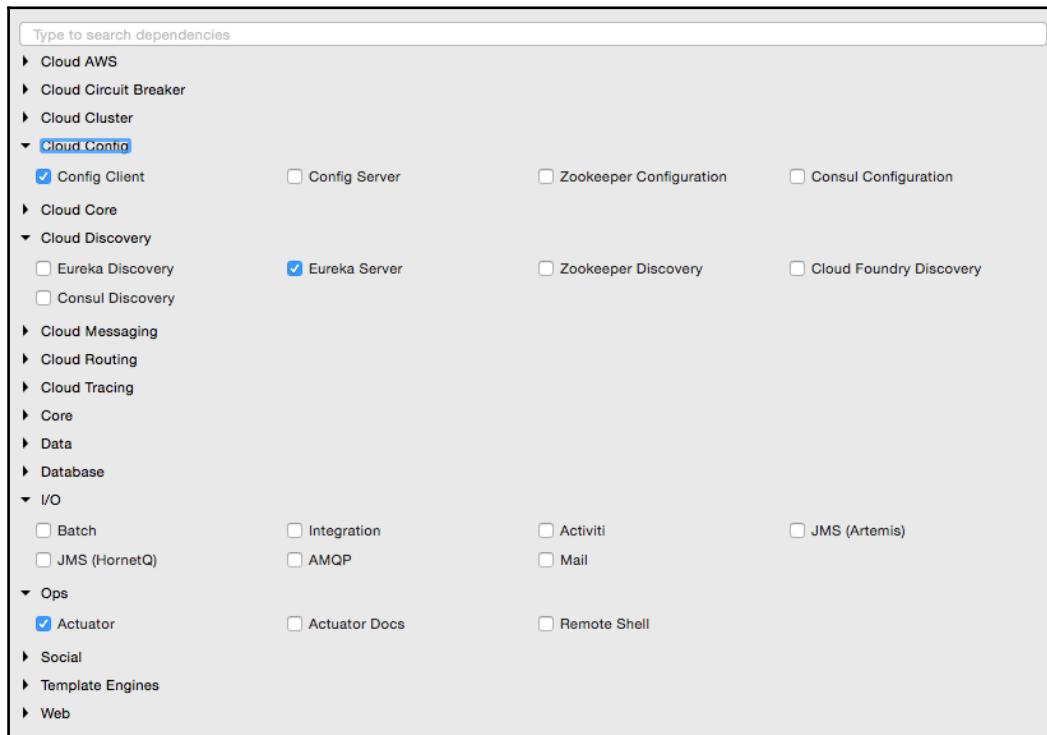
Setting up the Eureka Server

In this section, we will run through the steps required for setting up the Eureka Server.

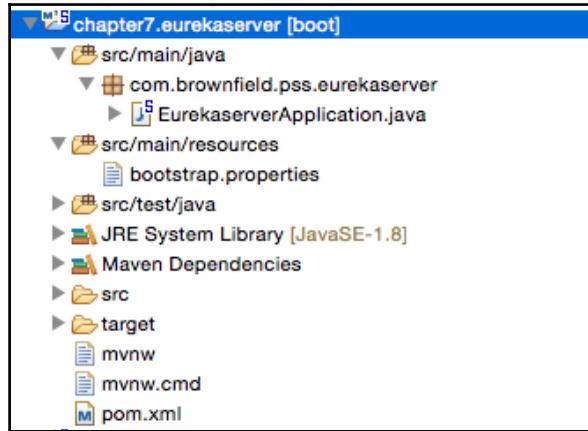


The full source code of this section is available under the `chapter7.eurekaserver` project in the code files under <https://github.com/rajeshrv/Spring5Microservice>. Note that the Eureka Server registration and refresh cycles takes up to 30 seconds. Hence, when running services and clients, wait for 40-50 seconds.

Start a new Spring Starter project and select **Config Client**, **Eureka Server**, and **Actuator**, as shown in the following screenshot:



The project structure of the Eureka server is shown in the following screenshot:



Note that the main application is named `EurekaserverApplication.java`.

Rename `application.properties` to `bootstrap.properties`, as this is using the Config Server. As we have done earlier, configure the details of the Config Server in the `bootstrap.properties` so that it can locate the Config Server instance. The `bootstrap.properties` will look like this:

```
spring.application.name=eureka-server1
server.port:8761
spring.cloud.config.uri=http://localhost:8888
```

The Eureka Server can be set up in a standalone mode or in a clustered mode. We will start with the standalone mode. By default, the Eureka Server itself is another Eureka Client. This is particularly useful when there are multiple Eureka servers running for high availability. The client component is responsible for synchronizing the state from other Eureka servers. The Eureka Client is taken to its peers by configuring the `eureka.client.serviceUrl.defaultZone` property.

In the standalone mode, we will point the `eureka.client.serviceUrl.defaultZone` back to the same standalone instance. Later, we will see how we can run Eureka servers in a clustered mode.

Perform the following steps to set up Eureka server:

1. Create a `eureka-server1.properties` file and update it in the Git repository. `eureka-server1` is the name of the application given in the `applications bootstrap.properties` in the previous step. As shown next, the `serviceUrl` points back to the same server. Once the following properties are added, commit the file to the Git repository:

```
spring.application.name=eureka-server1
eureka.client.serviceUrl.defaultZone:http://localhost:8761/eureka/
eureka.client.registerWithEureka:false
eureka.client.fetchRegistry:false
```

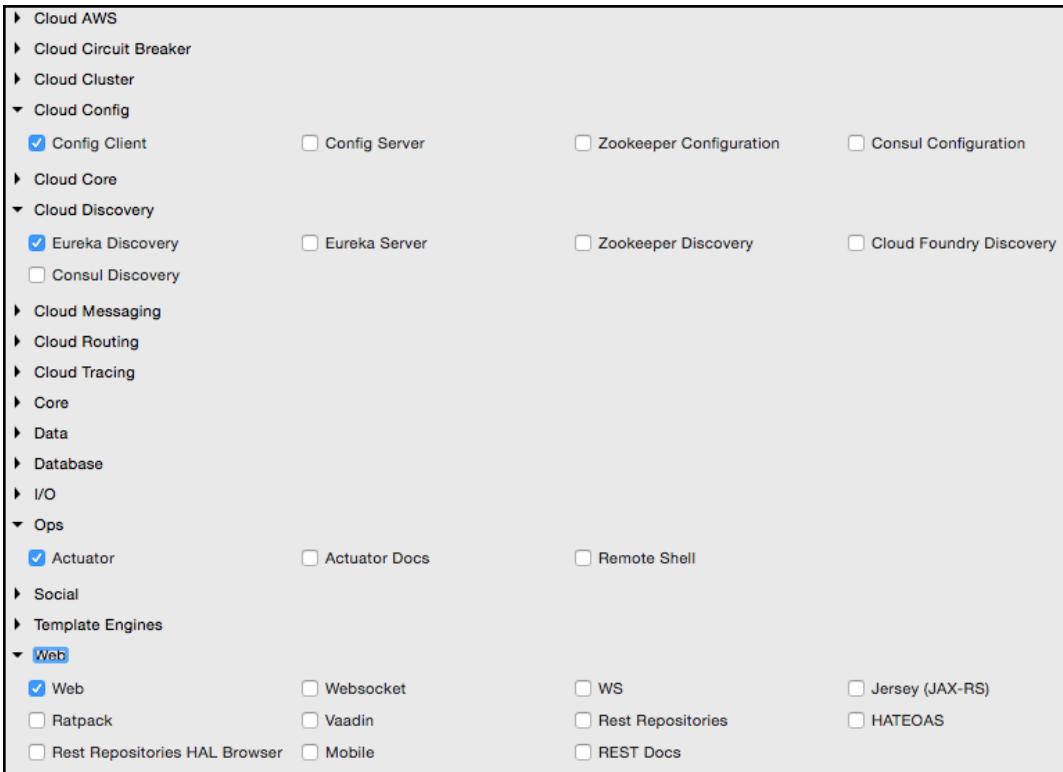
2. Change the default `Application.java`. In this example, the package is renamed as `com.brownfield.pss.eurekaserver` and the class name is also changed to `EurekasherApplication`. In `EurekasherApplication`, add `@EnableEurekaServer`:

```
@EnableEurekaServer
@SpringBootApplication
public class EurekasherApplication {
```

3. We are now ready to start the Eureka Server. Ensure the Config Server is also started. Right-click on the application and Run As, Spring Boot App. Once the application starts, open the following link in a browser to see the Eureka console:

```
http://localhost:8761
```

4. In the console, note that there is no instance registered under the instances currently registered with Eureka. Since there are no services started with the Eureka Client enabled, the list is empty at this point.
5. Making a few changes to our microservices will enable dynamic registration and discovery using the Eureka service. To do this, first we have to add Eureka dependencies in `pom.xml`. If the services are being built up fresh using the Spring Starter project, then select **Config Client**, **Actuator**, **Web**, as well as the **Eureka Discovery** client, as shown in the following screenshot:



6. Since we are modifying all our microservices, add the following additional dependency to all the microservices in their `pom.xml` files:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
```

The following property has to be added to all the microservices in their respective configuration files under `config-repo`. This will help the microservices to connect to the Eureka Server. Commit to Git once the updates are completed:

```
eureka.client.serviceUrl.defaultZone:
    http://localhost:8761/eureka/
```

7. Add `@EnableDiscoveryClient` to all the microservices in their respective Spring Boot main classes. This will ask Spring Boot to register these services at startup to advertise its availability.
8. Instead of `RestTemplate`, we will use `@FeignClient` in this example by introducing a `FareServicesProxy`, as shown next:

```
@FeignClient(name="fares-service")
public interface FareServiceProxy {
    @RequestMapping(value = "fares/get",
    method=RequestMethod.GET)
    Fare getFare(@RequestParam(value="flightNumber")
    String flightNumber, @RequestParam(value="flightDate")
    String flightDate);
}
```

9. In order to do this, we have to add a Feign dependency:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-feign</artifactId>
</dependency>
```

10. Start all services except website service.
11. If you go to the Eureka URL, you can see that all three instances are up and running:

`http://localhost:8761`

Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
BOOK-SERVICE	n/a (1)	(1)	UP (1) - 192.168.0.102:book-service:8060
CHECKIN-SERVICE	n/a (1)	(1)	UP (1) - 192.168.0.102:checkin-service:8070
FARES-SERVICE	n/a (1)	(1)	UP (1) - 192.168.0.102:fares-service:8080
SEARCH-SERVICE	n/a (1)	(1)	UP (1) - 192.168.0.102:search-service:8090

12. Change the website project `bootstrap.properties` to make use of Eureka rather than connecting directly to the service instances. We will use the load-balanced `RestTemplate`. Commit these changes to the Git repository:

```
spring.application.name=test-client
eureka.client.serviceUrl.defaultZone:
    http://localhost:8761/eureka/
```

13. Add `@EnableDiscoveryClient` to the Application class to make the client Eureka aware.
14. Edit both `Application.java` as well as `BrownFieldSiteController.java`. Add `RestTemplate` instances. This time, we annotate them with `@LoadBalanced` to ensure that we are using the load balancing features using Eureka and Ribbon. `RestTemplate` cannot be automatically injected. Hence, we have to provide a configuration entry, as follows:

```
@Configuration
class AppConfiguration {
    @LoadBalanced
    @Bean
    RestTemplate restTemplate() {
        return new RestTemplate();
    }
}
@Autowired
RestTemplate restClient;
```

15. We will use these `RestTemplate` instances to call the microservices. We will replace the hard-coded URLs with service IDs, which are registered in the Eureka Server. In the following code, we use the service names `search-service`, `book-service` and `checkin-service` instead of explicit host names and ports:

```
Flight[] flights = searchClient.postForObject(
    "http://search-service/search/get",
    searchQuery, Flight[].class);

long bookingId = bookingClient.postForObject(
    "http://book-service/booking/create", booking,
    long.class);

long checkinId = checkInClient.postForObject(
    "http://checkin-service/checkin/create", checkIn,
    long.class);
```

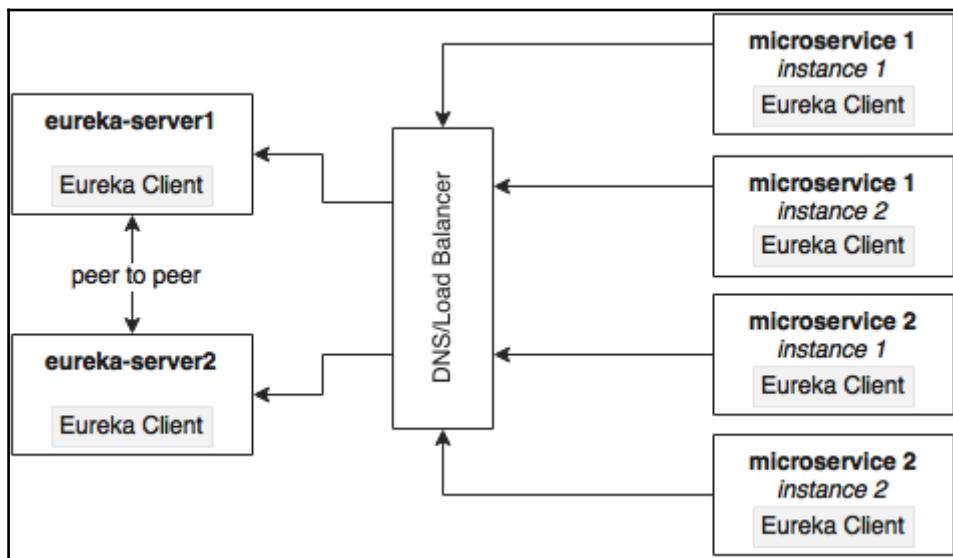
16. We are now ready to test. Run the website project. If everything is fine, the website project's `CommandLineRunner` will successfully perform search, book, and check-in.
17. The same can also be tested using the browser by pointing the browser to `http://localhost:8001`.

High availability for Eureka

In the previous example, there was only one Eureka Server in the standalone mode. This is not good enough for a real production system.

The Eureka Client connects to the server, fetches registry information, and stores it locally in a cache. The client always works with this local cache. The Eureka Client checks the server periodically for any state changes. In case of a state change, it downloads the changes from the server and updates the cache. If the Eureka Server is not reachable, then the Eureka Client can still work with the last known state of the servers based on the data available in the client cache. However, this could lead to stale state issues quickly.

This section will explore high availability of the Eureka Server. The high availability architecture is shown in this diagram:



The Eureka Server is built with a peer-to-peer data synchronization mechanism. The run-time state information is not stored in a database, but managed using an in-memory cache. The high availability implementation favors availability and partition tolerance in the CAP theorem, leaving out consistency. Since the Eureka Server instances are synchronized with each other using asynchronous mechanism, the states may not always match between server instances. Peer-to-peer synchronization is done by pointing service URLs to each other.

If there is more than one Eureka Server, each one has to be connected to at least one of the peer servers. Since the state is replicated across all peers, Eureka clients can connect to any one of the available Eureka servers.

The best way to achieve high availability for Eureka is to cluster multiple Eureka servers and run them behind a load balancer or a local DNS. The clients always connect to the server using the DNS or load balancer. At runtime, the load balancer will take care of selecting the appropriate servers. This load balancer address will be provided to the Eureka clients.

This section will showcase how to run two Eureka servers in a cluster for high availability. For this, define two property files--eureka-server1 and eureka-server2. These are peer servers--if one fails, the other one will take over. Each of these servers will also act as a client for the other so that they can sync their states. The two property files defined are defined next. Upload and commit these properties to the Git repository. In the following configurations, the client URLs are pointing to each other, forming a peer network:

```
#eureka-server1.properties
eureka.client.serviceUrl.defaultZone:http://localhost:8762/eureka/
eureka.client.registerWithEureka:false
eureka.client.fetchRegistry:false

#eureka-server2.properties
eureka.client.serviceUrl.defaultZone:http://localhost:8761/eureka/
eureka.client.registerWithEureka:false
eureka.client.fetchRegistry:false
```

Update `bootstrap.properties` of Eureka and change the application name to `eureka`. Since we are using two profiles, the Config Server will look for either `eureka-server1` or `eureka-server2` based on the active profile supplied at startup:

```
spring.application.name=eureka
spring.cloud.config.uri=http://localhost:8888
```

Start two instances of the Eureka servers--server1 on 8761 and server2 on 8762:

```
java -jar -Dserver.port=8761 -Dspring.profiles.active=server1
      demo-0.0.1-SNAPSHOT.jar

java -jar -Dserver.port=8762 -Dspring.profiles.active=server2
      demo-0.0.1-SNAPSHOT.jar
```

All our services are still pointing to the first server, server1. Open both the browser windows.

```
http://localhost:8761  
http://localhost:8762
```

Start all microservices. The one which opened 8761 will immediately reflect the changes, whereas the other one will take 30 seconds to reflect the states. Since both the servers are in a cluster, the state is synchronized between these two servers. If we keep these servers behind a load balancer or DNS, then the client will always connect to one of the available servers.

After completing this exercise, switch back to the standalone mode for the remaining exercises.

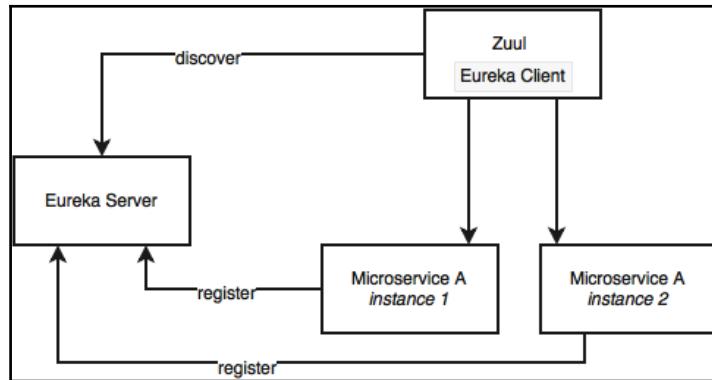
Zuul proxy as the API Gateway

In most microservices implementations, internal microservices endpoints are not exposed outside. They are kept as private services. A set of public services will be exposed to the clients using an API Gateway. There are many reasons to do this, some of which are listed as follows:

- Only a selected set of microservices are required by the clients.
- If there are client-specific policies to be applied, it is easy to apply them in a single place rather than in multiple places. An example of such a scenario is the cross-origin access policy.
- It is hard to implement client-specific transformations at the service endpoint.
- If there is data aggregation required, especially to avoid multiple client calls in a bandwidth restricted environment, then a gateway is required in the middle.

Zuul is a simple gateway service or edge service, which well suits such situations. Zuul also came from the Netflix family of microservices products. Unlike many enterprise API gateway products, Zuul provides complete control to developers to configure or program based on specific requirements.

The following diagram shows Zuul acting as a proxy and load balancer for **Microservice A**:



The Zuul proxy internally uses the **Eureka Server** for service discovery, and Ribbon for load balancing between service instances.

The Zuul proxy is also capable of routing, monitoring, managing resiliency, security, and so on. In simple terms, we can consider Zuul as a reverse proxy service. With Zuul, we can even change the behaviors of the underlying services by overriding them at the API layer.

Setting up Zuul

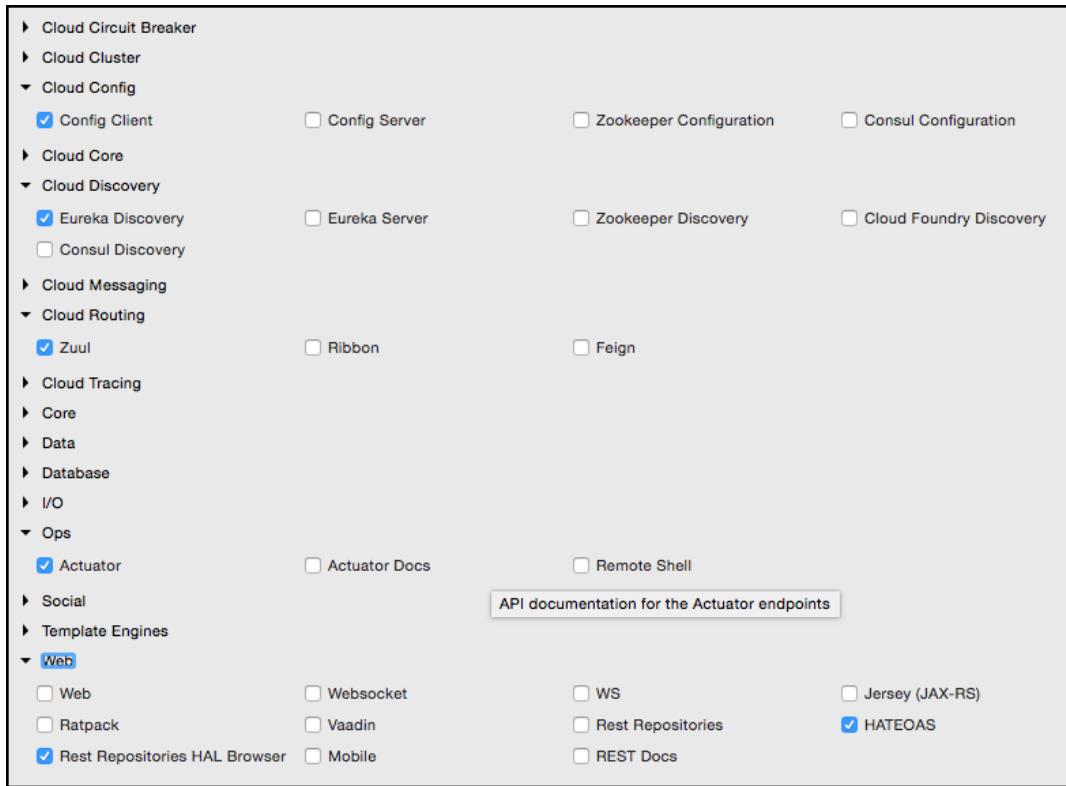
Unlike the Eureka Server and the Config Server, in typical deployments, Zuul is specific to a microservice. However, there are deployments in which one API Gateway covers many microservices. In this case, we are going to add Zuul for each of our microservices--Search, Booking, Fare, and Check-In.



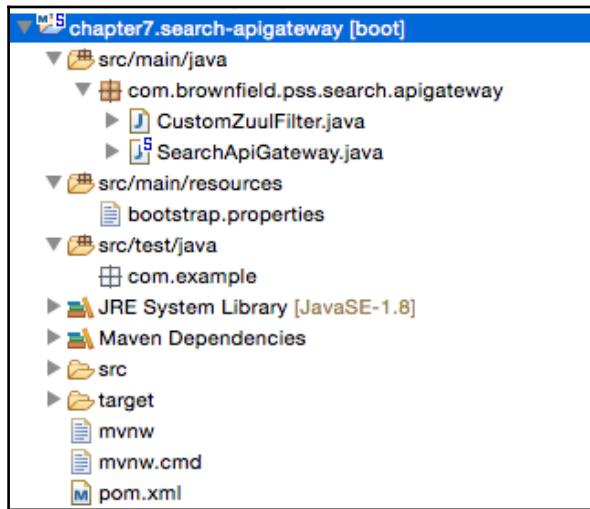
The full source code of this section is available under the `chapter7.*-apigateway` project in the code files.

