

Microservices Best Practices for Java

Michael Hofmann

Erin Schnabel

Katherine Stanley



 **Cloud**

WebSphere



International Technical Support Organization

Microservices Best Practices for Java

December 2016

Note: Before using this information and the product it supports, read the information in “Notices” on page vii.

First Edition (December 2016)

This edition applies to WebSphere Application Server Liberty v9.

© Copyright International Business Machines Corporation 2016. All rights reserved.

Note to U.S. Government Users Restricted Rights -- Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Notices	vii
Trademarks	viii
Preface	ix
Authors	ix
Now you can become a published author, too!	xi
Comments welcome	xi
Stay connected to IBM Redbooks	xi
Chapter 1. Overview	1
1.1 Cloud native applications	2
1.2 Twelve factors	2
1.3 Microservices	3
1.3.1 The meaning of “small”	4
1.3.2 Independence and autonomy	4
1.3.3 Resilience and Fault tolerance	6
1.3.4 Automated environment	6
1.4 Philosophy and team structure	7
1.5 Examples	7
1.5.1 Online retail store	7
1.5.2 Game On!	8
Chapter 2. Creating Microservices in Java	9
2.1 Java platforms and programming models	10
2.1.1 Spring Boot	10
2.1.2 Dropwizard	10
2.1.3 Java Platform, Enterprise Edition	10
2.2 Versioned dependencies	11
2.3 Identifying services	11
2.3.1 Applying domain-driven design principles	12
2.3.2 Translating domain elements into services	13
2.3.3 Application and service structure	15
2.3.4 Shared library or new service?	17
2.4 Creating REST APIs	18
2.4.1 Top down or bottom up?	18
2.4.2 Documenting APIs	18
2.4.3 Use the correct HTTP verb	19
2.4.4 Create machine-friendly, descriptive results	20
2.4.5 Resource URIs and versioning	20
Chapter 3. Locating services	25
3.1 Service registry	26
3.1.1 Third-party registration versus self-registration	26
3.1.2 Availability versus consistency	27
3.2 Service invocation	27
3.2.1 Server side	27
3.2.2 Client side	29
3.3 API Gateway	32

Chapter 4. Microservice communication	33
4.1 Synchronous and asynchronous	34
4.1.1 Synchronous messaging (REST)	34
4.1.2 Asynchronous messaging (events)	35
4.1.3 Examples	36
4.2 Fault tolerance	37
4.2.1 Resilient against change	37
4.2.2 Timeouts	39
4.2.3 Circuit breakers	39
4.2.4 Bulkheads	39
Chapter 5. Handling data	41
5.1 Data-specific characteristics of a microservice	42
5.1.1 Domain-driven design leads to entities	42
5.1.2 Separate data store per microservice	43
5.1.3 Polyglot persistence	45
5.1.4 Data sharing across microservices	45
5.1.5 Event Sourcing and Command Query Responsibility Segregation	47
5.1.6 Messaging systems	48
5.1.7 Distributed transactions	49
5.2 Support in Java	50
5.2.1 Java Persistence API	50
5.2.2 Enterprise JavaBeans	57
5.2.3 BeanValidation	59
5.2.4 Contexts and Dependency Injection	61
5.2.5 Java Message Service API	62
5.2.6 Java and other messaging protocols	65
Chapter 6. Application Security	67
6.1 Securing microservice architectures	68
6.1.1 Network segmentation	69
6.1.2 Ensuring data privacy	69
6.1.3 Automation	70
6.2 Identity and Trust	70
6.2.1 Authentication and authorization	71
6.2.2 Delegated Authorization with OAuth 2.0	71
6.2.3 JSON Web Tokens	73
6.2.4 Hash-Based Messaging Authentication Code	75
6.2.5 API keys and shared secrets	76
Chapter 7. Testing	77
7.1 Types of tests	78
7.2 Application architecture	78
7.3 Single service testing	79
7.3.1 Testing domain or business function	79
7.3.2 Testing resources	81
7.3.3 Testing external service requests	82
7.3.4 Testing data requests	84
7.3.5 Component testing	84
7.3.6 Security verification	85
7.4 Staging environment	85
7.4.1 Test data	85
7.4.2 Integration	86
7.4.3 Contract	86

7.4.4 End-to-end	86
7.4.5 Fault tolerance and resilience	86
7.5 Production environment	87
7.5.1 Synthetic monitoring	88
7.5.2 Canary testing	88
Chapter 8. From development to production	89
8.1 Deployment patterns	90
8.2 Deployment pipelines and tools	91
8.3 Packaging options	92
8.3.1 JAR, WAR, and EAR file deployment on preinstalled middleware	93
8.3.2 Executable JAR file	94
8.3.3 Containerization	96
8.3.4 Every microservice on its own server	97
8.3.5 Aspects of running multiple microservices on one server	98
8.3.6 Preferred practices on packaging	98
8.4 Configuration of the applications across stages	100
8.4.1 Environment variables	100
8.4.2 Configuration files	101
8.4.3 Configuration systems	103
8.4.4 Programming considerations	104
8.4.5 Preferred practices on configuration	105
Chapter 9. Management and Operations	107
9.1 Metrics and health checks	108
9.1.1 Dynamic provisioning	108
9.1.2 Health check	111
9.2 Logging	111
9.3 Templating	113
Related publications	115
IBM Redbooks	115
Other publications	115
Online resources	115
Help from IBM	116

Notices

This information was developed for products and services offered in the US. This material might be available from IBM in other languages. However, you may be required to own a copy of the product or product version in that language in order to access it.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing, IBM Corporation, North Castle Drive, MD-NC119, Armonk, NY 10504-1785, US

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you provide in any way it believes appropriate without incurring any obligation to you.