Perform the following steps for setting up Zuul:

1. Convert the microservices one by one. Start with the Search API Gateway. Create a new Spring Starter project and select **Zuul**, **Config Client**, **Actuator**, and **Eureka Discovery**:



The project structure for the search-apigateway is shown in this screenshot:



2. The next step is to integrate the API Gateway with Eureka and the Config Server. Create a `search-apigateway.property` file with the contents as given next, and commit to the Git repository.

This configuration also sets a rule on how to forward traffic. In this case, any request coming on the `/api` endpoint of the API Gateway should be sent to the search-service:

```
spring.application.name=search-apigateway
zuul.routes.search-apigateway.serviceId=search-service
zuul.routes.search-apigateway.path=/api/**
eureka.client.serviceUrl.defaultZone:
http://localhost:8761/eureka/
```

The `search-service` is the service ID of Search service, and it will be resolved using the Eureka Server.

3. Update `bootstrap.properties` of `search-apigateway` as follows. There is nothing new in this configuration--a name to the service, port, and the Config Server URL:

```
spring.application.name=search-apigateway
server.port=8095
spring.cloud.config.uri=http://localhost:8888
```

4. Edit `Application.java`. In this case, the package name and the class name also change to `com.brownfield.pss.search.apigateway` and `SearchApiGateway` respectively. Also add `@EnableZuulProxy` to tell Spring Boot that this is a Zuul proxy:

```
@EnableZuulProxy  
@EnableDiscoveryClient  
@SpringBootApplication  
public class SearchApiGateway {
```

5. Run this as Spring Boot app. Before that, ensure that the Config Server, the Eureka Server, and the Search microservice are running.
6. Change the website project's `CommandLineRunner`, as well as `BrownFieldSiteController`, to make use of the API Gateway:

```
Flight[] flights = searchClient.postForObject(  
    "http://search-apigateway/api/search/get",  
    searchQuery, Flight[].class);
```

In this case, the Zuul proxy acts as a reverse proxy, which proxies all microservice endpoints to consumers. In the preceding example, the Zuul proxy does not add much value, as we just pass through the incoming requests to the corresponding backend service.

Zuul is particularly useful when we have one or more requirements like these:

- Enforcing authentication and other security policies at the gateway instead of doing that on every microservice endpoint. The gateway can handle security policies, token handling, and so on before passing the request to the relevant services behind. It can also do basic rejections based on some business policies, such as blocking requests coming from certain black-listed users.
- Business insights and monitoring can be implemented at the gateway level. Collect real-time statistical data and push it to an external system for analysis. This will be handy, as we can do this at one place rather than applying it across many microservices.
- API gateways are useful in scenarios where dynamic routing is required based on fine-grained controls. For example, send requests based on certain specific business values such as gold customer to a different service instances. For example, all requests coming from a region to be sent to one group of service instances. Another example--all requests requesting for a particular product have to be routed to a group of service instances.

- Handling the load shredding and throttling requirements is another scenario where API gateways are useful. This is when we have to control load based on set thresholds such as number of requests in a day. For example, control requests coming from a low-value third-party online channel.
- The Zuul gateway is useful for fine-grained load balancing scenarios. Zuul, Eureka Client, and Ribbon together provide fine-grained controls over load balancing requirements. Since the Zuul implementation is nothing but another Spring Boot application, the developer has full control of the load balancing.
- The Zuul gateway is also useful in scenarios where data aggregation requirements are in place. If the consumer wants higher-level coarse-grained services, then the gateway can internally aggregate data by calling more than one service on behalf of the client. This is particularly applicable when the clients are working in low-bandwidth environments.

Zuul also provides a number of filters. These filters are classified under pre-filters, routing filters, post filters, and error filters. As the names indicate, these are applied at different life cycle stages of a service call. Zuul also provides an option for developers to write custom filters. In order to write a custom filter, extend from the abstract `ZuulFilter`, and implement methods shown as follows:

```
public class CustomZuulFilter extends ZuulFilter{  
    public Object run(){}
    public boolean shouldFilter(){}
    public int filterOrder(){}
    public String filterType(){}
}
```

Once a custom filter is implemented, add that class to the main context. In our example case, add this to the `SearchApiGateway` class as follows:

```
@Bean  
public CustomZuulFilter customFilter() {  
    return new CustomZuulFilter();  
}
```

As mentioned earlier, the Zuul proxy is a Spring Boot service. We can customize the gateway programmatically in the way we want. As shown in the following code, we can add custom endpoints to the gateway, which in turn can call the backend services:

```
@RestController
class SearchAPIGatewayController {
    @RequestMapping("/")
    String greet(HttpServletRequest req) {
        return "<H1>Search Gateway Powered By Zuul</H1>";
    }
}
```

In the preceding case, it just adds a new endpoint, and returns a value from the gateway. We can further use the `@Loadbalanced` RestTemplate to call a backend service. Since we have full control, we can do transformations, data aggregation, and so on. We can also use the Eureka APIs to get the server list and implement completely independent load balancing or traffic shaping mechanisms instead of the out-of-the-box load balancing features provided by Ribbon.

High availability of Zuul

Zuul is just a stateless service with an HTTP endpoint, hence, we can have as many Zuul instances as we need. There is no affinity or stickiness required. However, the availability of Zuul is extremely critical, as all the traffic from the consumer to the provider flows through the Zuul proxy. However, the elastic scaling requirements are not as critical as the backend microservices, where all the heavy lifting is happening.

The high availability architecture of Zuul is determined by the scenario in which we are using Zuul. The typical usage scenarios are as follows:

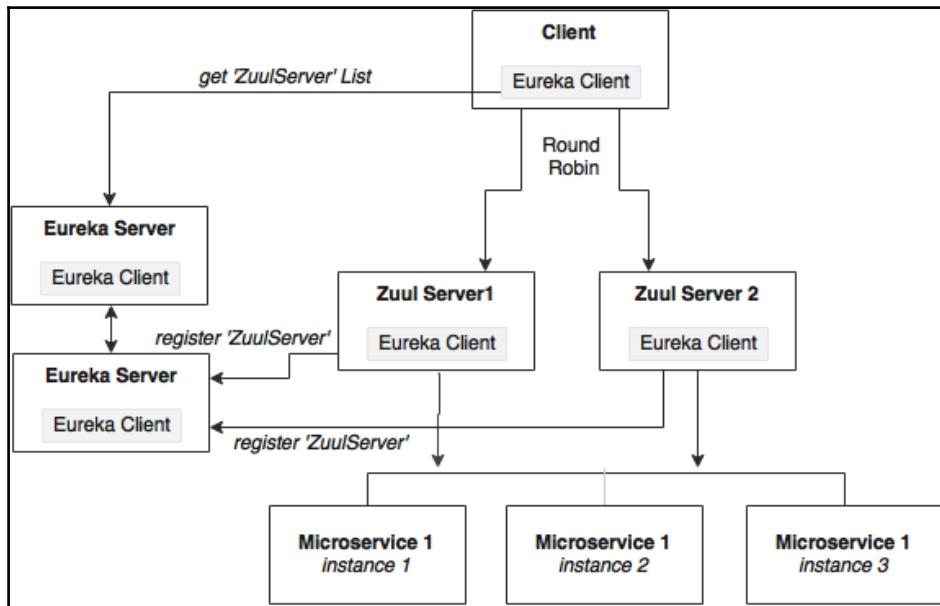
- When a client-side java script MVC such as Angular JS accesses Zuul services from a remote browser
- Another microservice or non-microservice accesses services via Zuul

In some cases, the client may not have capabilities to use the Eureka Client libraries, such as a legacy application written on PL/SQL. In some cases, organization policies do not allow Internet clients to handle client-side load balancing. In the case of browser-based clients, there are third-party Eureka JavaScript libraries available.

It all boils down to whether the client is using Eureka Client libraries or not. Based on this, there are two ways we can set up Zuul for high availability.

High availability of Zuul when the client is also a Eureka Client

In this case, since the client is also another Eureka Client, Zuul can be configured just like other microservices. Zuul itself registers to Eureka with a service ID. The clients then use Eureka and the service ID to resolve Zuul instances.



As shown in the preceding diagram, Zuul services register themselves with Eureka with a service ID, `search-apigateway` in our case. The **Eureka Client** will ask for the server list with the ID `search-apigateway`. The **Eureka Server** returns the list of servers based on the current Zuul topology. The **Eureka Client**, based on this list, picks up one of the servers, and initiates the call.

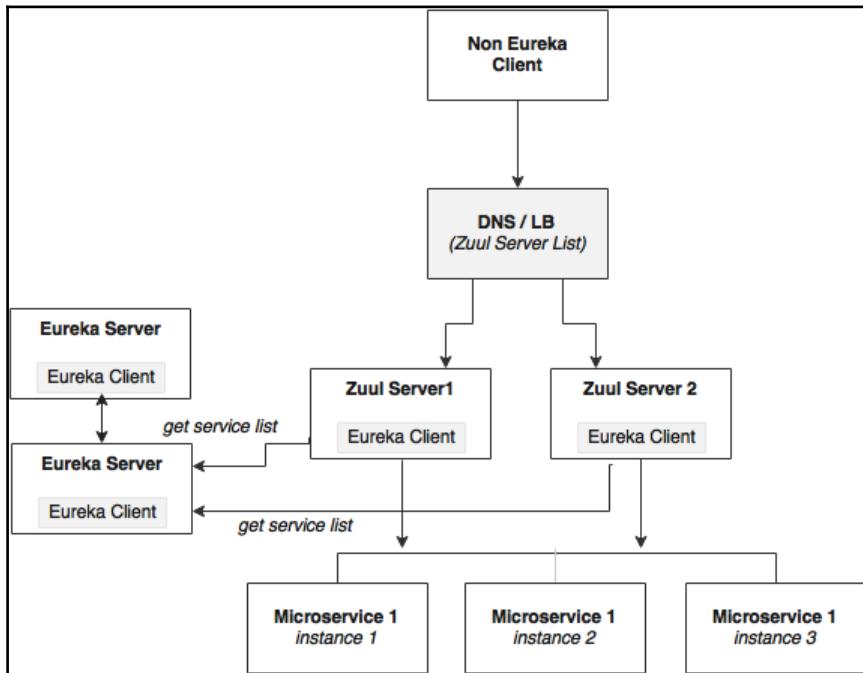
As we saw earlier, the client will use the service ID to resolve the Zuul instance. In the following case, search-apigateway is the Zuul instance ID registered with Eureka:

```
Flight[] flights = searchClient.postForObject(  
    "http://search-apigateway/api/search/get",  
    searchQuery, Flight[].class);
```

High availability when client is not a Eureka Client

In this case, the client is not capable of handling the load balancing by using the Eureka Server. As shown in the following diagram, the client sends the request to a load balancer, which, in turn, identifies the right Zuul service instance.

The Zuul instances, in this case, will be running behind a load balancer such as HAProxy, or a hardware load balancer like NetScaler:



The microservices will still be load balanced by Zuul using the Eureka Server.

Completing Zuul for all other services

In order to complete this exercise, add an API Gateway for all our microservices. The following steps are required to achieve this task:

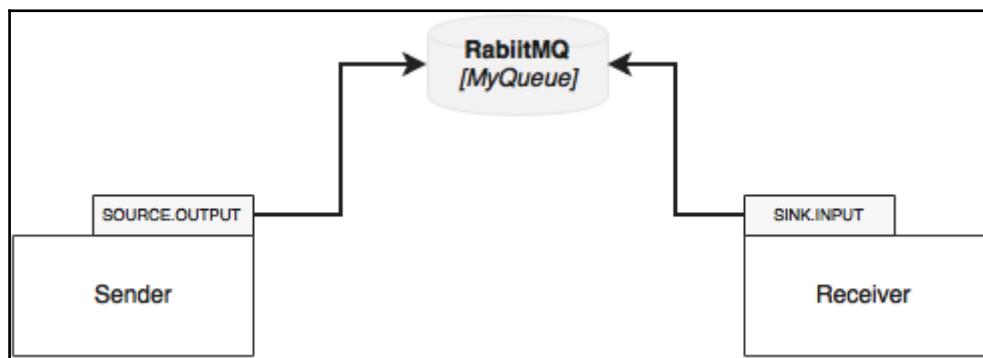
1. Create new property files per service and check in to Git repositories.
2. Change application.properties to bootstrap.properties and add the required configurations.
3. Add @EnableZuulProxy in the application.
4. @EnableDiscoveryClient in all applications.
5. Optionally, change the default generated package names and the filenames.

In the end, we will have the following API Gateway projects:

- chapter7.fares-apigateway
- chapter7.search-apigateway
- chapter7.checkin-apigateway
- chapter7.book-apigateway

Streams for reactive microservices

Spring Cloud Streams provides an abstraction over the messaging infrastructure. The underlying messaging implementation can be RabbitMQ, Redis, or Kafka. Spring Cloud Streams provides a declarative approach for sending and receiving messages.



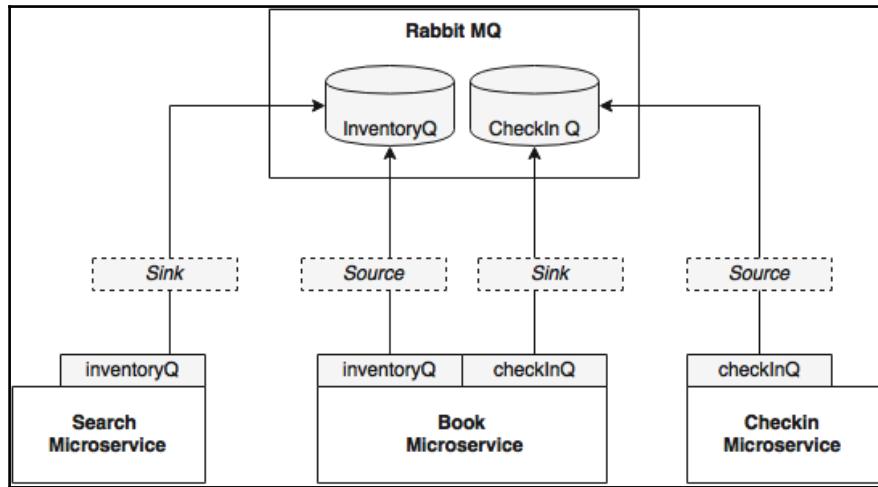
As shown in the preceding diagram, the Cloud Streams work with the concept of a Source and a Sink. The Source represents the sender perspective of the messaging, and Sink represents the receiver perspective of the messaging.

In the given example, the **Sender** defines a logical queue called `Source.OUTPUT` to which the **Sender** sends messages. The **Receiver** defines a logical queue called `Sink.INPUT` from which the **Receiver** retrieves messages. The physical binding of `OUTPUT` to `INPUT` is managed through the configuration. In this case, both link to the same physical queue, `MyQueue` on RabbitMQ. So, at one end, `Source.OUTPUT` will be pointed to `MyQueue`, and on the other end, `Sink.INPUT` will be pointed to the same `MyQueue`.

Spring Cloud offers the flexibility to use multiple messaging providers in one application, such as connecting an input stream from the Kafka to a Redis output stream without managing the complexities. Spring Cloud Streams is the basis for message-based integration. The Cloud Stream Modules sub-project is another Spring Cloud library that provides many endpoint implementations.

As the next step, rebuild the inter-microservice messaging communication with the Cloud Streams. As shown in the diagram, we will define a `SearchSink` connected to `InventoryQ` under the Search microservice. `Booking` will define a `BookingSource` for sending inventory change messages connected to `InventoryQ`. Similarly, `Checkin` defines a `CheckinSource` for sending check-in messages. `Booking` defines a sink `BookingSink` for receiving messages, both bound to the `CheckinQ` queue on Rabbit MQ.

The following diagram shows the example setup using stream based architecture:



In this example, we will use RabbitMQ as the message broker. Perform the following steps:

1. Add the following Maven dependency to Booking, Search, and Check-in, as these are the three modules using messaging:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
```

2. Add the following two properties to booking-service.properties.

These properties bind the logical queues, inventoryQ to the physical inventoryQ, and the logical checkinQ to physical checkinQ.

```
spring.cloud.stream.bindings.inventoryQ.destination=inventoryQ
spring.cloud.stream.bindings.checkInQ.destination=checkInQ
```

3. Add the following property to search-service.properties. This property binds the logical queue inventoryQ to the physical inventoryQ:

```
spring.cloud.stream.bindings.inventoryQ.destination=inventoryQ
```

4. Add this next property to the checkin-service.properties. This property binds the logical queue checkinQ to the physical checkinQ:

```
spring.cloud.stream.bindings.checkInQ.destination=checkInQ
```

5. Commit all files to the Git repository.
6. The next step is to edit the code. The Search microservice consumes messages from the Booking microservice. In this case, Booking is the source and Search is the sink.
7. Add `@EnableBinding` to the `Sender` class of the Booking service. This enables the Cloud Stream to work on auto-configurations based on the message broker library available in the class path. In our case, it is RabbitMQ. The parameter, `BookingSource` defines the logical channels to be used for this configuration:

```
@EnableBinding(BookingSource.class)
public class Sender {
```

In this case, `BookingSource` defines a message channel called `inventoryQ`, which is physically bound to the Rabbit MQ `inventoryQ`, as configured in the configuration. `BookingSource` uses an annotation, `@Output`, to indicate that this is of the type `output`--a message that is outgoing from a module. This information will be used for the auto-configuration of the message channel:

```
interface BookingSource {
    public static String InventoryQ="inventoryQ";
    @Output("inventoryQ")
    public MessageChannel inventoryQ();
}
```

8. Instead of defining a custom class, we can also use the default `Source` class that comes with Spring Could Streams, if the service has only one source and sink:

```
public interface Source {
    @Output("output")
    MessageChannel output();
}
```

9. Define a message channel in the sender based on `BookingSource`. The following code will inject an output message channel with the name `inventory`, which is already configured in `BookingSource`:

```
@Output (BookingSource.InventoryQ)
@.Autowired
private MessageChannel messageChannel;
```

10. Reimplement the send message method in the Booking sender:

```
public void send(Object message) {
    messageChannel.send(
        MessageBuilder.withPayload(message).build());
}
```

11. Now add the following to the Search Receiver class the same way we did it for the Booking service:

```
@EnableBinding(SearchSink.class)
public class Receiver {
```

12. In this case, the SearchSink interface will look like the following. This will define the logical sink queue it is connected with. The message channel in this case is defined as @Input to indicate that this message channel is to accept messages:

```
interface SearchSink {
    public static String INVENTORYQ="inventoryQ";

    @Input("inventoryQ")
    public MessageChannel inventoryQ()
}
```

13. Amend the Search service to accept this message:

```
public void accept(Map<String, Object> fare) {
    searchComponent.updateInventory(
        (String)fare.get("FLIGHT_NUMBER"),
        (String)fare.get("FLIGHT_DATE"),
        (int)fare.get("NEW_INVENTORY"));}
```

14. We will still need the RabbitMQ configurations that we have in our configuration files to connect to the message broker:

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest
server.port=8090
```

15. Run all the services, and run the website project. If everything is fine, the website project successfully executes the search, book, and check-in functions. The same can also be tested using the browser by pointing to <http://localhost:8001>.

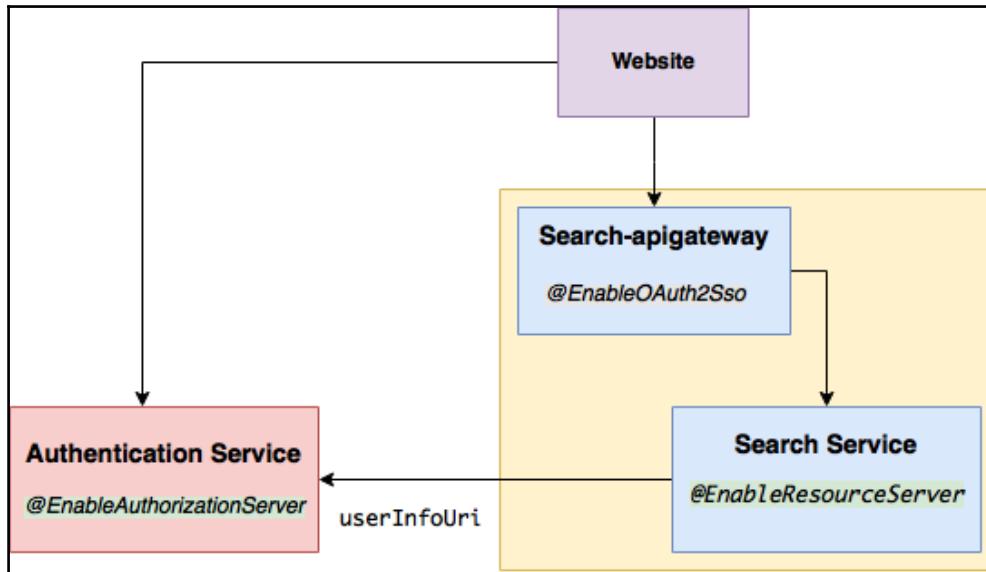
Protecting microservices with Spring Cloud Security

In a monolithic web application, once the user is logged in, user-related information will be stored in an HTTP session. All subsequent requests will be validated against the HTTP session. This is simple to manage, since all requests will be routed through the same session, either through the session affinity or offloaded, shared session store.

In the case of microservices, it is harder to protect from unauthorised access, especially, when many services are deployed and accessed remotely. A typical or rather simple pattern for microservices is to implement perimeter security by using gateways as security watchdogs. Any request coming to the gateway will be challenged and validated. In this case, it is then important to ensure that all requests to downstream microservices are funneled through the API Gateway. Generally, the load balancer sitting in the front will be the only client that sends requests to the gateway. In this approach, downstream microservices processes all requests, assuming they are trusted, without authenticating. It means all microservice endpoints will be open for all.

However, this solution may not be acceptable for enterprise cyber security. One of the ways to eliminate this concern is to create network segregation and zones so that the services are exposed only for the gateways to access. In order to simplify this landscape, a common pattern is to set up consumer-driven gateways, which combine multiple microservices access instead of the one-to-one gateways we have used in our example.

Another way of accomplishing this is through token relay, as shown in this diagram:

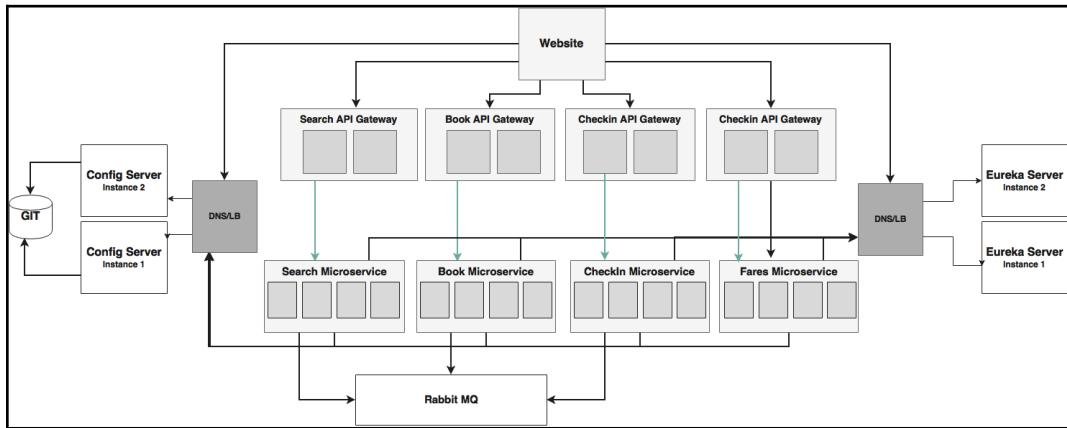


In this case, each microservice will also act as a resource server with a central server for authentication. The API Gateway will forward the request with the token to the downstream microservices for authentication.

Summarising the BrownField PSS architecture

The next diagram shows the overall architecture that we have created with the Config Server, Eureka, Feign, Zuul, and Cloud Streams.

The architecture also includes high availability of all the components. In this case, we are assuming that the client is using the Eureka Client libraries:



The summary of the projects and the port they are listening to is given in the following table:

Microservice	Projects	Port
Book Microservice	chapter7.book	8060-8064
Check In Microservice	chapter7.checkin	8070-8074
Fare Microservice	chapter7.fares	8080-8084
Search Microservice	chapter7.search	8090-8094
Website Client	chapter7.website	8001
Spring Cloud Config Server	chapter7.configserver	8888 / 8889
Spring Cloud Eureka Server	chapter7.eurekaserver	8761 / 8762
Book API Gateway	chapter7.book-apigateway	8095-8099
Check In API Gateway	chapter7.checkin-apigateway	8075-8079
Fares API Gateway	chapter7.fares-apigateway	8085-8089
Search API Gateway	chapter7.search-apigateway	8065-8069

Follow these steps to do a final run:

1. Run RabbitMQ.
2. Build all projects using `pom.xml` at the root level:

```
mvn -Dmaven.test.skip=true clean install
```

3. Run the following projects from the respective folders. Note that you should wait for 40-50 seconds before starting the next service. This will ensure that the dependent services are registered and are available before we start a new service.

```
java -jar target/config-server-1.0.jar  
java -jar target/eureka-server-1.0.jar  
java -jar target/fares-1.0.jar  
java -jar target/fares-1.0.jar  
java -jar target/search-1.0.jar  
java -jar target/checkin-1.0.jar  
java -jar target/book-1.0.jar  
java -jar target/fares-apigateway-1.0.jar  
java -jar target/search-apigateway-1.0.jar  
java -jar target/checkin-apigateway-1.0.jar  
java -jar target/book-apigateway-1.0.jar  
java -jar target/website-1.0.jar
```

4. Open the browser window, and point to `http://localhost:8001`. Follow the steps mentioned in Chapter 6, *Microservices Evolution – A Case Study*, Running and Testing Projects section.

Summary

In this chapter, we learned how to scale Twelve-Factor Spring Boot microservices using the Spring Cloud project. Our learnings were applied on the BrownField Airline's PSS microservice, which we developed in the previous chapter.

We explored the Spring Config Server for externalizing microservices configurations, and how to deploy the Config Server for high availability. We also learned Eureka for load balancing, dynamic service registration, and discovery. Implementation of an API Gateway was examined by implementing Zuul. Finally, we concluded with the reactive style integration of microservices using Spring Cloud Streams.

The BrownField Airline's PSS microservices are now deployable for Internet scale. Other Spring Cloud components such as Hystrix, Sleuth, and others will be covered in the next chapter.

19

Logging and Monitoring Microservices

One of the biggest challenges due to the very distributed nature of internet-scale microservices deployment is the logging and monitoring of individual microservices. It is difficult to trace end-to-end transactions by correlating logs emitted by different microservices. Like monolithic applications, there is no single pane of glass for monitoring microservices. This is important, especially when we deal with enterprise-scale microservices with a number of technologies, as discussed in the previous chapter.

This chapter will cover the necessity and importance of logging and monitoring in microservice deployments. This chapter will further examine the challenges and solutions to address logging and monitoring with a number of potential architectures and technologies.

By the end of this chapter, you will have learned about the following:

- The different options, tools, and technologies for log management
- The use of **Spring Cloud Sleuth** for tracing microservices
- The different tools for end-to-end monitoring of microservices
- The use of **Spring Cloud Hystrix** and **Turbine** for circuit monitoring
- The use of **Data Lake** for enabling business data analysis

Understanding log management challenges

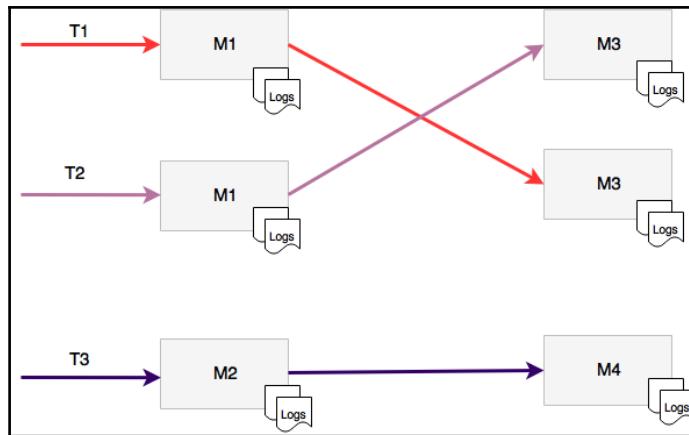
Logs are nothing but streams of events coming from a running process. For traditional JEE applications, a number of frameworks and libraries are available for logging. **Java Logging (JUL)** is an option off-the-shelf from Java itself. Log4j, Logback, and SLF4J are some of the other popular logging frameworks available. These frameworks support both UDP as well as TCP protocols for logging. The applications send log entries to the console or the filesystem. File recycling techniques are generally employed to avoid logs filling up all disk space.

One of the best practices for log handling is to switch off most of the log entries in production due to the high cost of disk IOs. The disk IOs not only slow down the application, but can also severely impact the scalability. Writing logs into the disk also requires a high disk capacity. Running out of the disk space scenario can bring down the application. Logging frameworks provide options to control logging at runtime to restrict what has to be printed and what not. Most of these frameworks provide fine-grained controls over the logging controls. It also provides options for changing these configurations at runtime.

On the other hand, logs may contain important information and have a high value if properly analyzed. Therefore, restricting log entries essentially limits our ability to understand the application behavior.

When moved from traditional deployment to cloud deployment, applications are no longer locked to a particular, predefined machine. Virtual machines and containers are not hardwired with an application. The machines used for deployment can change from time to time. Moreover, containers such as Docker are ephemeral. This essentially means one cannot rely on the persistent state of the disk. Logs written to the disk will be lost once the container is stopped and restarted. Therefore, we cannot rely on the local machine's disk to write log files.

As we discussed in [Chapter 2, Related Architecture Styles and Use Cases](#), one of the principles of the Twelve-Factor application is to avoid routing or storing log files by the application itself. In the context of microservices, they will be running on isolated physical or virtual machines, resulting in fragmented log files. In this case, it would be almost impossible to trace end-to-end transactions that span across multiple microservices.



As shown in the preceding diagram, each microservice emits logs to a local file system. In this case, transaction **T1** calls **M1** followed by **M3**. Since **M1** and **M3** runs on different physical machines, both of them writes respective logs to different log files. This makes it harder to correlate and understand the end-to-end transactions flow. Also, since two instances of **M1** and **M3** are running on two different machines, log aggregation at service level is hard to achieve.

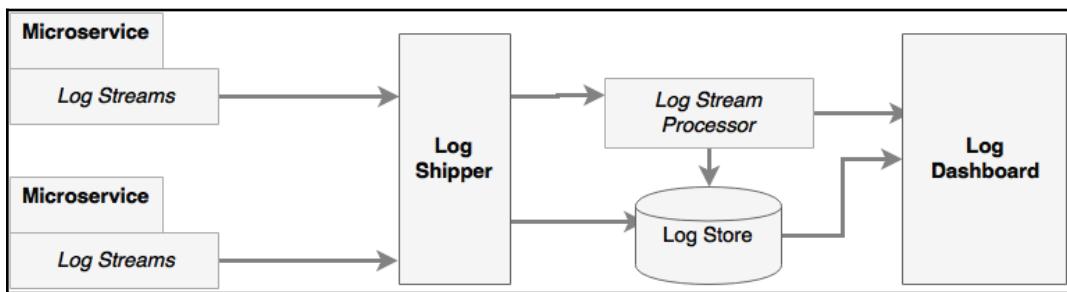
Centralized logging solution

In order to address the earlier stated challenges, traditional logging solutions require serious rethinking. The new logging solution, in addition to addressing the preceding challenges, is also expected to support the capabilities summarized here:

- Ability to collect all log messages and run analytics on top of the log messages
- Ability to correlate and track transactions end-to-end
- Ability to keep log information for longer time periods for trending and forecasting
- Ability to eliminate dependency on the local disk system
- Ability to aggregate log information coming from multiple sources, such as network devices, operating system, microservices, and so on

The solution to these problems is to centrally store and analyze all log messages, irrespective of the source of the log. The fundamental principle employed in the new logging solution is to detach log storage and processes from the service execution environments. Big data solutions are better suited for storing and processing a large amount of log messages more effectively than storing and processing them in the microservice execution environments.

In the centralized logging solution, log messages will be shipped from the execution environment to a central big data store. Log analysis and processing will be handled using big data solutions.



As shown in the preceding logical diagram, there are a number of components in the centralized logging solution. These are explained as follows:

- **Log streams:** These are streams of log messages coming out of the source systems. The source system can be microservices, other applications, or even network devices. In typical Java-based systems, these are equivalent to streaming the Log4j log messages.
- **Log shippers:** These are responsible for collecting the log messages coming from different sources or endpoints. The log shippers then send these messages to another set of endpoints, such as writing to a database, pushing to a dashboard, or sending it to a stream processing endpoint for further real-time processing.
- **Log store:** This is the place where all log messages will be stored for real-time analysis, trending, and so on. Typically, the log store will be a NoSQL database, such as HDFS, capable of handling large data volumes.
- **Log stream processor:** This is capable of analyzing real-time log events for quick decision making. Stream processors take actions such as sending information to a dashboard, sending alerts, and so on. In the case of self-healing systems, stream processors can even take action to correct the problems.

- **Log dashboard:** This dashboard is a single pane of glass for displaying log analysis results, such as graphs and charts. These dashboards are meant for operational and management staff.

The benefits of this centralized approach is that there are no local IOs or blocking disk writes. It is also not using the local machine's disk space. This architecture is fundamentally similar to Lambda architecture for big data processing.



Follow this link to read more on Lambda architecture:

<http://lambda-architecture.net>

It is important to have each log message, a context, message and a correlation ID. The context typically will have the timestamp, IP address, user information, process details (such as service, class, functions), log type, classification, and so on. The message will be plain and simple free text information. The correlation ID will be used to establish the link between service calls so that calls spanning across microservices can be traced.

Selection of logging solutions

There are a number of options available for implementing a centralized logging solution. These solutions use different approaches, architectures, and technologies. It is important to understand the capabilities required and select the right solution that meets the needs.

Cloud services

There are a number of cloud logging services available as SaaS solution. **Loggly** is one of the most popular cloud-based logging service. The Spring Boot microservices can use the Loggly's Log4j and Logback appenders to directly stream log messages into the Loggly service.

If the application or service is deployed in AWS, the AWS **CloudTrail** can be integrated with Loggly for log analysis.

Papertrial, **Logsene**, **Sumo Logic**, **Google Cloud Logging**, and **Logentries** are examples of other cloud-based logging solutions. Some of the tools in the **Security Operations Center (SOC)** are also qualified for centralized log management.

The cloud logging services take away the overhead of managing complex infrastructures and large storage solutions by providing them as simple to integrate services. However, latency is one of the key factors to be considered when selecting cloud logging as a service.

Off-the-shelf solutions

There are many purpose-built tools to provide end-to-end log management capabilities that are installable locally on an on-premise data center or in the cloud.

Graylog is one of the popular open source log management solutions. It uses Elasticsearch for log storage and MongoDB as a metadata store. It also uses GELF libraries for Log4j log streaming.

Splunk is one of the popular commercial tools available for log management and analysis. It uses the log file shipping approach compared to log streaming used by other solutions for collecting logs.

Best of the breed integration

The last approach is to pick and choose the best of the breed components and build a custom logging solution.

Log shippers

There are log shippers that can be combined with other tools to build an end-to-end log management solution. The capabilities differ between different log shipping tools.

Logstash is a powerful data pipeline tool that can be used for collecting and shipping log files. It acts as a broker that provides a mechanism to accept streaming data from different sources and sinks them to different destinations. Log4j and Logback appenders can also be used to send log messages directly from Spring Boot microservices to Logstash. The other end of the Logstash will be connected to Elasticsearch, HDFS, or any other databases.

Fluentd is another tool that is very similar to Logstash. Logspout is another similar tool to Logstash, but it is more appropriate in a Docker container-based environment.

Log stream processors

Stream processing technologies are optionally used for processing log streams on the fly. For example, if a 404 error is continuously received as a response to a particular service call, it means there is something wrong with the service. Such situations have to be handled as soon as possible. Stream processors are pretty handy in such cases, as they are capable of reacting to certain streams of events compared to traditional reactive analysis.

A typical architecture used for stream processing is a combination of **Flume** and **Kafka** together, with either **Storm** or **Spark Streaming**. Log4j has Flume appenders that are useful for collecting log messages. These messages will be pushed into distributed Kafka message queues. The stream processors collect data from Kafka and process it on the fly before sending it to Elasticsearch and other log stores.

Spring Cloud Stream, **Spring Cloud Stream modules**, and **Spring Cloud Data Flow** can also be used to build the log stream processing.

Log storage

Real-time log messages are typically stored in Elasticsearch, which allows clients to query based on the text-based indexes. Apart from Elasticsearch, HDFS is also commonly used to store archived log messages. MongoDB or Cassandra are used to store summary data, such as monthly aggregated transaction counts. Offline log processing can be done using Hadoop map reduce programs.

Dashboards

The last piece required in the central logging solution is a dashboard. The most commonly used dashboard for log analysis is **Kibana** on top of an Elasticsearch data store. **Graphite** and **Grafana** are also used to display log analysis reports.

Custom logging implementation

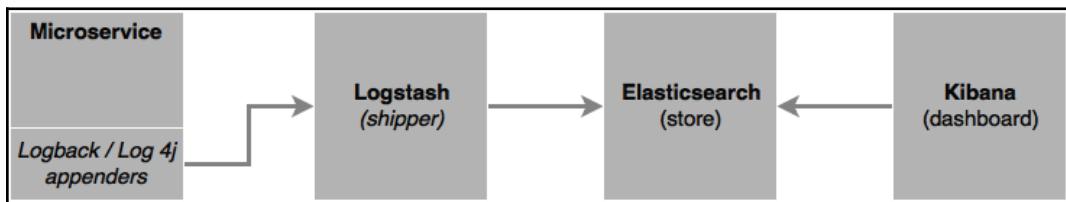
The tools mentioned in the preceding section can be leveraged to build a custom end-to-end logging solution. The most commonly used architecture for custom log management is a combination of **Logstash**, **Elasticsearch**, and **Kibana**, also known as the **ELK** stack.



The full source code of this chapter is available under the `chapter8` projects in the code files under <https://github.com/rajeshrv/Spring5Microservice>. Copy `chapter7.configserver`, `chapter7.eurekaserver`, `chapter7.search`, `chapter7.search-apigateway`, and `chapter7.website` into a new STS workspace and rename `chapter8.*`.

Note: Even though Spring Cloud Dalston SR1 officially supports Spring Boot 1.5.2.RELEASE, there are a few issues around Hystrix. In order to run the Hystrix examples, it is advised to upgrade the Spring Boot version to 1.5.4.RELEASE.

The following diagram shows the log monitoring flow:



In this section, a simple implementation of a custom logging solution using the ELK stack will be examined.

Follow these steps to implement the ELK stack for logging:

1. Download and install Elasticsearch, Kibana, and Logstash from <https://www.elastic.co>.
2. Update the Search microservice (`chapter8.search`). Review and ensure that there are some log statements in `Application.java` of the Search microservice. The log statements are nothing special but simple log statements using `slf4j` as shown in the following code snippet:

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
//other code goes here
private static final Logger logger = LoggerFactory
    .getLogger(SearchRestController.class);

//other code goes here

logger.info("Looking to load flights...");
```

```
for (Flight flight : flightRepository
```

```
    .findByOriginAndDestinationAndFlightDate
    ("NYC", "SFO", "22-JAN-18")) {
        logger.info(flight.toString());
    }
```

3. Add the Logstash dependency to integrate logback to logstash in the Search service's pom.xml:

```
<dependency>
    <groupId>net.logstash.logback</groupId>
    <artifactId>logstash-logback-encoder</artifactId>
    <version>4.6</version>
</dependency>
```

4. Override the default logback configuration. This can be done by adding a new logback.xml under src/main/resources. A sample log configuration is shown as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <include resource="org/springframework/boot/logging
        /logback/default.xml"/>
    <include resource="org/springframework/boot/logging
        /logback/console-appender.xml" />
    <appender name="stash"
        class="net.logstash.logback.appenders
        .LogstashTcpSocketAppender">
        <destination>localhost:4560</destination>
        <!-- encoder is required -->
        <encoder class="net.logstash.logback.encoder
        .LogstashEncoder" />
    </appender>
    <root level="INFO">
        <appender-ref ref="CONSOLE" />
        <appender-ref ref="stash" />
    </root>
</configuration>
```

The preceding configuration overrides the default logback configuration by adding a new TCP socket appender, which streams all log messages to a Logstash service that is listening on port 4560. It is important to add an encoder, as mentioned in the preceding configuration.

5. Create a configuration, as shown next, and store it in a `logstash.conf` file. The location of this file is irrelevant, since it will be passed as an argument when starting Logstash. This configuration will take input from the socket, listening on port 4560 and send the output to Elasticsearch running on port 9200. The `stdout` is optional and set for debugging:

```
input {  
    tcp {  
        port => 4560  
        host => localhost  
    }  
}  
output {  
    elasticsearch { hosts => ["localhost:9200"] }  
    stdout { codec => rubydebug }  
}
```

6. Run Logstash, Elasticsearch, and Kibana from their respective installation folders:

```
./bin/elasticsearch  
./bin/kibana  
./bin/logstash -f logstash.conf
```

7. Run the Search microservice. This will invoke the unit test cases and result in printing the log statements mentioned earlier. Ensure that RabbitMQ, Config Server, and Eureka servers are running.
8. Go to a browser and access Kibana:

```
http://localhost:5601
```

Go to settings and configure an index pattern, as shown in the following screenshot:

Configure an index pattern

In order to use Kibana you must configure at least one index pattern. Index patterns are used to identify the Elasticsearch index to run search and analytics against. They are also used to configure fields.

Index contains time-based events
 Use event times to create index names

Index name or pattern
 Patterns allow you to define dynamic index names using * as a wildcard. Example: logstash-*

logstash-*

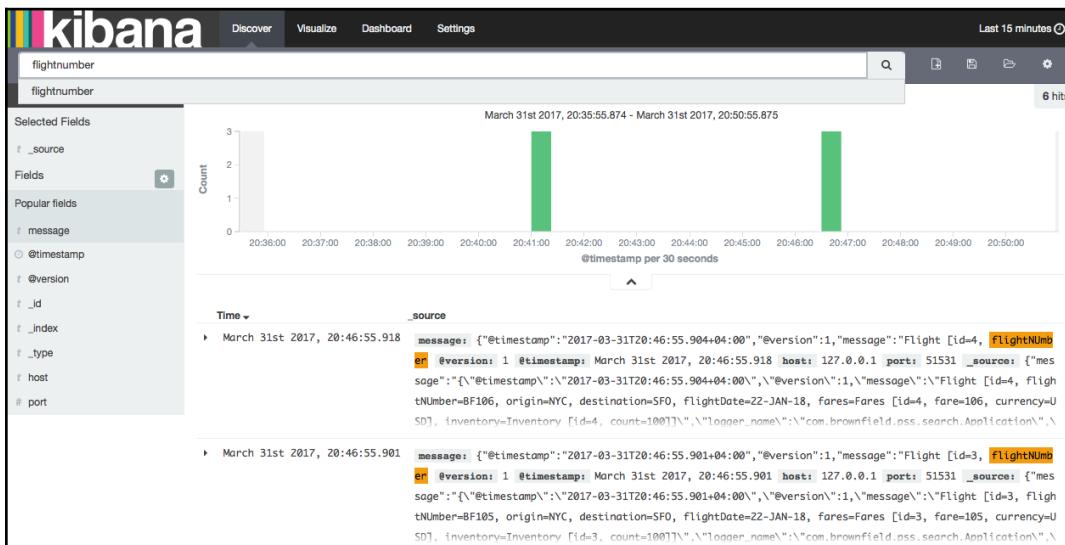
Time-field name @timestamp

Create

9. Go to discover menu to see the logs. If everything is successful, we will see the following Kibana screenshot. Note that the log messages are displayed in the Kibana screen.

Kibana provides out-of-the-box features to build summary charts and graphs using log messages.

The Kibana UI will look like the following screenshot:



Distributed tracing with Spring Cloud Sleuth

The previous section addressed the microservices distributed and fragmented the logging issue by centralizing the log data. With the central logging solution, we have all the logs in central storage. However, still, it is almost impossible to trace end-to-end transactions. In order to do end-to-end tracking, transactions spanning across microservices need to have a correlation ID.

Twitter's **Zipkin**, Cloudera's **HTrace**, and Google's **Dapper** are examples of distributed tracing systems. The Spring Cloud provides a wrapper component on top of these using the Spring Cloud Sleuth library.

Distributed tracing works with the concepts of **Span** and **Trace**. Span is a unit of work, such as **calling a service**, identified by a 64-bit span ID. A set of spans form a tree-like structure called trace. Using the trace ID, a call can be tracked end-to-end as shown in the following diagram:



As shown in the preceding diagram, **Microservice 1** calls **2**, and **2** calls **3**. In this case, as shown in the diagram, the same **Trace-id** will be passed across all microservices, which can be used to track transactions end-to-end.

In order to demonstrate this, we will use the Search API Gateway and Search microservices. A new endpoint has to be added in the Search API Gateway (`chapter8.search-apigateway`), which internally calls the Search service to return data. Without the trace ID, it is almost impossible to trace or link calls coming from a website to `search-apigateway` to the Search microservice. In this case, it only involves two to three services; whereas, in a complex environment, there can be many interdependent services.