The performance data and client examples cited are presented for illustrative purposes only. Actual performance results may vary depending on specific configurations and operating conditions.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.


Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation, registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at “Copyright and trademark information” at <http://www.ibm.com/legal/copytrade.shtml>

The following terms are trademarks or registered trademarks of International Business Machines Corporation, and might also be trademarks or registered trademarks in other countries.

AppScan®
Bluemix®
Cloudant®

IBM®
PureApplication®
Redbooks®

Redbooks (logo) ®
WebSphere®

The following terms are trademarks of other companies:

Resilient, and Resilient SystemsÆ are trademarks or registered trademarks of Resilient Systems, an IBM Company.

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

Java, and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

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

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

Preface

Microservices is an architectural style in which large, complex software applications are composed of one or more smaller services. Each of these microservices focuses on completing one task that represents a small business capability. These microservices can be developed in any programming language.

This IBM® Redbooks® publication covers Microservices best practices for Java. It focuses on creating cloud native applications using the latest version of IBM WebSphere® Application Server Liberty, IBM Bluemix® and other Open Source Frameworks in the Microservices ecosystem to highlight Microservices best practices for Java.

Authors

This book was produced by a team of specialists from around the world working at the International Technical Support Organization, Raleigh Center.



Michael Hofmann has a Master's degree in Compute Science (Diploma from the Technical University of Munich) and a certificate in Economics (Verwaltungs- und Wirtschaftsakademie Munich). He has been working with Java and Java EE since 1999 in different positions. He is currently the manager of a software development department, but always with a hands-on mindset. He still works as senior software architect (technical project lead) in several large projects. As trainer and coach in Java EE, software architect, DevOps consultant, and as a speaker at conferences, he shares his experience with others.



Erin Schnabel is a Senior Software Engineer at IBM. She specializes in composable runtimes and microservice architectures, including the application of object-oriented and service-oriented technologies and design patterns to decompose existing software systems. Erin has over 15 years of experience in the WebSphere Application Server development organization, with 7 years spent as development lead and architect for WebSphere Liberty. Erin is passionate about the Java developers' experience, particularly concerning the role of community-driven, standards-compliant software in the cloud environment.



Katherine Stanley is a Software Engineer at IBM in the WebSphere Application Server Liberty development organization. She specializes in microservices architectures built using Java. As a part of the Liberty team, Katherine has created samples to demonstrate industry best practices for microservices. She has run workshops and presentations at European conferences, including DevovxUK in London and JFokus in Sweden. Katherine has been at IBM since 2014 and is based in the Hursley UK lab.

This project was led by:

- Margaret Ticknor is an IBM Technical Content Services Project Leader in the Raleigh Center. She primarily leads projects about WebSphere products and IBM PureApplication® System. Before joining the IBM Technical Content Services, Margaret worked as an IT specialist in Endicott, NY. Margaret attended the Computer Science program at the State University of New York at Binghamton.

Thanks to the following people for their contributions to this project:

Roland Barcia
Distinguished Engineer, CTO: Cloud Programming Models / NYC Bluemix Garage, IBM Cloud

Kyle Brown
Distinguished Engineer, CTO: Cloud Architecture, IBM Cloud

Andy Gibbs
Offering Management, IBM Cloud

Alan Little
Distinguished Engineer, Architect for WebSphere Foundation, IBM Cloud

Ozzy Osborne
Software Developer - WebSphere Liberty Profile, IBM Cloud

Ilan Pillemer
Senior Solutions Architect (Global) at CHEP Edgware, Greater London, United Kingdom

Shriram Rajagopalan
Research Staff Member, Cloud Platforms & Services, IBM Research

Kevin Sutter
STSM, Java EE, and Liberty SLE Architect, IBM Cloud

Thanks to the following people for their support of this project:

- Diane Sherman, IBM Redbooks Content Specialist
- Ann Lund, IBM Redbooks Residency Administrator

Now you can become a published author, too!

Here's an opportunity to spotlight your skills, grow your career, and become a published author—all at the same time! Join an ITSO residency project and help write a book in your area of expertise, while honing your experience using leading-edge technologies. Your efforts will help to increase product acceptance and customer satisfaction, as you expand your network of technical contacts and relationships. Residencies run from two to six weeks in length, and you can participate either in person or as a remote resident working from your home base.

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

We want our books to be as helpful as possible. Send us your comments about this book or other IBM Redbooks publications in one of the following ways:

- Use the online **Contact us** review Redbooks form found at:

ibm.com/redbooks

- Send your comments in an email to:

redbooks@us.ibm.com

- Mail your comments to:

IBM Corporation, International Technical Support Organization
Dept. HYTD Mail Station P099
2455 South Road
Poughkeepsie, NY 12601-5400

Stay connected to IBM Redbooks

- Find us on Facebook:

<http://www.facebook.com/IBMRedbooks>

- Follow us on Twitter:

<http://twitter.com/ibmredbooks>

- Look for us on LinkedIn:

<http://www.linkedin.com/groups?home=&gid=2130806>

- Explore new Redbooks publications, residencies, and workshops with the IBM Redbooks weekly newsletter:

<https://www.redbooks.ibm.com/Redbooks.nsf/subscribe?OpenForm>

- Stay current on recent Redbooks publications with RSS Feeds:

<http://www.redbooks.ibm.com/rss.html>



Overview

Application architecture patterns are changing in the era of cloud computing. A convergence of factors has led to the concept of “cloud native” applications:

- ▶ The general availability of cloud computing platforms
- ▶ Advancements in virtualization technologies
- ▶ The emergence of agile and DevOps practices as organizations looked to streamline and shorten their release cycles

To best take advantage of the flexibility of cloud platforms, cloud native applications are composed of smaller, independent, self-contained pieces that are called *microservices*.

This chapter provides a brief overview of the concepts and motivations that surround cloud native applications and microservice architectures:

- ▶ Cloud native applications
- ▶ Twelve factors
- ▶ Microservices
- ▶ Philosophy and team structure
- ▶ Examples

1.1 Cloud native applications

Cloud computing environments are dynamic, with on-demand allocation and release of resources from a virtualized, shared pool. This elastic environment enables more flexible scaling options, especially compared to the up-front resource allocation typically used by traditional on premises data centers.

According to the Cloud Native Computing Foundation, cloud native systems have the following properties:

- ▶ Applications or processes are run in software containers as isolated units.
- ▶ Processes are managed by using central orchestration processes to improve resource utilization and reduce maintenance costs.
- ▶ Applications or services (microservices) are loosely coupled with explicitly described dependencies.

Taken together, these attributes describe a highly dynamic system composed of many independent processes working together to provide business value: a distributed system.

For more information, see the Cloud Native Computing Foundation (“CNCF”) Charter at: <https://cncf.io/about/charter>

Distributed computing is a concept with roots that stretch back decades. The eight fallacies of distributed computing were drafted in 1994, and deserve a mention:

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. The network is secure.
5. Topology doesn't change.
6. There is one administrator.
7. Transport cost is zero.
8. The network is homogeneous.

Taking the fallacies into account, something becomes clear: Cloud native applications run in an environment of constant, unpredictable change, and must expect failures to occur at any time. For more information about the fallacies, see *Fallacies of Distributed Computing Explained*, available at:

<http://www.rgoarchitects.com/Files/fallacies.pdf>

1.2 Twelve factors

The 12-factor application methodology was drafted by developers at Heroku. The characteristics mentioned in the 12 factors are not specific to cloud provider, platform, or language. The factors represent a set of guidelines or best practices for portable, resilient applications that will thrive in cloud environments (specifically software as a service applications). The following are the 12 factors in brief:

1. There should be a one-to-one association between a versioned codebase (for example, an IT repository) and a deployed service. The same codebase is used for many deployments.
2. Services should explicitly declare all dependencies, and should not rely on the presence of system-level tools or libraries.

3. Configuration that varies between deployment environments should be stored in the environment (specifically in environment variables).
4. All backing services are treated as attached resources, which are managed (attached and detached) by the execution environment.
5. The delivery pipeline should have strictly separate stages: Build, release, and run.
6. Applications should be deployed as one or more stateless processes. Specifically, transient processes must be stateless and share nothing. Persisted data should be stored in an appropriate backing service.
7. Self-contained services should make themselves available to other services by listening on a specified port.
8. Concurrency is achieved by scaling individual processes (horizontal scaling).
9. Processes must be disposable: Fast startup and graceful shutdown behaviors lead to a more robust and resilient system.
10. All environments, from local development to production, should be as similar as possible.
11. Applications should produce logs as event streams (for example, writing to **stdout** and **stderr**), and trust the execution environment to aggregate streams.
12. If admin tasks are needed, they should be kept in source control and packaged alongside the application to ensure that it is run with the same environment as the application.

These factors do not have to be strictly followed to achieve a good microservice environment. However, keeping them in mind allows you to build portable applications or services that can be built and maintained in continuous delivery environments.

For more information about the 12 factors, see *The Twelve-Factor App* by Adam Wiggins, available at:

<http://12factor.net>

1.3 Microservices

Although there is no standard definition for microservices, most reference Martin Fowler's seminal paper. The paper explains that microservices are used to compose complex applications by using small, independent (autonomous), replaceable processes that communicate by using lightweight APIs that do not depend on language.