Follow these steps to create an example using Sleuth:

1. Update Search and Search API Gateway. Before that, the Sleuth dependency has to be added to the respective pom files:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>
```

2. Add the Logstash dependency to the Search service as well as the logback configuration, as shown in the previous example.
3. The next step is to add the service name property in the logback configuration of the respective microservices:

```
<property name="spring.application.name"
  value="search-service"/>
<property name="spring.application.name"
  value="search-apigateway"/>
```

4. Add a new endpoint to the Search API Gateway, which will call the Search service, as follows. This is to demonstrate the propagation of the trace ID across multiple microservices. This new method in the gateway returns the operating hub of the airport by calling the search service. Note--the Rest Template (with @Loadbalanced) and Logger details also need to be added to the SearchAPIGateway.java class:

```
@RequestMapping("/hubongw")
String getHub(HttpServletRequest req){
    logger.info("Search Request in API gateway
        for getting Hub, forwarding to search-service ");
    String hub = restTemplate.getForObject("http://search-
        service/search/hub", String.class);
    logger.info("Response for hub received, Hub "+ hub);
    return hub;
}
```

5. Add another endpoint in the Search service, as follows:

```
@RequestMapping("/hub")
String getHub() {
    logger.info("Searching for Hub, received from
        search-apigateway ");
    return "SFO";
}
```

6. Once added, run both services. Hit the gateway's new hub on the gateway (/hubongw) endpoint using a browser. Copy and paste the following link:

<http://localhost:8095/hubongw>

As mentioned earlier, the Search API Gateway service is running on 8095 and the Search service is running on 8090.

7. Notice the console logs to see the trace ID and span IDs printed. The following print is from the Search API Gateway:

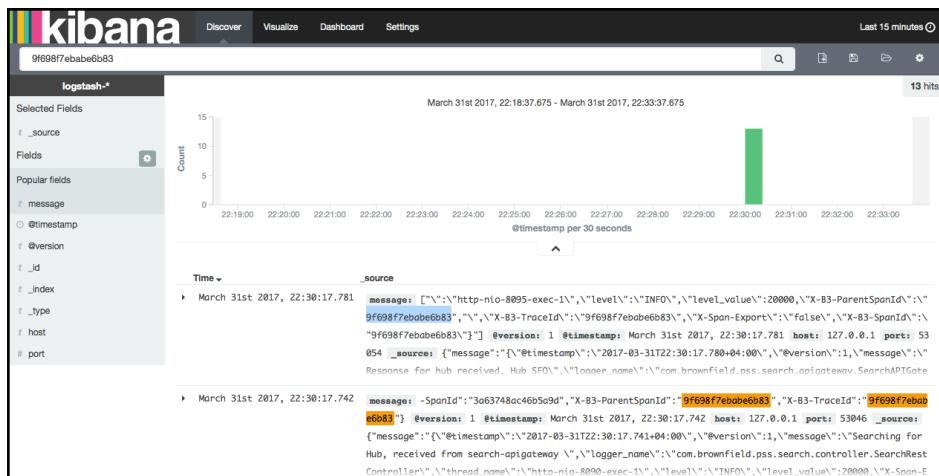
```
2017-03-31 22:30:17.780 INFO [search-apigateway,9f698f7ebabe6b83,9f698f7ebabe6b83,false] 47158 --- [nio-8095-exec-1] c.b.p.s.a.SearchAPIGatewayController: Response for hub received, Hub SFO
```

The following log is coming from the Search service:

```
2017-03-31 22:30:17.741 INFO [search-service,9f698f7ebabe6b83,3a63748ac46b5a9d,false] 47106---[nio-8090-exec-1]c.b.p.s.controller.SearchRestController : Searching for Hub, received from search-apigateway
```

Note that the trace IDs are the same in both cases.

8. Open the Kibana console and search for the trace ID using the trace ID printed in the console. In this case, it is `9f698f7ebabe6b83`. As shown in the following screenshot, with a trace ID, one can trace service calls that span across multiple services:



Monitoring microservices

Microservices are truly distributed systems with fluid deployment topology. Without a sophisticated monitoring in place, the operations team may run into trouble managing large-scale microservices. Traditional monolithic application deployments are limited to a number of known services, instances, machines, and so on. This is easier to manage as compared to a large number of microservices instances potentially running across different machines. To add more complications, these services dynamically change its topologies. The centralized logging capability only addresses part of the issue. It is important for the operations team to understand the runtime deployment topology, and also the behavior of the systems. This demands more than centralized logging can offer.

In general, application monitoring is more of a collection of metrics and aggregation and validating them against certain baseline values. If there is a service-level breach, then monitoring tools generate alerts and send to administrators. With hundreds and thousands of interconnected microservices, traditional monitoring does not really offer true value. A one-size-fits-all approach to monitoring, or monitoring everything with a single pane of glass is not easy to achieve in large-scale microservices.

One of the main objectives of microservice monitoring is to understand the behaviors of the system from the user experience point of view. This will ensure that the end-to-end behavior is consistent and in line with what is expected by users.

Monitoring challenges

Similar to the fragmented logging issue, the key challenge in monitoring microservices is that there are many moving parts in a microservice ecosystem.

Typical issues are summarized as follows:

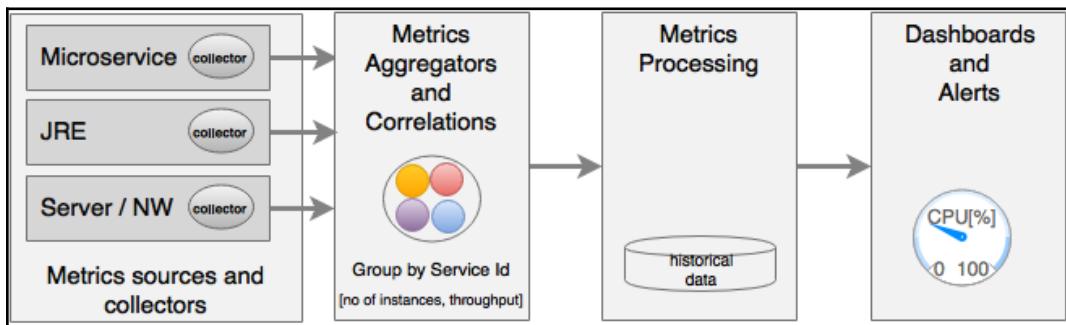
- The statistics and metrics are fragmented across many services, instances, and machines.
- Heterogeneous technologies may be used to implement microservices, making things even more complex. A single monitoring tool may not give all required monitoring options.
- Microservices deployment topologies are dynamic, making it impossible to preconfigure servers, instances, and monitoring parameters.

Many of the traditional monitoring tools are good for monitoring monolithic applications, but fall short in monitoring large-scale distributed and interlinked microservice systems. Many of the traditional monitoring systems are agent-based, preinstall agents on the target machines or application instances. This poses the following two challenges:

- If the agents require deep integration with the services or operating systems, then this will be hard to manage in a dynamic environment
- If these tools impose overheads when monitoring or instrumenting the application, they can hinder performance issues

Many traditional tools need baseline metrics. Such systems work with preset rules, such as if the CPU utilization goes above 60% and remains at that level for two minutes, then send an alert to the administrator. It is extremely hard to preconfigure these values in large internet-scale deployments.

New generation monitoring applications self learn the application behavior and set automatic threshold values. This frees up the administrators from performing this mundane task. Automated baselines are sometimes more accurate than human forecasts.



As shown in the preceding diagram, key areas of microservices monitoring are as follows:

- **Metrics sources and data collectors:** The metrics collection at the source will be done by either the server pushing metrics information to a central collector or by embedding lightweight agents to collect information. The data collectors collect monitoring metrics from different sources, such as network, physical machines, containers, software components, application, and so on. The challenge is to collect this data using auto-discovery mechanisms instead of static configurations.

This will be done by either running agents on the source machines, streaming data from the sources, or polling at regular intervals.

- **Aggregation and correlation of metrics:** The aggregation capability is required to aggregate metrics collected from different sources, such as user transaction, service, infrastructure, network, and so on. Aggregation can be challenging, as it requires some level of understanding of the applications behaviors, such as service dependencies, service grouping, and so on. In many cases, these are automatically formulated based on the metadata provided by the sources.

Generally, this will be done by an intermediary that accepts the metrics.

- **Processing metrics and actionable insights:** Once the data is aggregated, then the next step is to take measurements. Measurements are typically done by using set thresholds. In the new generation monitoring systems, these thresholds are automatically discovered. The monitoring tools then analyze the data and provide actionable insights.

These tools may use big data and stream analytics solutions.

- **Alerting, actions and dashboards:** As soon as issues are detected, they have to be notified to the relevant people or systems. Unlike traditional systems, the microservices monitoring systems should be capable of taking actions on a real-time basis. Proactive monitoring is essential to achieving self-healing. Dashboards are used to display SLAs, KPIs, and so on.

Dashboards and alerting tools are capable of handling these requirements.

Microservice monitoring is typically done with three approaches. A combination of them is really required for effective monitoring:

- **Application Performance Monitoring (APM)** (sometimes referred to as **Digital Performance Monitoring or DPM**) is more of a traditional approach of system metrics collection, processing, alerting, and rendering dashboards. These are more from the system's point of view. Application topology discovery and visualization are new capabilities implemented by many of the APM tools. The capabilities vary between different APM providers.

- **Synthetic monitoring** is a technique that is used to monitor system behavior using end-to-end transactions with a number of test scenarios in a production or production-like environment. Data will be collected to validate the system behavior and potential hotspots. Synthetic monitoring helps us understand system dependencies as well.
- **Real user monitoring (RUM)** or user experience monitoring is typically a browser-based software that records real user statistics, such as response times, availability, and service levels. With microservices, with a more frequent release cycle and dynamic topology, users experience that monitoring is more important.

Monitoring tools

There are many tools available for monitoring microservices. There are also overlaps between many of these tools. Selection of monitoring tools really depends upon the ecosystem that needs to be monitored. In most cases, more than one tool is required to monitor the overall microservice ecosystem.

The objective of this section is to familiarize a number of common microservices-friendly monitoring tools:

- **AppDynamics**, **Dynatrace** and **New Relic** are top commercial vendors in the APM space, as per Gartner magic quadrant 2015. These tools are microservice-friendly and support microservice monitoring effectively in a single console. **Ruxit**, **Datadog**, and **Dataloop** are other commercial offerings that are purpose-built for distributed systems that are essentially microservices-friendly. Multiple monitoring tools can feed data to Datadog using plugins.
- Cloud vendors come with their own monitoring tools, but, in many cases, these monitoring tools alone may not be sufficient for large-scale microservices monitoring. For instance, AWS uses **CloudWatch** and Google Cloud Platform uses **Cloud Monitoring** to collect information from various sources.
- Some of the data collecting libraries, such as **Zabbix**, **statd**, **collectd**, **jmxtrans**, and so on, operate at a lower level in collecting runtime statistics, metrics, gauges, and counters. Typically, this information will be fed into data collectors and processors, such as **Riemann**, **Datadog**, and **Librato**, or dashboards, such as **Graphite**.

- Spring Boot **Actuator** is one of the good vehicles for collecting microservices metrics, gauges, and counters, as we saw in Chapter 3, *Building Microservices with Spring Boot*. Netflix's **Servo** is a metric collector similar to Actuator. **QBit** and **Dropwizard** metrics also fall in the same category of metric collectors. All these metrics collectors need an aggregator and dashboard to facilitate full-sized monitoring.
- Monitoring through logging is popular, but a less effective approach in microservices monitoring. In this approach, as discussed in the previous section, log messages will be shipped from various sources, such as microservices, containers, networks, and so on, to a central location. Then, use the log files to trace transactions, identify hotspots, and so on. **Loggly**, **ELK**, **Splunk**, and **Trace** are candidates in this space.
- **Sensu** is a popular choice for microservice monitoring from the open source community. **Weave scope** is another tool, primarily targeting containerized deployments. SimianViz (formerly **Spigo**) is one of the purpose-built microservices, monitoring the system closely aligned with the Netflix stack. Cronitor is also another useful tool.
- **Pingdom**, **New Relic synthetic**, **Runscope**, **Catchpoint**, and so on, provide options for synthetic transaction monitoring and user experience monitoring on live systems.
- **Circonus** is classified more towards DevOps monitoring tools, but can also do microservices monitoring. **Nagios** is a popular open source monitoring tool, but it falls more into the traditional monitoring systems.
- **Prometheus** provides a time series database and visualization GUI useful for building custom monitoring tools.

Monitoring microservice dependency

When there are a large number of microservices with dependencies, it is important to have a monitoring tool that can show the dependencies between microservices. It is not a scalable approach to statically configure and manage these dependencies. There are many tools that are useful for monitoring microservice dependencies.

Mentoring tools, such as **AppDynamics**, **Dynatrace**, and **New Relic**, can draw dependencies between microservices. End-to-end transaction monitoring can also trace transaction dependencies. Other monitoring tools such as Spigo are also useful for microservices dependency management.

CMDB tools, such as **Device42**, or purpose-built tools, such as **Accordance**, are useful for managing dependency of microservices. **Vertias Risk Advisor (VRA)** is also useful for infrastructure discovery.

A custom implementation with a Graph database such as Neo4j can also be used. In this case, microservices has to be preconfigured with its direct and indirect dependencies. At the startup of the service, it publishes and cross-checks its dependencies with this Neo4j database.

Spring Cloud Hystrix for fault-tolerant microservices

This section will explore the Spring Cloud Hystrix as a library for fault-tolerant and latency-tolerant microservice implementation. The Hystrix is based on fail-fast and rapid recovery principles. If there is an issue with a service, Hystrix helps isolate the issue. It helps to fail-fast quickly by falling back to another preconfigured fallback service. It is another battle-tested library from Netflix and is based on the **Circuit Breaker** pattern.



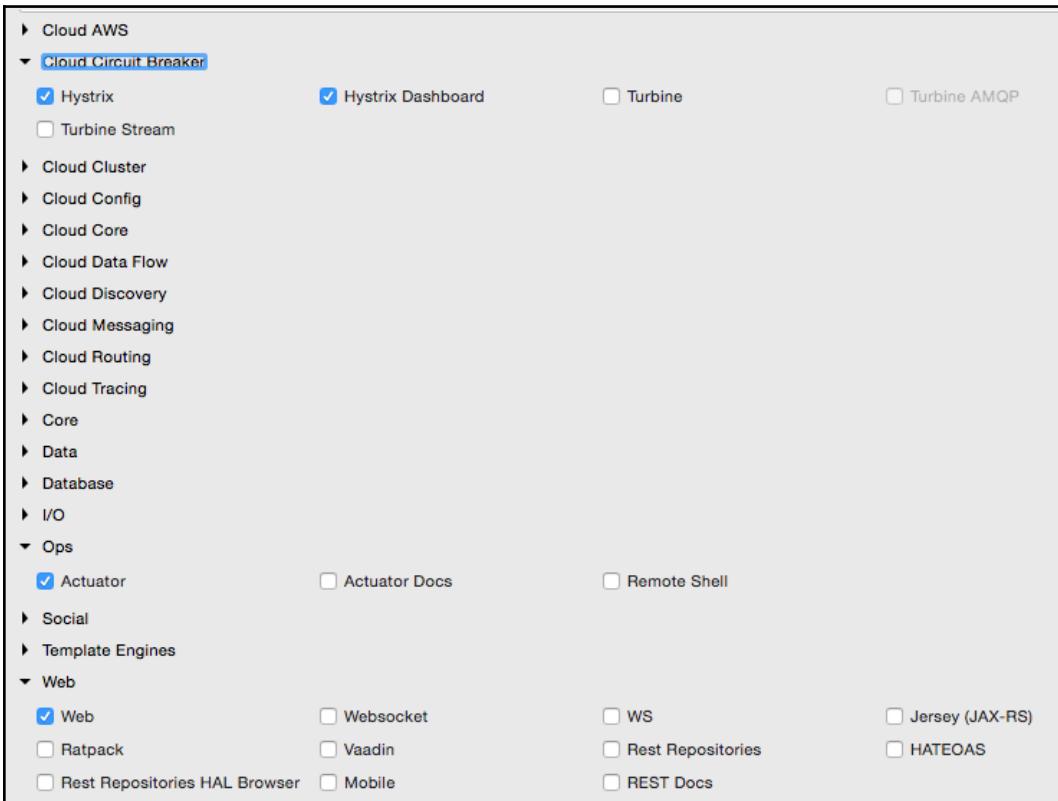
Read more about the Circuit Breaker pattern at <https://msdn.microsoft.com/en-us/library/dn589784.aspx>.

In this section, we will build a circuit breaker with the Spring Cloud Hystrix. Follow these steps to change the Search API Gateway service to integrate with the Hystrix. Update the Search API Gateway service.

Add the Hystrix dependency to the service, as follows:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-hystrix</artifactId>
</dependency>
```

If developing from scratch, select the following libraries:



In the Spring Boot Application class (`SearchAPIGateway`), add `@EnableCircuitBreaker`. This command will tell Spring Cloud Hystrix to enable circuit breaker for this application. It also exposes the `/hystrix.stream` endpoint for metrics collection.

Add a component class to the Search API Gateway service with a method; in this case; `getHub` annotated with `@HystrixCommand`. This tells Spring that this method is prone to failure. The Spring Cloud libraries wrap these methods to handle fault-tolerance and latency-tolerance by enabling circuit breaker. The `HystrixCommand` typically follows with a `fallbackMethod`. In case of failure, Hystrix automatically enables the `fallbackMethod` mentioned and diverts the traffic to the `fallbackMethod`. As shown in the following code, in this case, `getHub` will fall back to `getDefaultHub`:

```
@Component
class SearchAPIGatewayComponent {
    @LoadBalanced
    @Autowired
    RestTemplate restTemplate;
    @HystrixCommand(fallbackMethod = "getDefaultValue")
    public String getHub() {
        String hub = restTemplate
            .getForObject("http://search-service/search/hub",
            String.class);
        return hub;
    }

    public String getDefaultValue() {
        return "Possibly SFO";
    }
}
```

The `getHub` method of `SearchAPIGatewayController` calls the `getHub` method of `SearchAPIGatewayComponent`:

```
@RequestMapping("/hubongw")
String getHub(){
    logger.info("Search Request in API gateway for getting Hub,
        forwarding to search-service ");
    return component.getHub();
}
```

The last part of this exercise is to build a Hystrix dashboard. For this, build another Spring Boot application. Include **Hystrix**, **Hystrix Dashboard**, and **Actuator** when building this application.

In the Spring Boot Application class, add the `@EnableHystrixDashboard` annotation.

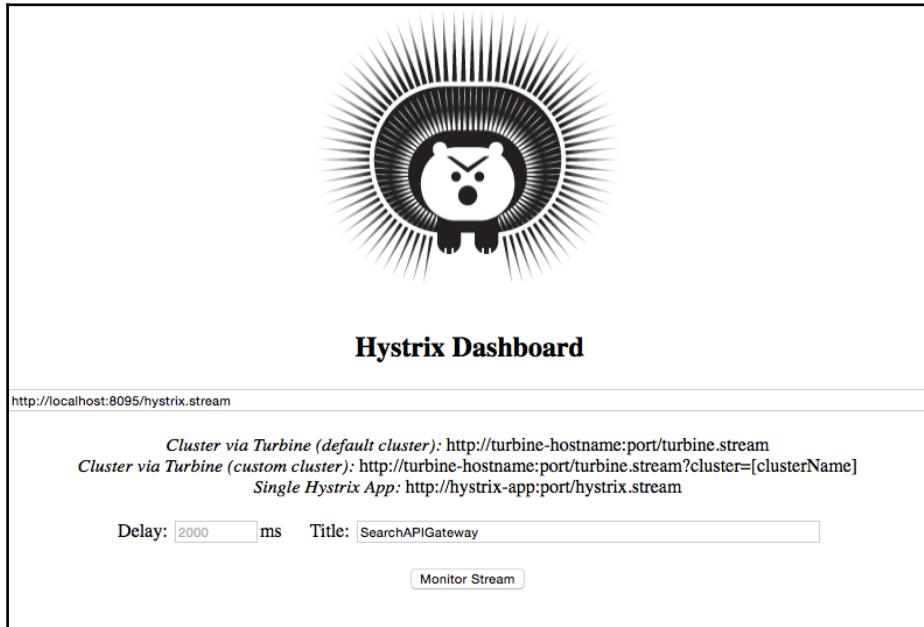
Start the Search service, Search API Gateway, and Hystrix Dashboard applications. Point the browser to the Hystrix dashboard application's URL. In this example, the Hystrix dashboard is started on port 9999.

Open the following URL:

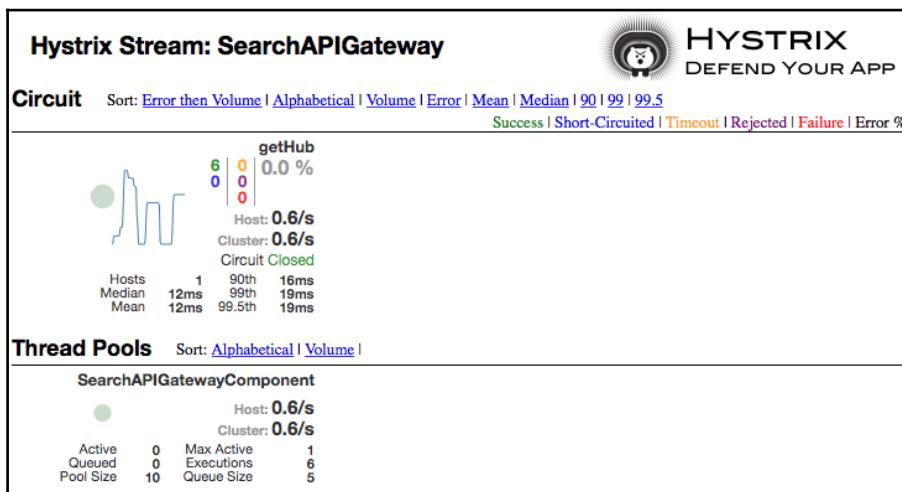
<http://localhost:9999/hystrix>

A screen as shown in the following screenshot will be displayed. In the **Hystrix Dashboard**, enter the URL of the service to be monitored.

In this case, the Search API Gateway is running on the 8095 port. Hence the `hystrix.stream` URL will be `http://localhost:8095/hystrix.stream`:



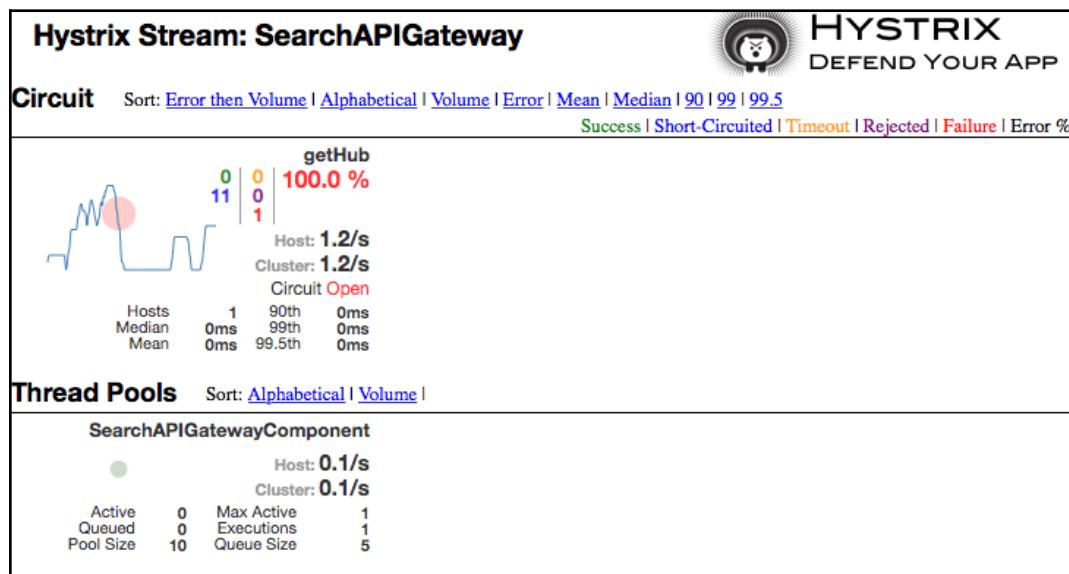
The Hystrix dashboard will be displayed as follows:



Note that at least one transaction has to be executed to see the display. This can be done by hitting `http://localhost:8095/hubongw`.