For more information, see "Microservices: A new architectural term" by Martin Fowler, which is available at:

<http://martinfowler.com/articles/microservices.html>

Each microservice in the simple example that is shown in Figure 1-1 is a 12-factor application, with replaceable backing services providing a message broker, service registry, and independent data stores. For more information, see 1.2, “Twelve factors” on page 2.

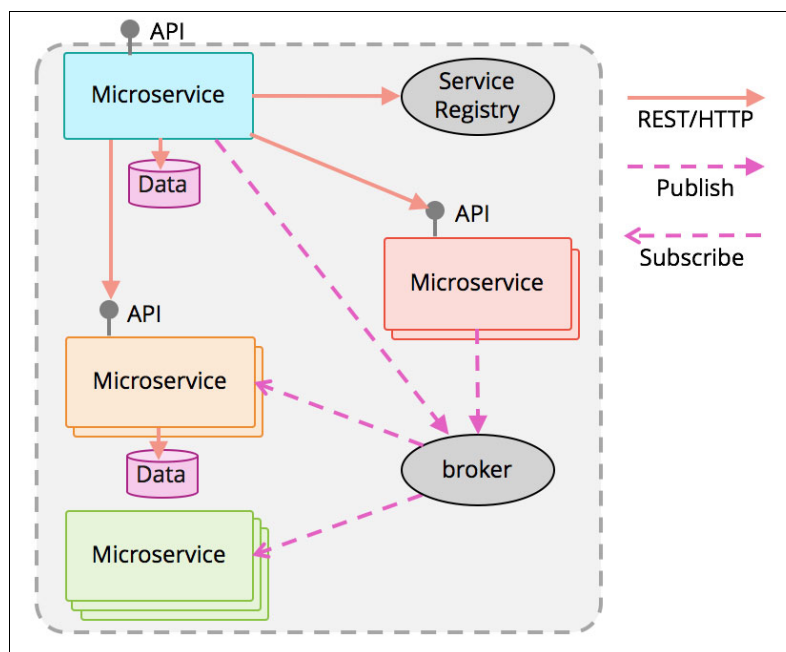


Figure 1-1 A microservices application

1.3.1 The meaning of “small”

Many descriptions make parallels between the roles of individual microservices and chained commands on the UNIX command line:

```
$ ls | grep 'service' | sort -r
```

These UNIX commands each do different things. You can use the commands without awareness of what language the commands are written in, or how large their codebases are.

The size of a microservice is not related to the number of lines of code. Instead, it describes its scope. The use of the word “small”, as applied to a microservice, essentially means that it is focused in purpose. The microservice should do one thing, and do that one thing well.

Strategies for defining the scope of a service are discussed in detail in Chapter 2, “Creating Microservices in Java” on page 9.

1.3.2 Independence and autonomy

The emotion and passion that accompany a grass roots push for creating microservices can be viewed as a mutiny against monolithic environments. This trend is due to a broad range of reasons, including these:

- ▶ Infrequent, expensive, all-or-nothing maintenance windows.
- ▶ Constrained technology choice due to well intentioned, but heavy handed, centralized governance.

In these environments, change cannot be made without significant risk because everything is interrelated with everything else. It is the opposite of an agile ecosystem.

A dominant factor in creating and preserving agility is ensuring each service is independent and autonomous. Each service must be capable of being started, stopped, or replaced at any time without tight coupling to other services. Many other characteristics of microservice architectures follow from this single factor.

Microservices should act as encapsulations (or bounded contexts, as discussed in Chapter 2, “Creating Microservices in Java” on page 9) guarding the implementation details of the application. This configuration allows the service’s implementation to be revised and improved (or even rewritten) over time. Data sources, especially, must be private to the service, as described in depth in Chapter 5, “Handling data” on page 41.

If deploying changes requires simultaneous or time sensitive updates to multiple services, you are doing it wrong. There might be good, pragmatic, reasons behind requiring these updates, but it should prompt a review of the relationships between your services. There are a few possible outcomes:

- ▶ You need to revise your versioning strategy to preserve the independent lifecycle of services.
- ▶ The current service boundaries are not drawn properly, and services should be refactored into properly independent pieces.

Refactoring services will happen, especially when creating an application from scratch. Identifying services is described more in Chapter 2, “Creating Microservices in Java” on page 9, but take it as a given that you will get some things wrong. The ability to react and refactor services as problems are discovered is a benefit of the microservices approach.

A quick note about polyglot and language-agnostic protocols

Polyglot is a frequently cited benefit of microservice-based architectures. Being able to choose the appropriate language or data store to meet the needs of each service can be powerful, and can bring significant efficiency. However, the use of obscure technologies can complicate long-term maintenance and inhibit the movement of developers between teams. Create a balance between the two by creating a list of supported technologies to choose from at the outset, with a defined policy for extending the list with new technologies over time. Ensure that non-functional or regulatory requirements like maintainability, auditability, and data security can be satisfied, while preserving agility and supporting innovation through experimentation.