Create a failure scenario by shutting down the Search service. Note that the fallback method will be called when hitting the following URL:
`http://localhost:8095/hubongw`

If there are continuous failures, then the circuit status will be changed to open. This can be done by hitting the preceding link a number of times. In the open state, the original service will no longer be checked. The Hystrix dashboard will show the status of the circuit as **Open**, as shown in the following screenshot. Once the circuit is opened, periodically, the system will check for the original service status for recovery. When the original service is back, the circuit breaker falls back to the original service and the status will be set to **Closed**:



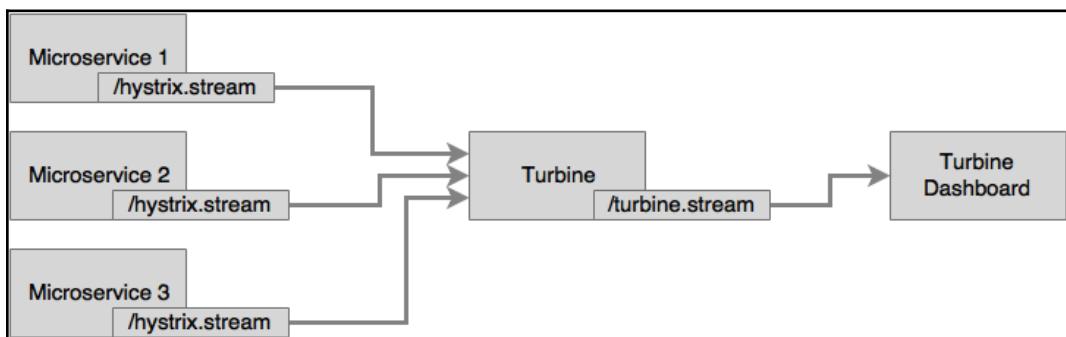
The following Hystrix Wiki URL shows the meaning of each of these parameters:

<https://github.com/Netflix/Hystrix/wiki/Dashboard>

Aggregate Hystrix streams with Turbine

In the previous example, the `/hystrix.stream` endpoint of our microservice was given in the Hystrix dashboard. Hystrix dashboard can only monitor one microservice at a time. If there are many microservices, then the Hystrix dashboard pointing to the service has to be changed every time when switching the microservices to the monitor. Looking into one instance at a time is tedious, especially when there are many instances of a microservice or multiple microservices.

We have to have a mechanism to aggregate data coming from multiple `/hystrix.stream` instances, and consolidate them into a single dashboard view. Turbine does exactly the same. It is another server that collects the Hystrix streams from multiple instances and consolidates them into one `/turbine.stream`. Now the Hystrix dashboard can point to `/turbine.stream` to get the consolidated information. Take a look at the following diagram:



Turbine works only with different host names. Each instance has to run on separate hosts. If testing multiple services locally on the same host, update the host file (`/etc/hosts`) to simulate multiple hosts. Once done, the `bootstrap.properties` have to be configured as follows:

```
eureka.instance.hostname: localdomain2.
```

The following example showcases how to use Turbine to monitor circuit breakers across multiple instances and services. We will use the Search service and Search API Gateway in this example. Turbine internally uses Eureka to resolve service IDs that are configured for monitoring.

Follow these steps to build and execute this example.

1. The Turbine server can be created as just another Spring Boot application using Spring Boot Starter. Select **Turbine** to include the Turbine libraries.
2. Once the application is created, add `@EnableTurbine` to the main Spring Boot Application class. In this example, both the Turbine and the Hystrix dashboard are configured to run on the same Spring Boot Application. This is possible by adding the following annotations to the newly created Turbine application:

```
@EnableTurbine  
@EnableHystrixDashboard  
@SpringBootApplication  
public class TurbineServerApplication {
```

3. Add the following configuration to the `yaml` or property file to point to instances that we are interested in `monitor.spring`:

```
application:  
  name : turbineserver  
turbine:  
  clusterNameExpression: new String('default')  
  appConfig : search-service,search-apigateway  
  server:  
    port: 9090  
    eureka:  
      client:  
        serviceUrl:  
          defaultZone: http://localhost:8761/eureka/
```

4. The preceding configuration instructs the Turbine server to look up the Eureka server to resolve the `search-service` and `search-apigateway` services. The `search-service` and `search-apigateway` services are the service IDs used to register services with Eureka. Turbine will use these names to resolve the actual service host and port by checking with the Eureka server. It then uses this information to read `/hystrix.stream` from each of these instances. Turbine then reads all individual Hystrix streams, aggregates all of them together, and exposes them under the Turbine server's `/turbine.stream` URL.

The cluster name expression points to the default cluster, since there is no explicit cluster configurations done in this example. If clusters are manually configured, then the following configuration has to be used:

```
turbine:  
  aggregator:
```

```
clusterConfig: [comma separated cluster names]
```

5. Change the Search service and SearchComponent to add another circuit breaker:

```
@HystrixCommand(fallbackMethod = "searchFallback")  
public List<Flight> search(SearchQuery query) {
```

6. Also add @EnableCircuitBreaker to the main class in the Search service. In this example, we will run two instances of search-apigateway--One on localdomain1:8095 and another one on localdomain2:8096. We will also run one instance of search-service on localdomain1:8090.
7. Run the microservices with command-line overrides to manage different host addresses, as follows:

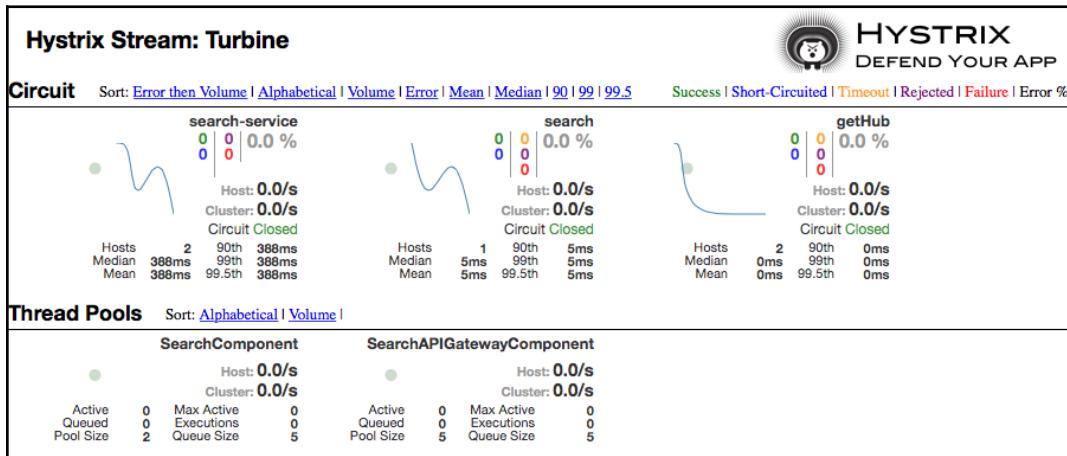
```
java -jar -Dserver.port=8096 -Deureka.instance.hostname=localdomain2 -Dserver.address=localdomain2 target/search-apigateway-1.0.jar
```

```
java -jar -Dserver.port=8095 -Deureka.instance.hostname=localdomain1 -Dserver.address=localdomain1 target/search-apigateway-1.0.jar
```

```
java -jar -Dserver.port=8090 -Deureka.instance.hostname=localdomain1 -Dserver.address=localdomain1 target/search-1.0.jar
```

8. Open the Hystrix dashboard by pointing the browser to the following URL:
<http://localhost:9090/hystrix>
9. Instead of giving /hystrix.stream, this time, we will point to /turbine.stream. In this example, the Turbine stream is running on 9090. Hence, the URL to be given in the Hystrix dashboard is as follows:
<http://localhost:9090/turbine.stream>
10. Fire a few transactions by opening the browser window and hitting
<http://localhost:8095/hubongw> and
<http://localhost:8096/hubongw>.
11. Once this is done, the dashboard page will show the getHub service.
12. Run chapter8.website. Execute the search transaction using the following website:
<http://localhost:8001>

13. After executing the preceding search, the dashboard page will show `search-service` as well. This is shown in the following screenshot:



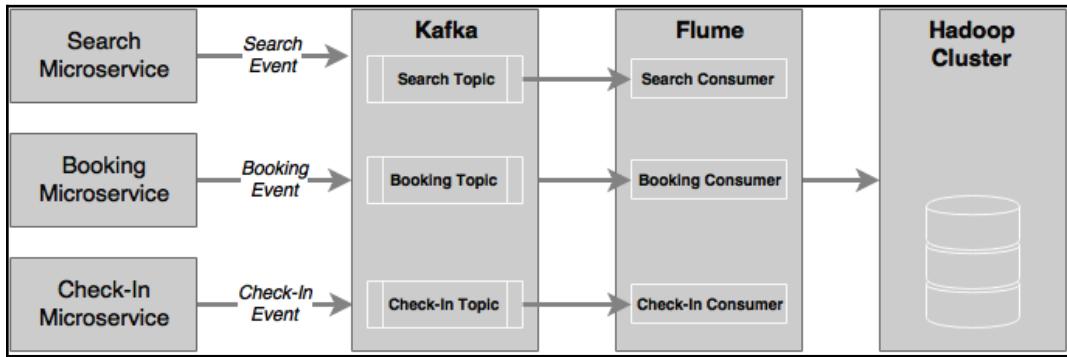
As we can see in the dashboard, `search-service` and `getHub` is coming from the Search API Gateway. Since we have two instances of the Search API Gateway, `getHub` is coming from two hosts, indicated by **Hosts 2**. The search is coming from the Search microservice. Data has been provided by the two components we created--`SearchComponent` in Search microservice and the `SearchAPIGateway` component in the Search API Gateway microservices.

Data analysis using Data Lake

Just like the scenario of fragmented logs and monitoring, fragmented data is another challenge in microservice architecture. Fragmented data poses challenges in data analytics. This data may be used for simple business event monitoring, data auditing, or even for deriving business intelligence out of the data.

Data Lake or a data hub is an ideal solution to handle such scenarios. The event-sourced architecture pattern is generally used to share state and state changes as events with an external data store. When there is a state change, microservices publish the state change as events. Interested parties may subscribe to these events and process them based on their requirements. A central event store can also subscribe to these events and store them in a big data store for further analysis.

One of the commonly followed architectures for such data handling is shown in the following diagram:



The state change events generated from the microservices, in our case, **Search**, **Booking**, and **Check-In** events, are pushed to a distributed high performance messaging system such as **Kafka**. A data ingestion, such as **Flume**, can subscribe these events and update them to an **HDFS** cluster. In some cases, these messages will be processed in real time by **Spark Streaming**. To handle heterogeneous sources of events, **Flume** can also be used between event sources and **Kafka**.

Spring Cloud Streams, **Spring Cloud Streams modules**, and **Spring Cloud Data Flow** are also useful as an alternative for high velocity data ingestion.

Summary

In this chapter, we learned about the challenges around logging and monitoring when dealing with internet-scale microservices.

We explored the various solutions for centralized logging and also learned how to implement a custom centralized logging using **Elasticsearch**, **Logstash**, and **Kibana** (**ELK**). In order to understand distributed tracing, we upgraded the BrownField microservices using the Spring Cloud Sleuth.

In the second half of this chapter, we went deeper into the capabilities required for microservices monitoring solutions and different approaches for monitoring. Subsequently, we examined a number of tools available for microservices monitoring.

The BrownField microservices were further enhanced with the Spring Cloud Hystrix and Turbine for monitoring latencies and failures in inter-service communications. The examples also demonstrated how to use the Circuit Breaker pattern to fall back to another service in case of failures.

Finally, we also touched upon the importance of Data Lake and how to integrate a Data Lake architecture in a microservice context.

Microservice management is another important challenge we have to tackle when dealing with large-scale microservices deployments. The next chapter will explore how containers can help in simplifying microservice management.

20

Scaling Dockerized Microservices with Mesos and Marathon

In order to leverage full power of a cloud-like environment, the dockerized microservice instances should also be capable of scaling out and shrinking automatically, based on the traffic patterns. However, this could lead to another problem. Once there are many microservices, it is not easy to manually manage thousands of dockerized microservices. It is essential to have an infrastructure abstraction layer and a strong container orchestration platform to successfully manage internet-scale dockerized microservice deployments.

This chapter will explain the basic scaling approaches and the need and use of Mesos and Marathon as an infrastructure-orchestration layer to achieve optimized resource usage in a cloud-like environment when deploying microservices at scale. This chapter will also provide a step-by-step approach to setting up Mesos and Marathon in a cloud environment. Finally, this chapter will demonstrate how to manage dockerized microservices into the Mesos and Marathon environment.

By the end of this chapter, we will have learned about the following:

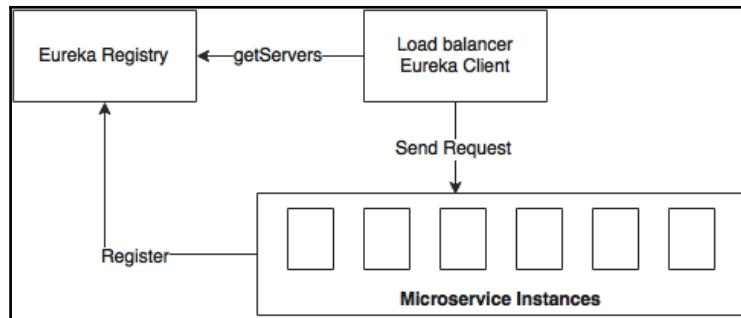
- Options to scale containerized Spring Boot microservices
- The need to have an abstraction layer and a container orchestration software
- An understanding Mesos and Marathon from the context of microservices
- How to manage dockerized BrownField Airline's PSS microservices with Mesos and Marathon

Scaling microservices

At the end of [Chapter 7, Scale Microservices with Spring Cloud Components](#), we discussed two options for scaling either using Spring Cloud components or dockerized microservices using Mesos and Marathon. In [Chapter 7, Scale Microservices with Spring Cloud Components](#), you learned how to scale the Spring Boot microservices using the Spring Cloud components.

The two key concepts of Spring Cloud that we have implemented are self-registration and self-discovery. These two capabilities enable automated microservices deployments. With self-registration, microservices can automatically advertise the service availability by registering service metadata to a central service registry as soon as the instances are ready to accept traffic. Once microservices are registered, consumers can consume newly registered services from the very next moment by discovering service instances using the registry service. In this model, registry is at the heart of this automation.

The following diagram shows the Spring Cloud method for scaling microservices:



In this chapter, we will focus on the second approach, scaling dockerized microservices using Mesos and Marathon. It also provides us with one additional capability that you didn't learn about in [Chapter 7, Scale Microservices with Spring Cloud Components](#). When there is a need for an additional microservice instance, a manual task is required to kick off a new instance. Similarly, when there is not enough traffic, there should be an option to turn off unused instances. In an ideal scenario, the start and stop of microservices instances also require automation. This is especially relevant when services are running on a pay as per usage cloud environment.

Understanding autoscaling

Autoscaling is an approach to automatically scale out instances based on the resource usage to meet agreed SLAs by replicating the services to be scaled.

The system automatically detects an increase in traffic, spins up additional instances, and makes them available for traffic handling. Similarly, when the traffic volumes go down, the system automatically detects and reduces the number of instances by taking active instances back from the service. It is also required to ensure that there is a set number of instances always up and running. In addition to this, the physical or virtual machines also need a mechanism to automatically provision machines. The latter part is much more easy to handle using APIs provided by different cloud providers.

Autoscaling can be done by considering different parameters and thresholds. Some of them are easy to handle whereas some of them are complex to handle. The following are the points that summarizes some of the commonly followed approaches:

- **Scale with resource constraints:** This approach is based on real-time service metrics collected through monitoring mechanisms. Generally, the resource scaling approach makes decisions based on the CPU, memory, or the disk of machines. It can also be done by looking at the statistics collected on the service instances itself, such as heap memory usage.
- **Scale during specific time periods:** Time-based scaling is an approach to scale services based on certain periods of the day, month, or year to handle seasonal or business peaks. For example, some services may experience higher number of transactions during office hours, and considerably less number of transactions outside office hours. In this case, during the day time, services autoscale to meet the demand and automatically downsizes during the off office hours.
- **Scale based on message queue length:** This is particularly useful when the microservices are based on asynchronous messaging. In this approach, new consumers will be automatically added when the messages in the queue goes beyond certain limits.
- **Scale based on business parameters:** In this case, adding instances will be based on certain business parameters. For example, spinning up a new instance just before handling sales, closing transactions. As soon as the monitoring service receives a preconfigured business event, such as sales closing minus 1 hour, a new instance will be brought up in anticipation of large volumes of transactions. This will provide fine grained controls on scaling based on business rules.

- **Predictive autoscaling:** This is a new paradigm of autoscaling, which is different from the traditional real-time metrics-based autoscaling. A prediction engine will take multiple inputs, such as historical information, current trends, and more, to predict possible traffic patterns. Autoscaling will be done based on these predictions. Predictive autoscaling helps in avoiding hardcoded rules and time windows. Instead, the system can automatically predict such time windows. In more sophisticated deployments, the predictive analysis may use cognitive computing mechanisms to predict autoscaling.

The missing pieces

In order to achieve autoscaling as mentioned previously, it requires a lot of scripting at the operating system level. Docker is a good step toward achieving this as it provides a uniform way of handling the containers, irrespective of the technologies used by the microservices. It also helped us in isolating microservices to avoid resource stealing by nosy neighbors.

However, Docker and scripting only address the issues partially. In the context of large-scale Docker deployments, some of the key questions to be answered are as follows:

- How do we manage thousands of containers?
- How do we monitor them?
- How do we apply rules and constraints when deploying artifacts?
- How do we ensure that we utilize containers properly to gain resource efficiency?
- How do we ensure that at least a certain number of minimal instances are running at any point in time?
- How do we ensure that dependent services are up and running?
- How do we do rolling upgrades and graceful migrations?
- How do we rollback faulty deployments?

All these preceding questions point to the need of having a solution to address the following two key capabilities:

- A container abstraction layer that provides a uniform abstraction over many physical or virtual machines
- A container orchestration and init system to manage deployments intelligently on top of the cluster abstraction

The rest of this chapter focuses on addressing these two points.

Container orchestration

Container orchestration tools provide a layer of abstraction for developers and infrastructure teams to deal with large-scale containerized deployments. The features offered by the container orchestration tools vary between providers. However, common denominators are provision, discovery, resource management, monitoring, and deployments.

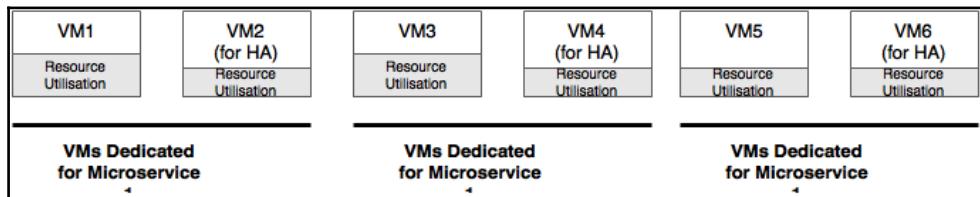
Why is container orchestration is important

Since microservices break applications into different micro applications, many developers request more server nodes for deployment. In order to manage microservices properly, developers tend to deploy one microservice per VM, which further drives down the resource utilization. In many cases, this results in over-allocation of CPUs and memory.

In many deployments, the high availability requirements of microservices force engineers to add more and more service instances for redundancy. In reality, although it provides the required high availability, this will result in under-utilized server instances.

In general, microservices deployment requires more infrastructure compared to monolithic application deployments. Due to the increase in cost of the infrastructure, many organizations fail to see the value of microservices.

The following diagram shows dedicated VMs for each microservices:



In order to address the issue stated in the preceding image, we need a tool that is capable of the following:

- Automating a number of activities such as allocation of containers to infrastructure efficiently, which are transparent to the developers and administrators
- Providing a layer of abstraction for the developers so that they can deploy their application against a data center without knowing which machine is to be used for hosting their applications
- Setting rules or constraints against deployment artifacts
- Offering higher levels of agility, with minimal management overheads for developers and administrators, perhaps with minimal human interactions
- Building, deploying, and managing applications cost effectively by driving maximum utilization of the available resources

Containers solve an important issue in this context. Any tools that we select with these capabilities can handle containers in a uniform way, irrespective of the underlying microservice technologies.

What does container orchestration do?

Typical container orchestration tools help virtualize a set of machines and manage them as a single cluster. The container orchestration tools also help move the workload or containers across machines transparent to the consumer. Technology evangelists and practitioners use different terminologies, such as container orchestration, cluster management, data center virtualization, container schedulers, container life cycle management, data center operating system, and so on.

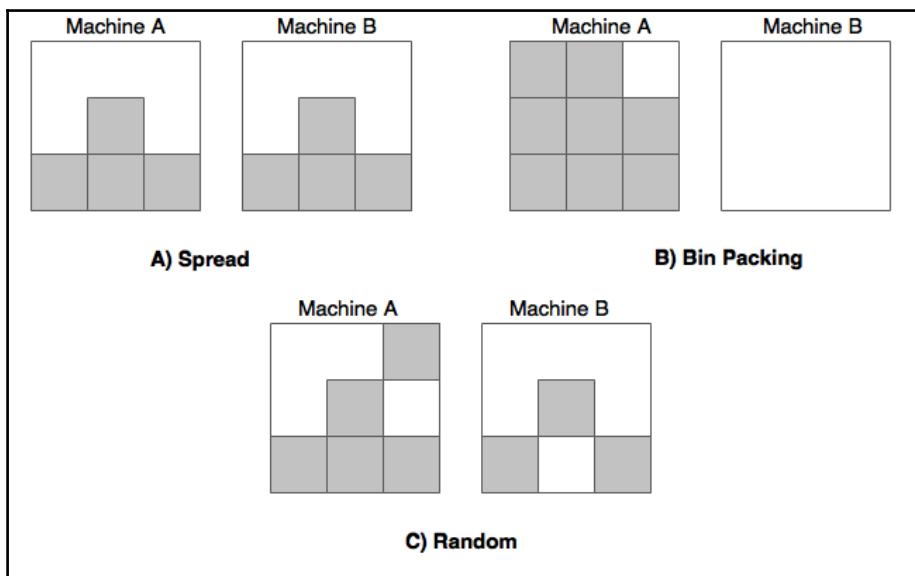
Many of these tools are currently supporting both Docker-based containers as well as non-containerized binary artifact deployments, such as the standalone Spring Boot application. The fundamental function for these container orchestration tools are to abstract the actual server instance from the application developers and administrators.