Polyglot applications are only possible with language-agnostic protocols. Representational State Transfer (REST) architecture patterns define guidelines for creating uniform interfaces that separate the on-the-wire data representation from the implementation of the service. RESTful architectures require the request state to be maintained by the client, allowing the server to be stateless. The shared concepts between REST and microservices should be apparent.

For more information about REST, see “Chapter 5: Representational State Transfer (REST)”, Fielding, Roy Thomas (2000), which is available at:

http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

Hypertext Transfer Protocol (HTTP) is a common communication language for all applications on the internet. It has an expressive set of verbs and return codes that are only just realizing their potential with the adoption of REST.

REST/HTTP interactions are inherently synchronous (request/response) in nature. Asynchronous communications also play a significant role in maintaining loose coupling between microservices. Again, language-agnostic protocols come to the forefront, with standards like Advanced Message Queuing Protocol (AMQP) and Message Queuing

Telemetry Transport (MQTT) edging out language-specific mechanisms like Java Message Service (JMS). Chapter 3, “Locating services” on page 25 explores the differences between synchronous and asynchronous interaction patterns in more detail.

JavaScript Object Notation (JSON) has emerged in microservices architectures as the wire format of choice for text-based data, displacing Extensible Markup Language (XML) with its comparative simplicity and conciseness. Binary serialization frameworks do exist, such as Apache Avro¹, Apache Thrift², and Google Protocol Buffers³. However, use of these protocols should be limited to internal services and deferred until the interactions between services are understood.

1.3.3 Resilience and Fault tolerance

Creating a resilient system places requirements on all of the services in it. As mentioned earlier, the dynamic nature of cloud environments demands that services be written to expect and respond gracefully to the unexpected. This condition could mean receiving bad data, being unable to reach a required backing service, or dealing with conflicts due to concurrent updates in a distributed system.

Concerning independently evolving but inter-dependent services, the robustness principle provides the best guidance: “Be liberal in what you accept, and conservative in what you send”⁴. Assume that APIs will evolve over time, and be tolerant of data you do not understand. To quote the RFC:

As a simple example, consider a protocol specification that contains an enumeration of values ... this enumeration must be assumed to be incomplete. Thus, if a protocol specification defines four possible error codes, the software must not break when a fifth code shows up. An undefined code might be logged..., but it must not cause a failure.

Microservices must also prevent failures from cascading through the system. Chapter 4.2, “Fault tolerance” on page 37 describes strategies for dealing with API changes. Isolation patterns like circuit breakers and bulkheads are described in Chapter 3, “Locating services” on page 25.

1.3.4 Automated environment

Microservice architectures in cloud environments create a sprawling landscape of independently moving parts. Creating visualizations for interaction patterns between microservices eventually results in a confusing mess of lines. A high degree of automation is required for applications that use microservices.

Strategies for building and deploying Java based services using continuous integration or continuous deployment practices and tools are described in Chapter 8, “From development to production” on page 89.

Automated monitoring and alerting is also important. Although individual services should be less concerned about managing their log data in cloud environments, they should ensure that appropriate log data and metrics are produced to facilitate automated problem detection and alerting. This topic is explored further in Chapter 9, “Management and Operations” on page 107.

¹ <https://avro.apache.org/>

² <https://thrift.apache.org/>

³ <https://developers.google.com/protocol-buffers/>

⁴ “1.2.2 Robustness Principle”, RFC 1122, <https://tools.ietf.org/html/rfc1122#page-12>

1.4 Philosophy and team structure

There is a philosophical aspect to the relationship between microservices and team organization. This aspect comes from the application of Conway's law⁵, which states that “any organization that designs a system will inevitably produce a design whose structure is a copy of the organization's communication structure.” In “Building Microservices”, Sam Newman puts Conway's law into context, and devotes an entire chapter to the interaction between an organization's structure and system design.

Conway's law specifically focuses on communication and coordination between disparate groups of people. If the organization sequesters database administrators off into a group separate from application developers, the resulting architecture has a distinct data tier managed in a way that is optimal for database administrators. Application developers would then need to coordinate changes that they need with database administrators. Conversely, if the development team handles data management itself, this kind of coordination is unnecessary.

A move to building microservices applications should include a discussion about organizational structure and what changes might be necessary to streamline communication patterns to complement the creation and growth of independent microservices. This is a common thread in the convergence of DevOps and microservices. Creating an agile ecosystem that is capable of fast and frequent change requires streamlined communications, and lots of automation.

1.5 Examples

Throughout this book, we use two example applications to help explain concepts and provide perspective:

- ▶ Online retail store
- ▶ Game On!

1.5.1 Online retail store

The online retail store is an example that is often used when discussing application architectures. This book considers a set of microservices that together provide an application where users can browse for products and place orders.

The architecture includes the following services:

- ▶ Catalog service
 - Contains information about what can be sold in the store.
- ▶ Account service
 - Manages the data for each account with the store.
- ▶ Payment service
 - Provides payment processing.
- ▶ Order service
 - Manages the list of current orders and performing tasks that are required in the lifecycle of an order.

⁵ “How Do Committees Invent”, Melvin Conway, *Datamation magazine*, April 1968

There are other ways to structure services for an online retail store. This set of services was chosen to illustrate specific concepts that are described in this book. We leave the construction of alternate services as an exercise for the reader.

1.5.2 Game On!

Game On!⁶ is an extensible, text-based adventure game built by using microservices. Unlike the online store, Game On! is a real application, with code available on GitHub⁷, and a GitBook⁸ describing the architecture and tools that are used to create the game in depth. This deliberately non-standard example is used as a contrast to traditional enterprise applications.

The game is essentially a boundless two-dimensional network of interconnected rooms. Rooms are provided by third parties (for example, developers at conferences or in workshops). A set of core services provide a consistent user experience. The user logs in and uses simple text commands to interact with and travel between rooms. Both synchronous and asynchronous communication patterns are used. Interactions between players, core services, and third-party services are also secured.

The architecture of Game On! is described in Chapter 2, “Creating Microservices in Java” on page 9.

⁶ <https://game-on.org/#/>

⁷ <https://github.com/gameontext/>

⁸ <https://gameontext.gitbooks.io/gameon-gitbook/content/>



Creating Microservices in Java

The prospect of creating an application composed of microservices raises some questions in any language:

- ▶ How large should a microservice be?
- ▶ What does the notion of focused services that do one thing mean for traditional centralized governance?
- ▶ What does it do to traditional ways of modeling data?

Some of these topics are covered in more depth in later chapters. This chapter focuses on how you identify and create the microservices that compose your app, with specific emphasis on how identified candidates are converted into RESTful APIs, and then implemented in Java.

This chapter covers the following topics:

- ▶ Java platforms and programming models
- ▶ Versioned dependencies
- ▶ Identifying services
- ▶ Creating REST APIs

2.1 Java platforms and programming models

New Java applications should minimally use Java 8, and should embrace new language features like lambdas, parallel operations, and streams. Using annotations can simplify the codebase by eliminating boilerplate, but can reach a subjective point where there is “too much magic”. Try to establish a balance between code cleanliness and maintainability.

You can create perfectly functional microservices using nothing but low-level Java libraries. However, it is generally recommended to have something provide a reasonable base level of support for common, cross-cutting concerns. It allows the codebase for a service to focus on satisfying functional requirements.

2.1.1 Spring Boot

Spring Boot provides mechanisms for creating microservices based on an opinionated view of what technologies should be used. Spring Boot applications are Spring applications, and use a programming model unique to that ecosystem.

Spring Boot applications can be packaged to run as an application hosted by an application server, or run as a runnable .jar file containing the composition of dependencies and some type of embedded server (usually Tomcat). Spring Boot emphasizes convention over configuration, and uses a combination of annotations and classpath discovery to enable additional functions.

Spring Cloud is a collection of integrations between third-party cloud technologies and the Spring programming model. Some are integrations between the spring programming model and third-party technologies. In fewer cases, Spring Cloud technologies are usable outside of the Spring programming model, like Spring Cloud Connectors Core.

For more information about Spring Cloud Connectors, see the following website:

<http://cloud.spring.io/spring-cloud-connectors/spring-cloud-connectors.html>

2.1.2 Dropwizard

Dropwizard is another technology for creating microservices in Java. It also has an opinionated approach, with a preselected stack that enhances an embedded Jetty servlet container with additional capabilities. The set of technologies that Dropwizard supports is comparatively narrow, although third party or community provided plug-ins are available to fill the gaps. Some of their capabilities, specifically their support for application metrics, are quite good.

2.1.3 Java Platform, Enterprise Edition

Java Platform, Enterprise Edition (Java EE) is an open, community driven standard for building enterprise applications.

Java EE technologies are suited for creating microservices, especially Java API for RESTful Web Services (JAX-RS), and Contexts and Dependency Injection (CDI) for Java EE. Annotations are used for both specifications, with support for dependency injection, lightweight protocols, and asynchronous interaction patterns. Concurrency utilities in Java EE7 further provide a straightforward mechanism for using reactive libraries like RxJava in a container friendly way.

For more information about RxJava, see:

<https://github.com/ReactiveX/RxJava>

Modern Java EE application servers (WebSphere Liberty, Wildfly, TomEE, Payara) undercut claims of painful dealings with application servers. All of these servers can produce immutable, self-contained, lightweight artifacts in an automated build without requiring extra deployment steps. Further, composable servers like WebSphere Liberty allow for explicit declaration of spec-level dependencies, allowing the run time to be upgraded without creating compatibility issues with the application.

Applications use a reduced set of Java EE technologies and are packaged with application servers like WebSphere Liberty that tailor the run time to the needs of the application. This process results in clean applications that can move between environments without requiring code changes, mock services, or custom test harnesses.

The examples in this book primarily use Java EE technologies like JAX-RS, CDI, and Java Persistence API (JPA). These specifications are easy to use, and are broadly supported without relying on vendor-specific approaches.