Container orchestration tools help self-service and provisioning of infrastructure rather than requesting the infrastructure teams to allocate the required machines with a predefined specification. In this automated container orchestration approach, machines are no longer provisioned upfront and preallocated to the applications. Some of the container orchestration tools also help virtualize the data centers across many heterogeneous machines, or even across data centers, and create an elastic private cloud-like infrastructure. There is no standard reference model for container orchestration tools. Therefore, the capabilities vary between vendors.

Some of the key capabilities of the container orchestration software are summarized as follows:

- **Cluster management:** This manages a cluster of VMs and physical machines as a single large machine. These machines could be heterogeneous in terms of resource capabilities, but, by and large, machines with Linux as the operating system. These virtual clusters can be formed on cloud, on premises, or a combination of both.
- **Deployments:** These handle automatic deployments of applications and containers with a large set of machines. It supports multiple versions of the application containers, and also support rolling upgrades across a large number of cluster machines. These tools are also capable of handling a rollback of faulty promotes.
- **Scalability:** This handles automatic and manual scalability of application instances as and when required with optimized utilization as a primary goal.
- **Health:** This manages the health of the cluster, nodes, and applications. It removes faulty machines and application instances from the cluster.
- **Infrastructure abstraction:** This abstracts the developers from the actual machine where the applications are deployed. The developers need not worry about the machines, capacity, and so on. It is entirely the container orchestration software's decision to see how to schedule and run the applications. These tools also abstract the machine details, their capacity, utilization, and location from the developers. For application owners, these are equivalent to a single large machine with almost unlimited capacity.
- **Resource optimizations:** The inherent behavior of these tools is to allocate the container workloads across a set of available machines in an efficient way, thereby reducing the cost of ownership. Simple to extremely complicated algorithms can be used effectively to improve utilization.

- **Resource allocations:** These allocate servers based on resource availability and constraints set by the application developers. The resource allocation will be based on these constraints, affinity rules, port requirements, application dependencies, health, and so on.
- **Service availability:** This ensures that the services are up and running somewhere in the cluster. In case of a machine failure, container orchestration automatically handle failures by restarting those services on some other machine in the cluster.
- **Agility:** Agility tools are capable of quickly allocating workloads to available resources or move the workload across machines if there is change in resource requirements. Also, constraints can be set to realign resources based on business criticality, business priority, and more.
- **Isolation:** Some of these tools provide resource isolation out-of-the-box. Hence, even if the application is not containerized, resource isolation can be achieved.



A variety of algorithms are used for resource allocation ranging from simple algorithms to complex algorithms with machine learning and artificial intelligence. The common algorithms used are **Random**, **Bin Packing**, and **Spread**. Constraints set against applications will override the default algorithms based on resource availability.

The preceding diagram shows how these algorithms fill the available machines with deployments. In this case, it is demonstrated with two machines.

Three common strategies of resource allocation are explained as follows:

- **Spread**: This equally distributes the allocation of workloads across available machines., which is shown in diagram A.
- **Bin packing**: This tries to fill machine by machine and ensure the maximum utilization of machines. Bin packing is especially good when using cloud services in a pay as you use style. This is shown in diagram B.
- **Random**: This algorithm randomly chooses machines and deploys containers on randomly selected machines, which is shown in diagram C.

There are possibilities of using cognitive computing algorithms such as machine learning and collaborative filtering to improve efficiency. Techniques such as **oversubscriptions** allow for better utilization of resources by utilizing under-utilized resources allocated for high priority tasks, including revenue generating services for best effort tasks such as analytics, video, image processing, and more.

Relationship with microservices

The infrastructure for microservices, if not properly provisioned, can easily result in over-sized infrastructures, and, essentially, higher cost of ownership. As discussed in the previous sections, a cloud-like environment with a container orchestration tool is essential to realize the cost benefits when dealing with large-scale microservices.

The Spring Boot microservices turbo, charged with the Spring Cloud project, is the ideal candidate workload to leverage container orchestration tools. Since the Spring Cloud based microservices are location unaware, these services can be deployed anywhere in the cluster. Whenever services come up, it automatically registers to the service registry and advertises its availability. On the other hand, consumers always look for the registry to discover available service instances. This way the application supports a full fluid structure without preassuming deployment topology. With Docker, we are able to abstract the runtime so that the services could run on any Linux-based environments.

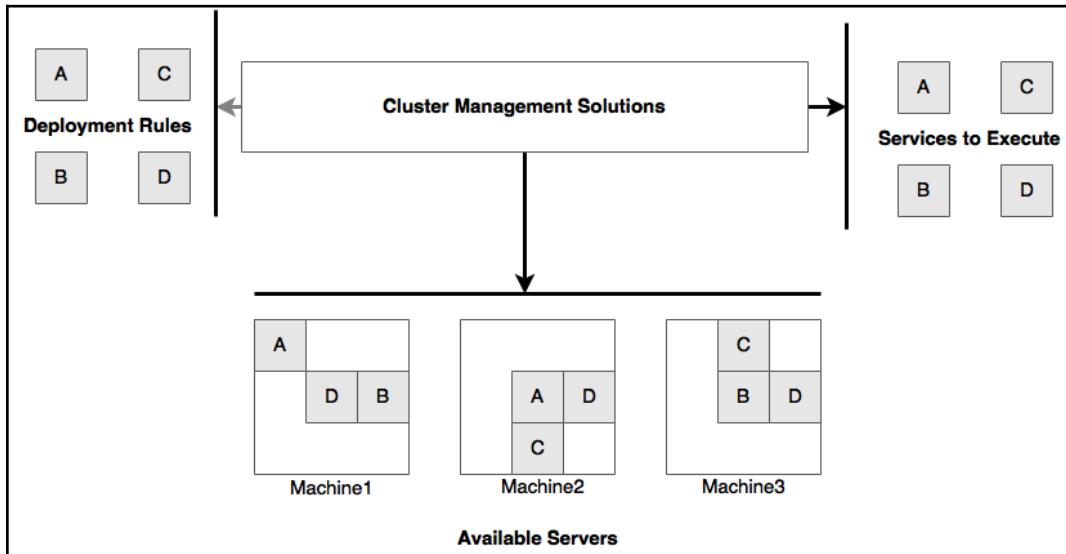
Relationship with virtualization

The container orchestration solutions are different from server virtualization solutions in many aspects. Container orchestration solutions run on top of the VMs or physical machines as an application component.

Container orchestration solutions

There are many container orchestration software tools available. It is unfair to do an apple to apple comparison between them. Even though there are no one-to-one components, there are many areas of overlap in capabilities between them. In many situations, organizations use a combination of one or more of these tools to fulfill their requirements.

The following diagram shows the position of container orchestration tools from the microservices context:



As shown in the preceding diagram, the container management or orchestration tools take a set of deployable artifacts in the form of containers (**Services to Execute**) and a set of constraints or rules as deployment descriptors then find the most optimal compute infrastructure for deployment which are available and fragmented across multiple machines.

In this section, we will explore some of the popular container orchestration solutions available in the market.

Docker Swarm

Docker Swarm is Docker's native container orchestration solution. Swarm provides native and deeper integration with Docker and exposes APIs that are compatible with Docker's remote APIs. It logically groups a pool of Docker hosts and manages them as a large single Docker virtual host. Instead of application administrators and developers deciding on which host the container is to be deployed, this decision making will be delegated to the Docker Swarm. It will decide which host to be used based on bin packing and spread algorithms.

Since the Docker Swarm is based on Docker's remote APIs, the learning curve for existing Docker users is much less compared to any other container orchestration tools. However, the Docker Swarm is a relatively new product in the market and it only supports the Docker containers.

Docker Swarm works with the concept of **manager** and **nodes**. The manager is the single point for administrations to interact and schedule the Docker containers for execution. The nodes are where the Docker containers are deployed and run.

Kubernetes

Kubernetes (k8s) is coming from Google's engineering, is written in Go language, and is battle tested for large-scale deployments at Google. Similar to Swarm, Kubernetes helps to manage containerized applications across a cluster of nodes. It helps to automate container deployments and scheduling and scalability of containers. It supports a number of useful out-of-the-box features, such as automatic progressive rollouts, versioned deployments, and container resiliency if containers fail for some reason.

Kubernetes architecture has the concept of **master**, **nodes**, and **pods**. The master and nodes together are called a Kubernetes cluster. The master node is responsible for allocating and managing the workload across a number of nodes. Nodes are nothing but a VM or a physical machine. Nodes are further subsegmented as pods. A node can host multiple pods. One or more containers are grouped and executed inside a pod. Pods are also helpful to manage and deploy co-located services for efficiency. Kubernetes also support the concept of labels as key-value pairs to query and find containers. Labels are user-defined parameters to tag certain types of nodes that perform common types of workloads, such as frontend web servers. The services deployed on the cluster will get a single IP/DNS to access the service.

Kubernetes has out-of-the-box support for Docker; however, Kubernetes' learning curve will be more compared to Docker Swarm. Red Hat offers commercial support for Kubernetes as part of its OpenShift platform.

Apache Mesos

Mesos is an open source framework originally developed by the University of California at Berkeley and is used by Twitter at scale. Twitter used Mesos primarily to manage a large Hadoop ecosystem.

Mesos is slightly different from the previous solutions. It is more of a resource manager that relies on other frameworks to manage workload execution. It sits between the operating system and the application, providing a logical cluster of machines.

Mesos is a distributed system kernel that logically groups and virtualizes many computers to a single large machine. It is capable of grouping a number of heterogeneous resources to a uniform resource cluster on which applications can be deployed. For these reasons, Mesos is also known as a tool for building a private cloud in a data center.

Mesos has the concept of the **master** and **slave** nodes. Similar to the earlier solutions, master nodes are responsible for managing the cluster, whereas slaves run the workloads. It internally uses **ZooKeeper** for cluster coordination and storage. It also supports the concept of frameworks. These frameworks are responsible for scheduling and running non-containerized applications and containers. **Marathon**, **Chronos**, and **Aurora** are popular frameworks for the scheduling and execution of applications. Netflix's **Fenzo** is another open source Mesos framework. Interestingly, Kubernetes also can be used as a Mesos framework.

Marathon supports the Docker container, as well as the non-containerized applications. The Spring Boot can be directly configured in Marathon. Marathon provides a number of out-of-the-box capabilities, such as support application dependencies, grouping of applications for scaling and upgrading services, start and shutdown of healthy and unhealthy instances, rolling promotes, rollback failed promoted, and so on.

Mesosphere offers commercial support for Mesos and Marathon as part of their DCOS platform.

HashiCorp Nomad

Nomad is from **HashiCorp**, another container orchestration software. Nomad is a container orchestration system that abstracts lower-level machine details and their locations. It has a simpler architecture compared to other solutions explored earlier. It is also lightweight. Similar to other container orchestration solutions, it will take care of resource allocations and the execution of applications. Nomad also accepts user-specific constraints and allocates resources based on that.

Nomad has the concept of servers where all jobs are managed. One server will act as the **leader** and others will act as **followers**. It has the concept of **tasks**, which are the smallest units of work. Tasks are grouped into **task groups**. A task group will have tasks that are to be executed in the same location. One or more task groups or tasks are managed as **jobs**.

Nomad supports many workloads, including Docker out of the box. Nomad also supports across data center deployments and is region data center aware.

CoreOS Fleet

Fleet is a container orchestration system from CoreOS. Fleet runs on a lower level and works on top of the `systemd`. It can manage application dependencies and make sure that all required services are running somewhere in the cluster. If a service fails, it restarts the service on another host. Affinity and constraint rules are possible to supply when allocating resources.

Fleet has the concept of **engine** and **agents**. There will only be one engine at any point in the cluster with multiple agents. Tasks are submitted to the engine and the agents run these tasks on a cluster machine. Fleet also supports Docker out of the box.

In addition to this, **Amazon EC2 Container Services (ECS)**, **Azure Container Services (ACS)**, Cloud Foundry **Diego**, and **Google Container Engine** provide container orchestration functions as part of their respective cloud platform offerings.

Container orchestration with Mesos and Marathon

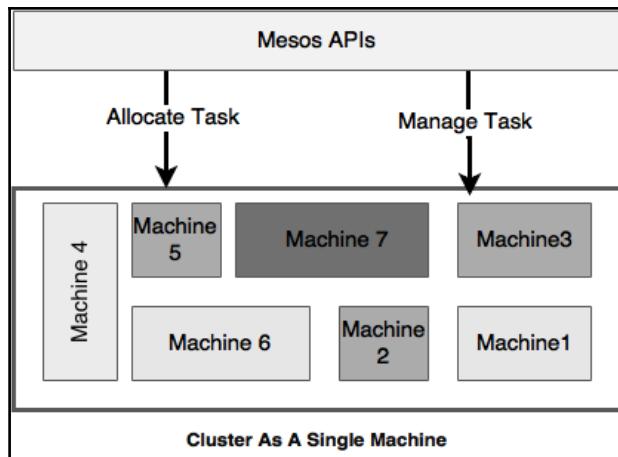
As we have seen in the previous section, there are many container orchestration solutions available. Different organizations choose different solutions to address problems based on their environments. Many organizations choose Kubernetes or Mesos with a framework such as Marathon. In most of the cases, Docker is used as a default containerization method to package and deploy workloads.

For the rest of this chapter, we will show how Mesos works with Marathon to provide the required container orchestration capability. Mesos is used by many organizations including Twitter, Airbnb, Apple, eBay, Netflix, Paypal, Uber, Yelp, and many others.

Mesos in details

Mesos can be treated as a data center kernel. Enterprise DCOS is the commercial version of Mesos supported by Mesosphere. In order to run multiple tasks on one node, Mesos uses resource isolation concepts. It relies on **cgroups** of the Linux kernel to achieve resource isolation similar to the container approach. It also supports containerized isolation using Docker. Mesos supports both, batch workloads as well as OLTP kind of workloads.

The following diagram shows Mesos logically abstracting multiple machines as a single resource cluster:

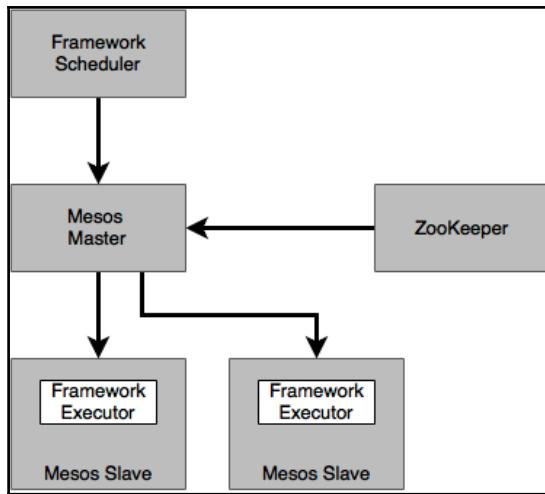


Mesos is an open source top-level Apache project under Apache License. It abstracts lower-level computing resources such as CPU, memory, and storage from the lower-level physical or virtual machines.

Before we examine why we need both Mesos and Marathon, let's understand the Mesos architecture.

Mesos architecture

The following diagram shows the simplest architecture representation of Mesos. The key components of Mesos include a **Mesos Master**, a set of slave nodes, a **ZooKeeper** service, and a Mesos framework. The Mesos framework is further subdivided into two components: a **Scheduler** and an **Executor**.



The boxes in the preceding diagram are explained as follows:

- **Master:** The **Mesos Master** is responsible for managing all Mesos slaves. It gets information on the resource availability from all slave nodes and takes responsibility of filling the resources appropriately, based on certain resource policies and constraints. The **Mesos Master** preempts available resources from all slave machines and pools them as a single large machine. Master offers resources to frameworks running on slave machines based on this resource pool.

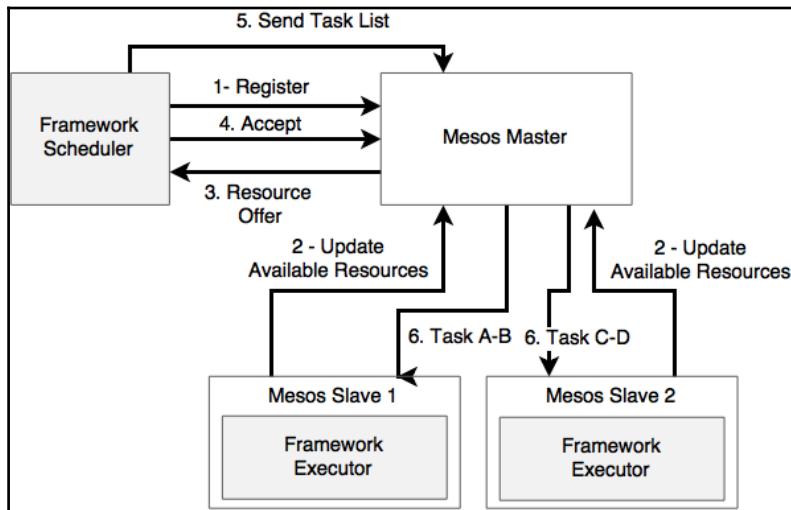
For high availability, the **Mesos Master** is supported by the Mesos master **standby** components. Even if the master is not available, existing tasks can be still executed. However, new tasks cannot be scheduled in the absence of a master node. The master standby nodes are nodes that wait for the failure of the active master and take over the master role in case of a failure. They use **ZooKeeper** for the master leader election. Minimum quorum requirements must be met in this case for leader election.

- **Slave:** The Mesos slaves are responsible for hosting task execution frameworks. Tasks are executed on the slave nodes. Mesos slaves can be started with attributes as key value pairs, such as *datacenter = X*. This will be used for constraint evaluations when deploying workloads. Slave machines share resource availability to the **Mesos Master**.
- **ZooKeeper:** This is a centralized coordination server used in Mesos to coordinate activities across the Mesos cluster. Mesos uses **ZooKeeper** for leader election in case of a **Mesos Master** failure.

- **Framework:** The Mesos framework is responsible for understanding the application constraints, accepting resource offers from the master, and, finally, running the tasks on the slave resources offered by the master. The Mesos framework consists of two components: **Framework Scheduler** and **Framework Executor**.
- The **Scheduler** is responsible for registering to Mesos and handles resource offers.
- The **Executor** runs the actual program on the Mesos slave nodes.

The framework is also responsible for enforcing certain policies and constraints. For example, a constraint can be, let's say, a minimum of 500 MB of RAM for execution.

The framework is a pluggable component and is replaceable with another framework. Its workflow is depicted in the following diagram:



The steps denoted in the preceding workflow diagram are elaborated as follows:

1. The framework registers with the **Mesos Master** and waits for the resource offers. The scheduler may have many tasks in its queue to be executed with different resource constraints (task A-D in this example). A task, in this case, is a unit of work that is scheduled. For example, a Spring Boot microservice.
2. The Mesos slave offers available resources to the **Mesos Master**. For example, the slave advertises the CPU and memory available with the slave machine.

3. **Mesos Master** creates a resource offer based on the allocation policies set and offers it to the scheduler component of the framework. The allocation policies determine to which framework the resources are to be offered and how many. The default policies can be customized by plugging additional allocation policies.
4. Scheduler framework components, based on the constraints, capabilities, and policies may accept or reject the resource offering. For example, the framework rejects the resource offer if the resources are insufficient as per the constraints and policies set.
5. If the scheduler component accepts the resource offer, it submits one or more task details to the **Mesos Master** with resource constraints per tasks. Let's say, in this example, it is ready to submit tasks **A-D**.
6. The **Mesos Master** sends this list of tasks to the slave where the resources are available. The framework executor components installed on the slave machines pick up and run these tasks.

Mesos supports a number of frameworks, such as the following:

- Marathon and Aurora for **long running** processes, such as web applications
- Hadoop, Spark, and Storm for **big data** processing
- Chronos and Jenkins for **batch scheduling**
- Cassandra and Elasticsearch for **data management**

In this chapter, we will use Marathon to run the dockerized microservices.

Marathon

Marathon is one of the Mesos framework implementations that can run both container as well as non-container execution. It is particularly designed for long running applications, such as a web server. It will ensure that the service started with Marathon will continue to be available even if the Mesos slave it is hosted on fails. This will be done by starting another instance.

Marathon is written in Scala and is highly scalable. It offers UI as well as REST APIs to interact with Marathon, such as starting, stopping, scaling, and monitoring applications.

Similar to Mesos, Marathon's high availability is achieved by running multiple Marathon instances pointing to a ZooKeeper instance. One of the Marathon instances will act as a leader and others will be in standby mode. In case the leading master fails, a leader election will take place and the next active master will be determined.

Some of the basic features of Marathon include the following:

- Setting resource constraints
- Scale up, scale down, and instance management of applications
- Application version management
 - Start and kill applications

Some of the advanced features of Marathon include the following:

- Rolling upgrades, rolling restarts, and rollbacks
- Blue-green deployments

Implementing Mesos and Marathon with DCOS

In Chapter 7, *Scale Microservices with Spring Cloud Components*, we discussed Eureka and Zuul for achieving load balancing. With container orchestration tools, load balancing and DNS services come out of the box and are much more simple to use. However, when developers need code-level control for load balancing and traffic routing, such as the business parameter based scaling scenarios mentioned earlier, Spring Cloud components may fit better.



In order to understand the technologies better, we will use Mesos and Marathon directly in this chapter. However, in all practical scenarios, it is better to go with Mesosphere DCOS rather than playing with plain vanilla Mesos and Marathon.

DCOS offers a number of supporting components on and above plain Mesos and Marathon to manage enterprise-scale deployments.



The DCOS architecture is well explained in the following link:
<https://dcos.io/docs/1.9/overview/architecture>

Let's do a side-by-side comparison of the components used for scaling microservices with Spring Cloud components and similar capabilities offered by DCOS.

The following points examines the major differences between Spring Cloud approach and DCOS:

- Configurations are managed using the Spring Cloud Config server when we approach with Spring Cloud for scaling microservices. When using DCOS, configurations can be managed by using one of Spring Cloud Config, Spring Profiles, or tools such as Puppet, Chef, and more.
- The Eureka server was used for managing service discovery and Zuul was used for load balancing with the Spring Cloud approach. DCOS has Mesos DNS, VIPs, and HAProxy based Marathon-lb components to achieve the same.
- Logging and monitoring, explained in [Chapter 8, Logging and Monitoring Microservices](#), is accomplished using Spring Boot Actuator, Seluth, and Hystrix . DCOS offers various log aggregation and metrics collection features out of the box.

Implementing Mesos and Marathon for BrownField microservices

In this section, the dockerized Brownfield microservices developed in [Chapter 9, Containerizing Microservices with Docker](#), will be deployed into the AWS cloud, and we will manage them with Mesos and Marathon.

For demonstration purposes, we will cover only two services (search and website deployment) in this chapter. Also, we will use one EC2 instance for the sake of simplicity.

Installing Mesos, Marathon, and related components

Launch a **t2.large** EC2 instance with the Ubuntu 16.04 version AMI, which will be used for this deployment. In this example, we are using another instance to run RabbitMQ; however, this can be done on the same instance as well.

Perform the following steps to install Mesos and Marathon:

- To install Mesos 1.2.0, follow the instructions documented in the following link. This will also install JDK 8:

```
https://mesos.apache.org/gettingstarted/.
```

- To install Docker, perform the following steps:

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

- To install Marathon 1.4.3, follow the steps documented in the following link:

```
curl -O http://downloads.mesosphere.com/marathon  
/v1.4.3/marathon-1.4.3.tgz  
tar xzf marathon-1.4.3.tgz
```

- To install ZooKeeper, perform the following steps:

```
wget http://ftp.unicamp.br/pub/apache/zookeeper/zookeeper-  
3.4.9/zookeeper-3.4.9.tar.gz  
tar -xzvf zookeeper-3.4.9.tar.gz  
rm -rf zookeeper-3.4.9.tar.gz  
cd zookeeper-3.4.9/  
cp zoo_sample.cfg zoo.cfg
```

Alternatively, you may use the DCOS package distribution on AWS using CloudFormation, which offers a high availability Mesos cluster out of the box.

Running Mesos and Marathon

Perform the following steps to run Mesos and Marathon:

1. Log in to the EC2 instance and run the following commands:

```
ubuntu@ip-172-31-19-249:~/zookeeper-3.4.9  
$ sudo -E bin/zkServer.sh start  
  
ubuntu@ip-172-31-19-249:~/mesos-1.2.0/build/bin  
$ ./mesos-master.sh --work_dir=/home/ubuntu/mesos-  
1.2.0/build/mesos-server  
  
ubuntu@ip-172-31-19-249:~/marathon-1.4.3
```

```
$ MESOS_NATIVE_JAVA_LIBRARY=/home/ubuntu/mesos-
  1.2.0/build/src/.libs/libmesos.so ./bin/start --master
  172.31.19.249:5050

ubuntu@ip-172-31-19-249:~/mesos-1.2.0/build
$ sudo ./bin/mesos-agent.sh --master=172.31.19.249:5050 --
  containerizers=mesos,docker --work_dir=/home/ubuntu/mesos-
  1.2.0/build/mesos-agent --resources='ports:[0-32000]'
```



Note that the relative path for a working directory may lead to execution problems. The `--resources` are required only if you want to force the host port to be in a certain range.

2. If you have more slave machines, you may repeat slave commands on those machines to add more nodes to the cluster.
3. Open the following URL to see the Mesos console. In this example, there are three slaves running, connecting to the master:

<http://ec2-54-68-132-236.us-west-2.compute.amazonaws.com:5050>

You will see the Mesos console similar to the following screenshot:

Agents	
Activated	3
Deactivated	0
Unreachable	0

Active Tasks					
Framework ID	Task ID	Task Name	State	Started ▾	Host
No active tasks.					

Unreachable Tasks				
Framework ID	Task ID	Task Name	Started ▾	Agent ID
No unreachable tasks.				

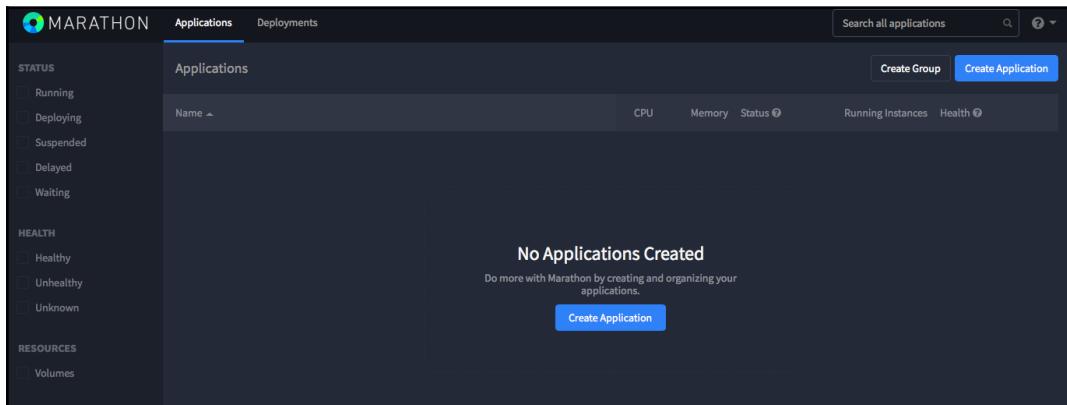
Completed Tasks						
Framework ID	Task ID	Task Name	State	Started ▾	Stopped	Host
No completed tasks.						

Orphan Tasks						
Framework ID	Task ID	Task Name	State	Started ▾	Stopped	Host
No orphan tasks.						

The **Agents** section of the console shows that there are three activated Mesos agents available for execution. It also indicates that there is no **Active Tasks**.

4. Open the following URL to inspect the Marathon UI. Replace the IP address with the public IP address of the EC2 instance:

<http://ec2-54-68-132-236.us-west-2.compute.amazonaws.com:8080>



Since there are no applications deployed so far, the **Applications** section of the UI is empty.

Preparing BrownField PSS services

In the previous section, we successfully set up Mesos and Marathon. In this section, we will see how to deploy the BrownField PSS application previously developed using Mesos and Marathon.



The full source code of this chapter is available under the chapter10 projects in the code files under <https://github.com/rajeshrv/Spring5Microservice>. Copy chapter9.* into a new STS workspace and rename it chapter10.*.

In this example, we will force the Mesos cluster to bind to fixed ports, but, in an ideal world, we will delegate the Mesos cluster to dynamically bind services to ports. Also, since we are not using a DNS or HA Proxy, we will hardcode the IP addresses. In the real world, a VIP for each service will be defined, and that VIP will be used by the services. This VIP will be resolved by the DNS and Proxy.

Perform the following steps to change the BrownField application to run on AWS:

1. Update search microservices (`application.properties`) to reflect the RabbitMQ IP and port. Also, update the website (`Application.java` and `BrownFieldSiteController.java`) to reflect the IP address of the EC2 machines.
2. Rebuild all microservices using Maven. Build and push the Docker images to the Docker Hub. The working directory has to be switched to the respective directories before executing these commands from the respective folders:

```
docker build -t search-service:1.0 .
docker tag search:1.0 rajeshrv/search:1.0
docker push rajeshrv/search:1.0

docker build -t website:1.0 .
docker tag website:1.0 rajeshrv/website:1.0
docker push rajeshrv/website:1.0
```

Deploying BrownField PSS services

The Docker images are now published to the Docker Hub registry. Perform the following steps to deploy and run the BrownField PSS services:

1. Start the dockerized RabbitMQ:

```
sudo docker run --net=host rabbitmq:3
```

2. At this point, the Mesos Marathon cluster is up and running and is ready to accept deployments. The deployment can be done by creating one JSON file per service, as follows:

```
{
  "id": "search-3.0",
  "container": {
    "type": "DOCKER",
    "docker": {
      "image": "rajeshrv/search:1.0",
      "network": "BRIDGE",
      "portMappings": [
        { "containerPort": 8090, "hostPort": 8090 }
      ]
    }
}
```

```
        },
        "instances": 1,
        "cpus": 0.5,
        "mem": 512
    }
```

3. The preceding JSON will be stored on the `search.json` file. Similarly, create a JSON file for other services as well.

The JSON structure is explained as follows:

- `id`: This is a unique `id` of the application and it can be a logical name.
 - `Cpus, mem`: This sets the resource constraints for this application. If the resource offer does not satisfy this resource constraint, Marathon will reject that resource offer from the Mesos Master.
 - `instances`: How many instances of this application to start with? In the preceding configuration, by default, it starts one instance as soon as it gets deployed. Marathon will maintain the number of instances mentioned at any point.
 - `container`: This parameter tells the Marathon executor to use the Docker container for execution.
 - `image`: This tells the Marathon scheduler which Docker image has to be used for deployment. In this case, this will download the `search-service:1.0` image from the Docker Hub repository, `rajeshrv`.
 - `network`: This value is used for Docker runtime to advice on the network mode to be used when starting the new Docker container. This can be `BRIDGE` or `HOST`. In this case, the `BRIDGE` mode will be used.
 - `portMappings`: The port mapping provides information on how to map internal and external ports. In the preceding configuration, `hostPort`, set as `8090`, tells the Marathon executor to use `8090` when starting the service. Since `containerPort` is set as `0`, the same host port will be assigned to the container. Marathon picks up random ports if the `hostPort` value is `0`.
4. Once this JSON is created and saved, deploy this to Marathon using the Marathon REST APIs as follows:

```
curl -X POST http://54.85.107.37:8080/v2/apps
      -d @search.json -H "Content-type: application/json"
```

5. Alternatively, use the Marathon console for deployment as follows:

```
1 {  
2   "id": "search-3.0",  
3   "container": {  
4     "type": "DOCKER",  
5     "docker": {  
6       "image": "rajeshrv/search:1.0",  
7       "network": "BRIDGE",  
8       "portMappings": [  
9         { "containerPort": 8090, "hostPort": 8090 }  
10      ]  
11    },  
12  },  
13  "instances": 1,  
14  "cpus": 0.5,  
15  "mem": 512  
16}  
17  
18 |
```

Cancel **Create Application**

6. Repeat this step for the website as well.

7. Open the Marathon UI. As shown in the following diagram, the UI shows that both the applications are deployed and are in the **Running** state. It also indicates that a **1 of 1** instance is in the **Running** state.

The screenshot shows the Marathon UI's Applications page. On the left, there are filters for STATUS (Running, Deploying, Suspended, Delayed, Waiting) and HEALTH (Healthy, Unhealthy, Unknown). The main area displays two applications: "search-3.0" and "website-3.0". Both applications are listed with 0.5 CPU, 512 MB Memory, and a status of "Running". The "Running Instances" column shows "1 of 1" for both. A "Create Group" and "Create Application" button are located at the top right.

8. If you inspect the search service, it shows ip and the port it is bound to:

The screenshot shows the Marathon UI for the "search-3.0" application. At the top, it says "Running (1 of 1 instances)" with 0 Healthy, 0 Unhealthy, and 1 Unknown (100%). Below this are buttons for "Scale Application", "Restart", and a gear icon. The "Instances" tab is selected, showing a table with columns: ID, Status, Error Log, Output Log, Version, and Updated. One instance is listed with the ID "search-3.0.47c76c0d-52d6-11e7-ab2a-0242da098c5a", status "Started", and bound to "ip-172-31-28-25.us-west-2.compute.internal:8090". The "Output Log" link is underlined.

9. Open the following URL in a browser to verify the website application:

<http://ec2-34-210-109-17.us-west-2.compute.amazonaws.com:8001>

Summary

In this chapter, we learned the different aspects of autoscaling applications and the importance of a container orchestration to efficiently manage dockerized microservices at scale.

We explored the different container orchestration tools before deep diving into Mesos and Marathon. We also implemented Mesos and Marathon in the AWS cloud environment to demonstrate how to manage dockerized microservices developed for the BrownField PSS.

So far, we have seen all the core and supporting technology capabilities required for a successful microservices implementation. A successful microservice implementation also requires processes and practices beyond technology. In the next chapter, the last in the book, we will cover the process and practice perspectives of microservices.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



Application Development with Spring 5.0 and Angular 6 [Video]

Alberto Di Martino, Alejandro Imass

ISBN: 978-1-78862-717-7

- Use the features of Spring 5.0 and Spring Boot with Java
- Test your code with JUnit 5.0
- Use Spring WebFlux and various other features
- Work with AngularJS to develop the front end
- Use Angular 6 and its features



Spring 5.0 Core Training [Video]

Izzet Mustafaiev

ISBN: 978-1-78728-860-7

- Get to know Aspect-oriented Programming for real-world use cases
- Take advantage of all features of Spring framework 5.0
- Learn reactive streams to build a robust back end
- Implement Spring MVC in your apps
- Integrate Spring MVC for a beautiful front-end design
- Create a robust and scalable Microservice based application on Spring Cloud, using Spring Boot

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index

@

`@CachePut` annotation
 cache key, customizing 441
 conditional caching 442

A

AB testing strategies 472
Abstract Factory pattern
 applying, common problems 304
 benefits 304
 implementing, in Spring 402
 implementing, in Spring (FactoryBean interface) 401
 implementing, in Spring Framework 305
 sample implementation 305, 402
 used, for resolving dependency 401
Accordance 667
accountCache 440
AccountServiceImpl class 436
ACID 471
Actuator 666
adapter design pattern
 benefits 324
 implementing, in Spring Framework 325
 requirements 325
 sample implementation 326
 UML diagram 325
Advanced maturity level
 characteristics 595
Advanced Message Queuing Protocol (AMQP)
 258
agents 691
Aggregate Hystrix streams
 with Turbine 672, 675
Airbnb
 about 516

URL 516
airline Bookings microservice 257
airline Payments microservice 257
Airline Ticket System
 about 218
 functionalities 218, 219
 solution diagram 220
Amazon EC2 Container Services (ECS) 691
Amazon
 about 517
 URL 517
AMQP 463
AngularJS
 application entry point, creating 70, 71, 73
 Category Controller, creating 74
 Category Service, creating 75
 CMS application, integrating with 68
 concepts 69
 controllers 69
 services 70
annotation-based configuration
 using, with dependency injection pattern 390
Annotation
 used, for enabling caching proxy 437
antifragility 474
Apache Camel 497
Apex 505
APIs
 testing 153
AppConfig.javaconfiguration class
 creating 382
 dependency injection pattern configuration, approach 384
 Spring beans, declaring into configuration class 382
 Spring beans, injecting 383
AppDynamics 665, 666

Application Performance Monitoring (APM) 664
approaches, for monitoring microservices
 Application Performance Monitoring (APM) 664
 Real user monitoring (RUM) 665
 synthetic monitoring 665
Archiva 499
Aurora 690
Auth microservice
 client credentials flow 278
 creating 272
 implicit grant flow 280
 security, configuring 273, 275, 277
 testing 278
Authentication microservice
 creating 271, 272
automation tools 589
autoscaling
 with resource constraints 680
 about 680
 based on business parameters 680
 based on message queue length 680
 in specific time periods 680
 predictive autoscaling 681
Autowiring DI pattern
 and disambiguation 399
 disambiguation, resolving 400
autowiring
 @Autowired, using with fields 398
 @Autowired, using with setter method 398
 about 391
 beans, annotating 397
Avro 463
AWS Lambda 505
Azure Container Services (ACS) 691
Azure Functions 505
Azure service fabric
 about 520
 URL 520

B

Backend as a Service (BaaS) 515
Basic maturity level
 characteristics 594
Behavioral design patterns

about 323, 354
Chain of Responsibility design pattern 355
Command Design pattern 356
benefits, microservices
 DevOps, enabling 492
 elastic scalability 483
 event-driven architecture, supporting 491
 experimentation, enabling 482
 innovation, enabling 482
 organic systems build, enabling 486
 polyglot architecture, supporting 481
 selective scalability 483
 self-organizing systems, building 490
 substitution, allowing 485
 technology debt, managing 487
 versions co-existence, allowing 488
best practices
 for configuring, DI pattern 404
 for JDBC 431
bill-of-materials (BOM) 530
bindings
 configuring, on Spring AMQP 162
 declaring, in yaml 162
Booking 486
Booking Service 480
breed integration
 about 653
 dashboards 654
 log shippers 653
 log storage 654
 log stream processors 654
Bridge design pattern
 about 328
 benefits 329
 implementing, in Spring Framework 329
 sample implementations 330
 solution, to issues 329
 UML structure 331
BrightWork 505
BrownField microservices
 Marathon, implementing 697
 Mesos, implementing 697
BrownField PSS services
 deploying 701, 704
 preparing 700

BrownField PSS
architecture, summarizing 644, 646
environment, setting up for 600

Builder design pattern
about 317, 318
applying, to common issues 319
benefits 317
implementation, for creating embedded data source 421
implementing, in Spring Framework 318
UML class structure 317

Business Layer 463

C

cache abstraction
about 435
cache configuration 436
caching declaration 435

cache storage configuration
about 448
CacheManager, setting up 448

cache
about 434
enabling, via Proxy pattern 436

caching proxy
enabling, Annotation used 437
enabling, XML namespace used 438

caching
best practices, in web application 452
using 434

capability model 577, 578

Cassandra 459

Catchpoint 666

central log management 583

centralized logging solution
about 650
components 651

cgroups 692

Chain of Responsibility design pattern
about 355
in Spring Framework 356

Chalice for Python 505

characteristics, microservices
antifragility 474
distributed 473

dynamic 473
environment, automation 471
fail fast 474
lightweight 469
polyglot architecture 470
self healing 474
services 467
supporting ecosystem 472

characteristics, services
composeability 469
discoverable 469
interoperable 469
loose coupling 468
reuse 468
service abstraction 468
service contract 468
statelessness 468

Circonus 666

Circuit Breaker
about 667
URL 667

Cloud Foundry Diego 691

Cloud Monitoring 665

Cloud
about 493, 508
as self-service infrastructure, for microservices 509

CloudTrail 652

CloudWatch 665

CMS project
creating 40
dependencies section 41
generating 41
metadata section 40

CMS
Docker image, creating for 99

cognitive computing 493

cold publishers 109

collectd 665

Command Design pattern
about 357
in Spring Framework 357

command line
used, for executing CMS application 47
using, with JAR file 48

using, with Spring Boot Maven plugin 47

Command Query Responsibility Segregation (CQRS) 515

commands, Docker

- docker container 35
- docker network 36
- docker run 34
- docker volume 37

component scanning

- about 391
- used, for searching beans 394

components, centralized logging solution

- log dashboard 652
- log shippers 651
- log store 651
- log stream processor 651
- log streams 651

components, Project Reactor

- publishers 105
- subscribers 107

composite design pattern

- about 334
- benefits 336
- sample implementation 337
- solutions, to issues 335
- UML structure 335

Config Client 604

Config Server project

- creating 222

Config Server

- about 598, 604
- accessing, from clients 609
- health, monitoring 616
- high availability, setting up 614
- microservices, building 604
- setting up 605

Configurable Items (Clis) 586

Configuration Management Database (CMDB) 586

constructor based dependency injection

- advantages 375
- disadvantages 376
- example 376
- versus setter-based dependency injection 379

container as service 582

container orchestration solutions

- Apache Mesos 690
- CoreOS Fleet 691
- Docker Swarm 688
- HashiCorp Nomad 690
- Kubernetes (k8s) 689

container orchestration

- about 682
- actions 683
- importance 682
- Marathon, using 691
- Mesos, using 691
- relationship, with microservices 687
- relationship, with virtualization 687
- solutions 687

container registry 590, 591

container scheduler 582

Container technologies 467

containerized solution

- running 205, 206

containers

- about 493, 508
- as self-service infrastructure, for microservices 509

Content Delivery Network

- about 71
- reference 71

Content Management System (CMS) 480

Content Management System (CMS) application

- @Component annotation 46
- @Configuration annotation 46
- @EnableAutoConfiguration annotation 46
- @SpringBootApplication annotation 46
- CMS project, creating 40
- executing 46
- executing, in command line 47
- executing, in IntelliJ IDEA 47
- integrating, with AngularJS 68
- project structure 42, 43, 45
- structure, creating 39

continuous integration (CI) 472

Conway's law

- about 515

URL 515
core capabilities
about 578
libraries 578
service endpoints 580
service implementation 579
service listeners 578
storage capability 579
core container 10
core design patterns
Behavioral design patterns 298
creational design pattern 298
structural design pattern 298, 323
Core Legacy System 458
CouchDB 459
creational design pattern
about 299
Abstract Factory pattern 304
factory design pattern 300
Cross Origin Resource Sharing (CORS)
about 553
enabling, for microservices interactions 553, 554
Cucumber 472
custom caching annotations
creating 451
custom health module
adding 558, 560
custom metrics
building 560
Customer Notification Service 563
Customer Profile Service 563
customer registration microservice example
developing 562, 564, 565, 567, 568, 570, 573, 574

D

Dalston SR1 600
DAO (Data Access Object) 82
Dapper 659
data access layer
creating 91
data access object (DAO) pattern
about 408
used, for abstracting database access 422

using, with Spring Framework 423
Data analysis
Data Lake, using 675
Data Centre Operating System (DCOS)
about 582, 692
differentiating, with Spring Cloud approach 697
reference 696
used, for implementing Marathon 696
used, for implementing Mesos 696
Data Lake 587, 648
data source
configuring 416
configuring, JDBC driver used 417
configuring, pool connections used 419
data types, Redis
lists 140
sets 141
strings 139
data-access
designing, approaches 408
Database Layer 463
Datadog 665
Dataloop 665
Declarative Annotation-based caching
@Cacheable annotation 439
@CacheConfig annotation 444
@CacheEvict annotation 443
@CachePut annotation 440
@Caching annotation 444
about 439
Declarative XML-based caching 445
Decorator design pattern
about 339
benefits 339
implementing 343
in Spring Framework 345
solutions, to issues 340
UML structure 342
dependency injection (DI) pattern
types 375
about 368, 369
annotation-based configuration, using 390
avoiding 370
configuring, best practices 404

configuring, with Spring 380
constructor-based 375
Java-based configuration, using with 381
setter-based 377
Stereotype annotations 391
used, for solving issues 369
using 373
XML-based configuration, using 385

dependency injector 373
dependency management 586
design patterns
 about 297
 characteristics 297
development environment
 setting up 27, 523
Device42 667
DevOps
 about 493, 508
 practice, for microservices 509
 process, for microservices 509
 URL 464
Digital Performance Monitoring (DPM) 664
discovery
 with Eureka 617
Docker commands 34
docker container command 35
Docker containers 33
Docker Hub
 image, pushing to 102
 RabbitMQ image, pulling from 158
 Redis image, pulling from 137
Docker image
 about 32
 creating, for CMS 99
docker network command 36
Docker network
 about 33
 creating, for application 136
docker run command 34
Docker Spring profile
 configuring 103
docker volume command 37
Docker volumes 34
docker-compose file
 creating 212

docker-compose project
 reference 202
docker-compose tool
 about 210
 installing 211
 network, testing 215
 solution, running 214
Docker-Maven-plugin
 adding, on pom.xml 100
 configuring 100
Docker
 about 459, 502, 681
 basics 78
 bridge 33
 infrastructure, placing on 251, 252, 254
 installing 30
 overlay network 33
 RabbitMQ server, starting with 158
 reference 31
Dockerized CMS
 running 103, 104
Domain-Driven Design (DDD) 218
domain-specific language (DSL) 17
Dynatrace 665, 666

E

eBay
 about 517
 URL 517
Eclipse 523
Ehcache-based cache
 about 449
 XML-based configuration 450
elastic 510
Elastic Search 471
Elasticsearch, Logstash and Kibana (ELK) 654, 666
Elvis operator 132
embedded data source
 creating, by implementing Builder pattern 421
engine 691
enterprise applications
 data access layer 409
 infrastructure layer 409
 service layer 409

Enterprise Integration Patterns (EIP) 495
Enterprise Service Bus (ESB)
 about 473, 495
entry points, for adoption
 about 596
 Brown Field approach 596
 Green Field approach 596
Erlang 471
Eureka Client 620
Eureka Server 620, 630
Eureka server main class
 creating 229, 230
Eureka server
 checking 246, 247
Eureka
 about 619
 discovery 618
 dynamic service registration 618
 Eureka Server, setting up 621
 high availability 627
 used, for discovery 617
 used, for registration 617
examples, microservices
 holiday portal 475
 travel agent portal 478
exchanges, RabbitMQ
 about 161
 direct exchanges 161
 fanout exchanges 161
 header exchanges 161
 topic exchanges 161
exchanges
 configuring, on Spring AMQP 162
 declaring, in yaml 162
Executor 692