2.2 Versioned dependencies

The twelve factor application methodology was introduced in Chapter 1, “Overview” on page 1. In keeping with the second factor, use explicit versions for external dependencies. The build artifact that is deployed to an environment must not make assumptions about what is provided.

Build tools like Apache Maven¹ or Gradle² provide easy mechanisms for defining and managing dependency versions. Explicitly declaring the version ensures that the artifact runs the same in production as it did in the testing environment.

Tools are available to make creating Java applications that use Maven and Gradle easier. These tools accept as inputs a selection of technologies, and output project structures complete with the Maven or Gradle build artifacts that are required to build and either publish or deploy the final artifact:

- ▶ WebSphere Liberty App Accelerator
<http://wasdev.net/accelerate>
- ▶ Spring Initializr
<http://start.spring.io/>
- ▶ Wildfly Swarm Project Generator
<http://wildfly-swarm.io/generator/>

2.3 Identifying services

As mentioned in Chapter 1, “Overview” on page 1, microservices are described as being “small.” The size of a service should be reflected in its scope, rather than in the size of its codebase. A microservice should have a small, focused scope: It should do one thing, and do that thing well. Eric Evans’ book, *Domain-Driven Design*, describes many concepts that are

¹ <https://maven.apache.org/>

² <https://gradle.org/>

helpful when deciding how to build an application as a composition of independent pieces. Both the online retail store and Game On! are referenced in the next few sections to explain why these concepts are useful for developing and maintaining microservice architectures in the long term.

2.3.1 Applying domain-driven design principles

In domain-driven design, a *domain* is a particular area of knowledge or activity. A *model* is an abstraction of important aspects of the domain that emerges over time, as the understanding of the domain changes. This model is then used to build the solution, for cross-team communications.

With monolithic systems, a single unified domain model exists for the entire application. This concept works if the application remains fairly simple. Using a single unified model falls apart when different parts of your application use common elements in different ways.

For example, the Catalog service in the Online Retail Store sample is focused on the items for sale, including descriptions, part numbers, product information, pictures, and so on. The Order service, by contrast, is focused on the invoice, with items represented minimally, perhaps only the part number, summary, and calculated price.

Bounded contexts allow a domain to be subdivided into independent subsystems. Each bounded context can then have its own models for domain concepts. Continuing the above example, both the Catalog and Order services can each have independent representations of items that best fit the needs of their service. This system usually works great, but how do the two services share information with other services, given they have different representations for the same concept?

Context mapping, including the development and use of *ubiquitous language*, can help ensure that disparate teams working on individual services understand how the big picture should fit together. In the case of modeling a domain for a microservices application, it is important not to get down to the class or interface level when identifying the pieces that comprise the system. For the first pass, keep things at a high, coarse level. Allow individual teams working on the individual services work within their bounded contexts to expand on the details.

Ubiquitous language in Game On!

To give a concrete example of the importance of language, we take a small detour into how Game On! was developed. The game was built to make it easier for developers to experiment with microservices concepts first hand, combining the instant gratification of a quick hello world example with different ways to think about the more complex aspects of microservices architectures. It is a text-based adventure game in which developers add their own individual services (rooms) to extend the game.

The original metaphor for the game was akin to a hotel, with three basic elements:

- ▶ A Player service to deal with all things relating to game players
- ▶ An initial Room service as a placeholder for what developers will build
- ▶ An additional service called the Concierge, the original purpose of which was to help facilitate the transition of a player from room to room

After letting these three pieces of the system evolve over time, it was discovered that different people working on the game had different interpretations of what the Concierge was supposed to do. The Concierge did not work as a metaphor because it was too ambiguous.

The Concierge was replaced with a new Map service. The Map is the ultimate source of truth for where a room is located in the game, which is clearly understood from the name alone.

2.3.2 Translating domain elements into services

Domain models contain a few well-defined types:

- ▶ An *Entity* is an object with a fixed identity and a well-defined “thread of continuity” or lifecycle. An frequently cited example is a Person (an Entity). Most systems need to track a Person uniquely, regardless of name, address, or other attribute changes.
- ▶ *Value Objects* do not have a well-defined identity, but are instead defined only by their attributes. They are typically immutable so that two equal value objects remain equal over time. An address could be a value object that is associated with a Person.
- ▶ An *Aggregate* is a cluster of related objects that are treated as a unit. It has a specific entity as its root, and defines a clear boundary for encapsulation. It is not just a list.
- ▶ *Services* are used to represent operations or activities that are not natural parts of entities or value objects.

Domain elements in Game On!

As mentioned in 1.3.1, “The meaning of “small” on page 4, the game was built around the notion of a Player, an obvious Entity. Players have a fixed ID, and exist for as long as the player’s account does. The player entity has a simple set of value objects, including the player’s user name, favorite color, and location.

The other significant element of the game, rooms, do not map so obviously to an Entity. For one thing, room attributes can change in various ways over time. These changeable attributes are represented by a Value Object containing attributes like the room’s name, descriptions of its entrances, and its public endpoint. Further, the placement of rooms in the map can change over time. If you consider each possible room location as a Site, you end up with the notion of a Map as an Aggregate of unique Site entities, where each Site is associated with changeable room attributes.