F

Fabric8
 about 520
fabric8
 reference 100
Fabric8
 URL 520
Facade design pattern
 about 346
 implementing 349
 in Spring Framework 351
 UML structure 350
 using, situations 346
factory design pattern
 benefits 300
 implementing, in Spring Framework 300
 issues 300
 sample implementation 301
Fail Fast 475
Fake SMTP server
 URL 573
Fares 486
Fat-JAR 45
Fenzo 690
Fleet 691
Fluentd 653
Flume 654, 676
Flux 544, 546
followers 691
framework 8
frameworks, microservice
 Azure service fabric 519
 Fabric8 519
 Gizmo 519
 Go Kit 519
 Go-fasthttp 519
 Hook.io 519
 JavaLite 519
 Jooby 519
 Lagom 519
 Lumen 519
 Micro 519
 Payra-micro 519
 Restlet 519
 Restx 519
 Seneca 519
 Spark 519
 Vamp 519
 Vert.x 519
 WSO2 Microservices For Java - MSF4J 519
Functions as a Service (FaaS) 504
functions
 declaring, in Kotlin 124

G

Gang of Four (GOF) pattern
about 367
overview 298

Gilt
about 517
URL 517

Git 21

GitHub repository addresses, for projects
references 292

Gizmo
about 520
URL 520

Go Kit
about 520
URL 520

Go-fasthttp
about 520
URL 520

Google Cloud Functions 505

Google Cloud Logging 652

Google Container Engine 691

Gordon 505

Gradle 499, 523

Grafana 654

Graphite 654, 666

graylog 503, 653

H

Hadoop 459

Hadoop Distributed File System (HDFS) 482, 507

HATEOAS-enabled Spring Boot microservice 535, 537, 538, 539, 540

Hexagonal Architecture
about 461
URL 461

Hibernate
reference 82

high availability
setting up for Rabbit MQ, URL 616
setting up, for Config Server 614
URL 616

holiday portal 475

honeycomb analogy 464, 513

Hook.io
about 520
URL 520

HTrace 659

HTTP protocol
about 194
and persistent connections 195

httpbin
reference 250

Hyper-V 470

HyperMedia As The Engine Of Application State (HATEOAS) 535

Hypertext Application Language (HAL) 535

Hystrix Dashboard 260, 261

Hystrix
collection commands statistics 289, 290, 291
URL 671

I

IBM OpenWhisk 505

IDE
installing 29

In-Memory Data Grid (IMDG) 485

Infrastructure as a Service (IaaS) 581

infrastructure as code 509

infrastructure capabilities
about 580
cloud 581
container orchestration 582
container runtime 581

inner architecture 578

Integration Platform as a Service (iPaaS)
about 459

IntelliJ IDEA 523
about 30
used, for executing CMS application 47

Intermediate maturity level
characteristics 594

Internet of Things (IoT) 456, 493

Interpreter Design pattern
about 358
in Spring Framework 359

Inversion of Control (IoC) pattern 373

Iterator Design Pattern

about 359
in Spring Framework 360

J

Jackson
 adding, for Kotlin 135
JAR file
 command line, using with 48
Java Logging (JUL) 649
Java Message Service (JMS) 463, 495
Java Persistence API (JPA) 408
Java-based configuration
 using, with Java-based configuration 381
JavaLite
 about 520
 URL 520
JBoss 466
JConsole
 used, for monitoring 557
JCP (Java Community Process) 82
Jdbc callback interfaces
 ResultSetExtractor, implementing 429
 RowCallbackHandler, implementing 428
 RowMapper class, creating 427
 RowMapper, implementing 427
JDBC driver
 used, for data source configuration 417
JdbcTemplate
 callback interfaces 427
 configuring 431
 creating, in application 425
 JDBC-based repository, implementing 425
 using 424
 working with 424
JDK 1.8
 URL 523
JEE design patterns
 about 299, 364
 at business layer 365
 at integration layer 365
 at presentation layer 365
Jetty 502
jmxtrans 665
jobs 691
Jooby

about 520
URL 520
JVM (Java Virtual Machine) 122

K

Kafka 654
key capabilities, container orchestration
 agility 685
 cluster management 684
 deployments 684
 health 684
 infrastructure abstraction 684
 isolation 685
 resource allocations 685
 resource optimizations 684
 scalability 684
 service availability 685
Kibana 654
Kotlin idioms
 null safety 131
 range expressions 130
 Smart Casts 129
 string interpolation 129
Kotlin
 basics 122
 characteristics 122
 companion objects 128
 data classes 126
 functions, declaring in 124
 functions, overriding 125
 Jackson, adding for 135
 objects 127
 project use case 133
 project, creating 133
 semantics 124
 simple function, with parameters 124
 simple function, with return type 124
 simple function, without return 125
 single expressions functions 125
 syntax 123
Kubernetes (k8s) 472, 689

L

Lagom
 about 520

URL 520
Lambda architecture
about 506
Big Data 507
Bots 507
cognitive computing 507
IoT 507
URL 652
Lambda architectures 493
Lambda Framework for Java 505
Librato 666
lists
about 140
main commands 140, 141
log management, challenges 649
Logentries 652
logging solutions
breed integration 653
cloud services 652
custom logging, implementation 654, 658
off-the-shelf solutions 653
selection 652
Spring Cloud Sleuth, distributed tracing 659, 661
Loggly 503, 652, 666
Logplex 503
Logsene 652
Logstash 503, 653
Lumen
about 520
URL 520

M

Mail microservice project
creating 264
Mail message, modeling 267
MailSender class 267
RabbitMQ dependencies, adding 264
RabbitMQ queue listener, creating 269
RabbitMQ stuff, configuring 265
Mail microservice
creating 262
running 270
manager 689
Mantl

about 520
URL 520
MapR 507
Marathon Load Balancer (marathon-lb) 583
Marathon
about 690, 695
container orchestration 691
executing 698
implementing, with DCOS 696
installing 697
URL, for installation 698
master 689
maturity model
about 592
Advanced maturity level 595
Basic maturity 594
Intermediate maturity level 594
Traditional maturity 593
Maven 499
Maven 3.3.1
URL 523
Maven plugins
for Kotlin 135
Maven
installing 28
Mean Time Between Failures (MTBF) 475
Mesos architecture
Executor 694
Framework 694
Master 693
Scheduler 694
Slave 693
workflow diagram 694
ZooKeeper 693
Mesos Master 692
Mesos
about 472
architecture 692
chain orchestration 691
executing 698
implementing, with DCOS 696
installation link 698
installing 697
Mesosphere DCOS 459
message based 510

messages
 consuming, with Spring messaging 165
 producing, with Spring messaging 166

metrics
 collecting, with Zipkin 285, 286, 287, 288

Micro
 about 520
 URL 520

Microservice A 630

microservice documentation 591

microservices evolution
 about 455
 business demand, as catalyst 456, 458
 imperative architecture evolution 459
 technology, as catalyst 459

microservices, monitoring
 about 662
 Aggregate Hystrix streams, with Turbine 672, 675
 approaches 664
 challenges 662, 663
 dependency, monitoring 666
 key areas 663
 Spring Cloud Hystrix, used for fault-tolerant microservices 667, 671
 tools 665, 666

microservices
 about 455, 460, 464
 and Spring Boot 26
 benefits 481
 characteristics 26, 467
 dockerizing 291
 documenting 561, 562
 examples 475
 frameworks 519
 integrations, testing 181
 monitoring 285
 monolithic migration 518
 principles 465
 relationship 686
 scaling 679
 URL 462
 use cases 514

modules, Spring 9

MongoDB
 preparing 78, 79, 80

Mono 546

N

Nagios 666

named client 20

Neo 4j 459

NetBeans 523

Netflix
 about 516
 URL 516

Netty
 about 14
 reference 14

New Event Aggregation Service 491

New Relic 665, 666

New Relic synthetic 666

NewSQL 459

Nexus 499

Nike
 about 518
 URL 518

nodes 689

Nomad 690

NoSQL 459

Notification Services 480

null safety, Kotlin
 Elvis operator 132
 safe calls 132

NuoDB 459

O

OAuth 2.0 flow
 entities 271

OAuth 2.0, for protecting microservices
 about 282
 application.yaml file, configuring 282
 JwtTokenStore Bean, creating 283
 security dependency, adding 282

OAuth2
 microservice, securing 550, 552, 553

object pool pattern
 configuring 416

object pooling
 connection pooling 431

statement pooling 431
Object Relational Mapping (ORM)
about 407
Observer Design Pattern
about 361
in Spring Framework 363
Open Group SOA Reference Architecture
URL 577
OpenJDK
installing 28
ops monitoring 585
Orbitz
about 517
URL 517
Order DB 578
Order Event 491, 511
organizations, using microservices
Airbnb 516
Amazon 517
eBay 517
Gilt 517
Netflix 516
Nike 518
Orbitz 517
Twitter 517
Uber 516
OSI model
reference 194
outer architecture 578
oversubscriptions 686

P

pagination 95
Papertrial 652
pay as you use model 506
Payment Card Industry Data Security Standard (PCI-DSS) 480
Payra-micro
about 520
URL 520
pgAdmin3
about 89
configuring 89
installing 89
Pingdom 666

Plain Old Java Object (POJO) 497, 559
Platform as a Service (PaaS) 459, 580
pods 689
POJO (Plain Old Java Object) pattern 91
polyglot architecture
supporting 481
pom.xml
configuring, for Spring Data JPA 82
pool connections
used, for data source configuration 419
Ports and Adapters pattern 461
PostgreSQL database
preparing 80
Presentation Layer 463
Pricing 486
principles, microservices
autonomous 466
single responsibility per service 465
process and governance capabilities
about 588
automation tools 589
container registry 590, 591
DevOps 589
libraries 592
microservice documentation 591
reference architecture 592
Project Object Model (POM) 499
Project Reactor
about 105
components 105
project
creating, with Spring Initializr 134
projects, Spring Cloud
Spring Cloud Bus 23
Spring Cloud Config 21
Spring Cloud Consul 22
Spring Cloud Netflix 19
Spring Cloud Security 22
Spring Cloud Stream 23
Prometheus 666
Promise theory 511
Protocol Buffers 463
Prototype design pattern
about 313
benefits 313

sample implementation 315
UML class structure 314

Proxy design pattern
about 352
caching, enabling via 436
implementing 353
in Spring Framework 354
purpose 352
UML structure 352

publishers 105

Q

QBit 512
Quality of Services (QoS) 483
queues
configuring, on Spring AMQP 162
declaring, in yaml 162

R

RabbitMQ image
pulling, from Docker Hub 158
RabbitMQ queues
consuming, reactively 199, 200
RabbitMQ Reactor beans
configuring 198
RabbitMQ server
starting 158, 159
starting, with Docker 158
RabbitMQ
bindings 162
exchanges 161
queues 162
Spring Application, integrating with 160
Spring beans, declaring for 163
URL 547
using, in reactive microservices 546, 547
range expressions, Kotlin
downTo case 131
simple case 130
step case 131
until case 130
reactive manifesto
URL 510
reactive microservices
about 493, 510, 541

order management system 512
Rabbit MQ, using 546, 547
Reactive Streams 545
Spring Boot, using 546, 547
Spring Cloud Streams 638, 642
Spring WebFlux, using 541, 542, 543

Reactive repository
creating 113
working 113
reactive sequences
hot 108

Reactive Spring 105
reactive Spring Boot application

URL 542
Reactive Streams
about 541, 545
Processor 546
Publisher 545
subscriber 545
Subscription 545
reactive types 108
Reactor project
URL 541
Reactor RabbitMQ 197
Reactor
working with 109

Real user monitoring (RUM) 665
Red Hat Funktion 505

Redis image
pulling, from Docker Hub 137
Redis instance
running 137
redis-cli tool
configuring 138
Redis
about 138
data types 139
reference architecture 592
registration
with Eureka 617
reliable messaging 588
resilient 510
resource management problem 411
responsive 510
REST layer

CategoryResource, modifying 118
modifying 116
Spring WebFlux dependency, adding 117

REST resources
Category class, creating 51
CategoryResource class, creating 53
creating 49, 53
Lombok dependency, adding 50
models, creating 51
News class, creating 52
NewsResource class, creating 56
Tag class, creating 51
User class, creating 52
UserResource class, creating 55

RESTful microservices
building, with Spring Boot 523, 524

Restlet
about 520
URL 520

Restx
about 520
URL 520

Riemann 665

route
creating, with Spring Cloud Gateway 247, 248, 249, 250

Runscope 666

Ruxit 665

RxJava 511

RxJS 512

S

SaaS (Software as a Service) 262

safe calls 132

Scale Cube
about 484
URL 484

Scheduler 692

security service 585

security
implementing 549
microservice, securing 550
microservice, securing with OAuth2 550, 552, 553

Selenium 472

SendGrid account
creating 263

SendGrid
about 262
reference 263

Seneca
about 520
URL 520

Sensu 666

Server-Sent Events (SSE) 193, 196, 260

Serverless 505
serverless computing 493, 504
service configuration 585
service discovery 584
service gateway 583
service layer
adding 57
CategoryService class, creating 59
CategoryService, modifying 114
fixing 114
model, modifying 57
new review, adding 58
news, checking 58
NewsService class, creating 61
prerequisites 58
Swagger, configuring 62
UserService class, creating 60

service listeners 578

service mocking 590

service virtualization 590

Service-Oriented Architecture (SOA)
about 455, 493, 494
application 496
legacy modernization 496
service-oriented integration 495
URL 469, 494
used, for monolithic migration 497

Service-Oriented Integration (SOI) 495

services
about 467
characteristics 468

sets
about 141
main commands 142

setter-based dependency injection

about 377
advantages 377
disadvantages 378
example 378
versus constructor 379
Simian Army 475
Simple Mail Transfer Protocol (SMTP) 490
Single Sign-On (SSO) 271
Singleton design pattern
 about 310
 applying, in issues 311
 benefits 311
 early instantiation 310
 implementation, Spring Framework 311
 lazy instantiation 310
 sample implementation 312
SMTP (Simple Mail Transfer Protocol) 262
software defined infrastructure 509
software-defined load balancer 583
SOLID design pattern
 about 465
 URL 465
Span 659
Spark Streaming 654, 676
Sparta for Go 505
Spigo 666
Splunk 503, 653, 666
Spring Actuator
 about 185
 application custom information 187
 endpoints 186
 endpoints, testing 188
Spring AMQP
 about 11
 adding, in pom.xml file 160
 bindings, configuring on 162
 exchanges, configuring 162
 queues, configuring 162
Spring Application
 integrating, with RabbitMQ 160
Spring beans, injecting
 constructor injection, using 387
 setter injection, using 388
Spring beans, Spring container
 about 374

Spring beans
 declaring, for RabbitMQ 163
Spring Boot Actuator
 adding, in pom.xml 186
Spring Boot actuators
 custom health module, adding 558, 560
 custom metrics, building 560
 monitoring, JConsole used 557
 monitoring, SSH used 557
 used, for microservices instrumentation 555
Spring Boot CLI 524
Spring Boot Maven plugin
 command line, using with 47
Spring Boot microservice
 developing 525, 526, 527, 528, 529, 532,
 533, 534
 reference 525
 testing 534, 535
 URL, for dependencies 530
Spring Boot
 about 25, 519, 524, 525
 microservices 26
 used, for building RESTful microservices
 523, 524
 using, in reactive microservices 546, 547
Spring Cloud Bus 21, 23
 for propagating configuration changes 613
Spring Cloud Config 21
Spring Cloud Config Server
 about 220, 221, 601
 Config Server health, monitoring 616
 Config Server URL 608
 Config Server, accessing from clients 609
 Config Server, for configuration files 616
 Config Server, setting up 605
 Config Server, using 617
 configuration changes, handling 613
 enabling 222
 Git repository, configuring as properties
 source 224
 GitHub, using as repository 223
 high availability, setting up for Config Server
 614
 microservices, building with Config Server
 604

running 224, 225
Spring Boot application, configuring 223
Spring Cloud Bus, for propagating configuration changes 613
testing 226
Spring Cloud Consul
about 22
reference 228
Spring Cloud Data Flow 654, 676
Spring Cloud Eureka server
configuring 230
running 232, 233
Spring Cloud Eureka
creating 229
Spring Cloud Feign 19
Spring Cloud Gateway main class
creating 243
Spring Cloud Gateway project
configuring 244, 245
creating 242, 243
Spring Cloud Gateway
about 241
route, creating with 247, 248, 249, 250
running 246
Spring Cloud Hystrix 20
about 648
using 667, 671
Spring Cloud Netflix 19
Spring Cloud Netflix Eureka 19
Spring Cloud Ribbon 20
Spring Cloud Security 22
used, for protecting microservices 643
Spring Cloud service discovery 227, 228
Spring Cloud Sleuth 234
about 648
distributed tracing 659, 661
Spring Cloud Stream 23, 654
Spring Cloud Streams modules 654, 676
Spring Cloud Streams
about 512, 676
using, in reactive microservices 638, 642
Spring Cloud Zipkin server
about 233
application.yaml, configuring 238
bootstrap.yaml, configuring 238
creating 236
infrastructure 234
running 239
Spring Cloud Zookeeper
reference 228
Spring Cloud
about 18, 519, 599
projects 18
releases 600
Spring community
projects 81
Spring container
Twitter credentials, declaring for 168
Spring Core Framework
about 10
core container 10
Spring Data Commons Documentation
reference 95
Spring Data JPA
about 82
data, checking on database structure 90
JPA repositories, adding in CMS application 85
models, mapping 84, 85
pom.xml, configuring for 82
Postgres connections, configuring 83
transactions, configuring 87
Spring Data MongoDB
about 91, 92
database connection, configuring 94
domain model, mapping 93
persistence, checking 96, 97, 98
PostgreSQL, removing 92
repository layer, adding 95
Spring Data project 81
Spring Data Reactive 112
Spring Data Reactive Redis
about 142
ReactiveRedisConnectionFactory, configuring 143
ReactiveRedisTemplate, providing 143
REST resources, exposing 147
Tracked Hashtag repository, creating 145
Spring Data
about 15

for Reactive Extensions 112
Spring Expression Language (SpEL) 359
Spring Fox 49
Spring Framework 5 519
Spring Initializr
about 535
project, creating 134
reference 40, 222
URL 525, 536
Spring Integration 24, 497
Spring IoC containers 10
Spring JMS 13
Spring Messaging AMQP 159
Spring messaging
about 11
messages, consuming with 165
messages, producing with 166
Spring MVC (model-view-controller) 10
Spring Reactive 511
Spring reactive web clients
about 169
authentication, with Twitter APIs 172
gather service, creating 174
models creation, for gathering tweets 170
server-sent events (SSE) 173
Twitter API, consuming 175
WebClient, producing 169
Spring Security 16
Spring Streams 519
Spring Team
projects 81
Spring Tool Suite (STS) 524
Spring Tool Suite 3.8.2 (STS)
URL 523
Spring Tools Suite 30
Spring Web MVC module 13
Spring WebFlux
about 14, 111
event-loop model 112
spring.io blog
reference 17
Spring
about 8
dependency injection pattern, configuring 380
modules 9
or Apache Kafka 12
SSH
used, for monitoring 557
starters 27
statd 665
stereotype annotation 46
Stereotype annotations
about 391
automatic wiring, creating 391
used, for creating auto searchable beans 392
Storm 654
strategies, resource allocation
bin packing 686
random 686
spread 686
streams
filtering 201
strings
about 139
main commands 139
structural design pattern
about 323
adapter design pattern 324
Bridge design pattern 328
composite design pattern 334
composition 323
Decorator design pattern 339
Facade Design Pattern 346
inheritance 323
Proxy design pattern 352
subscribers 107
Sumo Logic 652
supporting capabilities
about 582
central log management 583
data lake 587
dependency management 586
ops monitoring 585
reliable messaging 588
security service 585
service configuration 585
service discovery 584
service gateway 583
software-defined load balancer 583

Swagger
about 49
configuring 62
dependencies, adding to pom.xml 62
documented API, creating 64, 66, 68
Swarm Intelligence 511
system
running 293

T

t2.large EC2 697
task groups 691
template design pattern
about 363
benefits 363
implementing 412
issues, solving with Spring's JdbcTemplate 414
traditional JDBC, issues 413
The Software Development Kit Manager (SDKMAN)
URL 525
third-party cache implementations
about 449
Ehcache-based cache 449
Thrift 463
Tomcat 477
Trace 659, 666
Trace-id 659
Tracked Hashtag Service container
running 206, 207
Tracked Hashtag Service
bindings, creating 179
exchanges, creating 179
messages, sending to broker 180
modifying 177
queues, creating 179
RabbitMQ connections, configuring 178
running 182
Spring Starter RabbitMQ dependency, adding 177
Traditional maturity level
characteristics 593
Travel Agent Portal 479
Turbine 648

Turbine server 257
Turbine server microservice
creating 259, 260
Tweet Dispatcher container
running 209
Tweet Dispatcher project
additional dependencies 193
configuring 204
creating 192
Spring Initializr, using 192
Tweet Gathering container
running 208
Tweet Gathering project
configuring 202
Twelve-Factor Apps
about 493, 498
admin processes, packaging 504
backing services 500
build stage and release stage, isolating between 501
bundle dependencies 499
concurrency, for scale out 502
configurations, externalizing 500
development, production parity principle 503
disposability, with minimal overhead 503
logs, externalizing 503
process share, avoiding 502
release stage and run stage, isolating between 501
services expose, through port binding 502
single code base 499
stateless processes 502
Twitter application
container, running 152
creating 147, 148, 149
image, creating 151
pom.xml, configuring 150
Twitter credentials
configuring, in application.yaml 167
declaring, for Spring container 168
producing 167
Twitter Gathering project
creating 156
running 182, 183
structure 156, 157

- Twitter setting models
 about 167
 TwitterAppSettings 168
 Twittertoken 168
- Twitter
 about 517
 enabling, in application 166
 URL 517
- ## U
- Uber
 about 516
 URL 516
- ## V
- Vamp
 about 520
 URL 520
- version control system (VCS) 231
- Vert.x
 about 520
 URL 520
- Vertias Risk Advisor (VRA) 667
- virtualization 581
- ## W
- Weave scope 666
- Web API Endpoint
 URL 591
- web application
 caching best practices 452
- WebFlux
 about 541
 using, in reactive microservices 541, 542, 543
- Weblogic 466
- WebSockets 196
- WebSphere 466
- Webtask 505
- Wildfly Swarm 519
- Wrapper 339
- WSO2 Microservices For Java - MSF4J
 URL 520
- ## X
- XML namespace
 used, for enabling caching proxy 438
- XML-based configuration file
 creating 386
 Spring beans, declaring 386
 Spring beans, injecting 387
 using, with dependency injection pattern 385
- ## Y
- yaml
 bindings, declaring in 162
 exchanges, declaring in 162
 queues, declaring in 162
- Your Application Classes (POJOs) 381
- ## Z
- Zabbix 665
- ZeroMQ 463
- Zipkin 659
 metrics, collecting with 285, 287, 288
- ZooKeeper 690, 692
- Zuul proxy
 completing, for other services 638
 high availability 635
 high availability, with Eureka Client 636
 high availability, without Eureka Client 637
 setting up 630, 635
 using, as API Gateway 629