Mapping domain elements into services

Converting domain elements into microservices can be done by following a few general guidelines:

- ▶ Convert Aggregates and Entities into independent microservices, by using representations of Value Objects as parameters and return values.
- ▶ Align Domain services (those not attached to an Aggregate or Entity) with independent microservices.
- ▶ Each microservice should handle a single complete business function.

This section explores this process by applying these guidelines to the domain elements listed above for Game On!, as shown in Figure 2-1.

The Player service provides the resource APIs³ to work with Player entities. In addition to standard create, read, update, and delete operations, the Player API provides additional operations for generating user names and favorite colors, and for updating the player's location.

The Map service provides the resource APIs⁴ to work with Site entities, allowing developers to manage the registration of room services with the game. The Map provides create, retrieve, update, and delete operations for individual Sites within a bounded context. Data representations inside the Map service differ from those shared outside the service, as some values are calculated based on the site's position in the map.

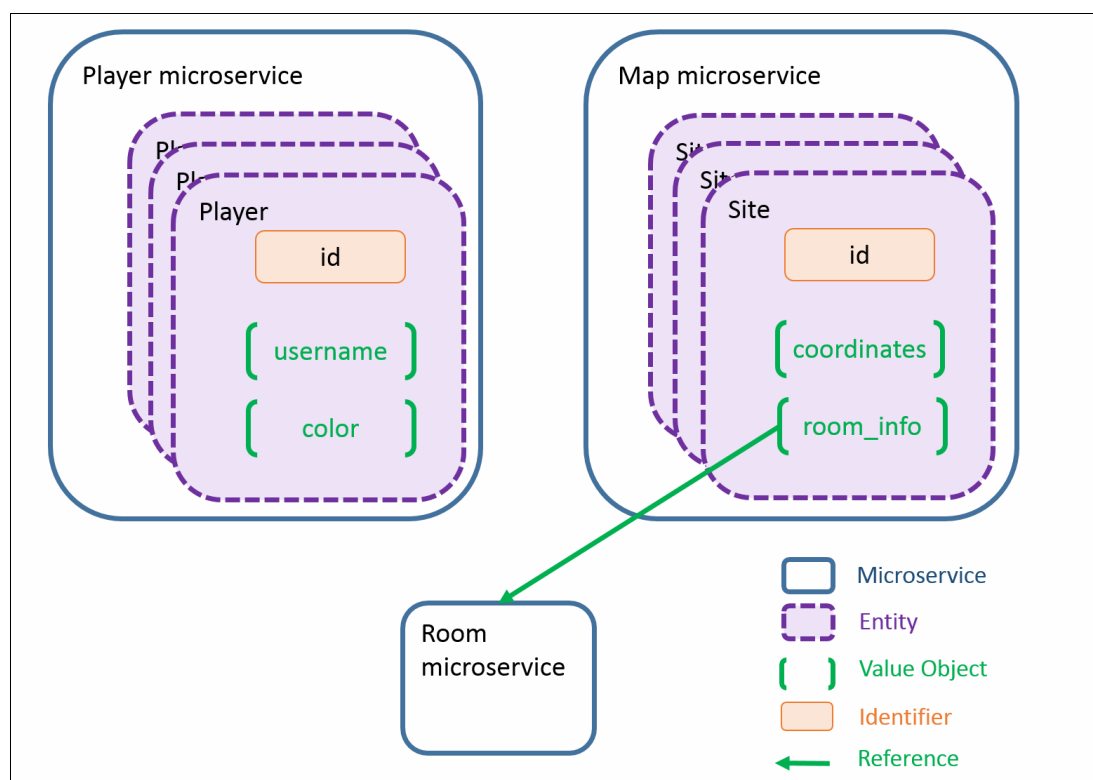


Figure 2-1 The Player and Map services

Some additional services from the game have not been discussed:

- ▶ **Mediator:** The Mediator is a service that splintered out of the first implementation of the Player service. Its sole responsibility is mediating between these WebSocket connections:
 - One long running connection between the client device and the Mediator.
 - Another connection between the Mediator and the target independent room service.
 From a domain modeling point of view, the Mediator is a domain service.
- ▶ **Rooms:** What about individual room services? Developers stand up their own instances of rooms. For modeling purposes, treat rooms, even your own, as outside actors.

³ <https://game-on.org/swagger/#!/players> Game On! Player API

⁴ <https://game-on.org/swagger/#!/map> Game On! Map API

2.3.3 Application and service structure

Monolithic Java EE applications are typically structured and deployed in tiers (client, web, business, and data), with long lived application servers hosting web archives (WAR files), enterprise archives (EAR files), or both for disparate parts of an application, as shown in the left side of Figure 2-2. Naming and packaging conventions become important as JSPs, JSFs, servlets, and EJBs for different parts of the application are deployed to the same application server. Common data stores provide consistency for data representation, but also introduce constraints in how data can be represented and stored.

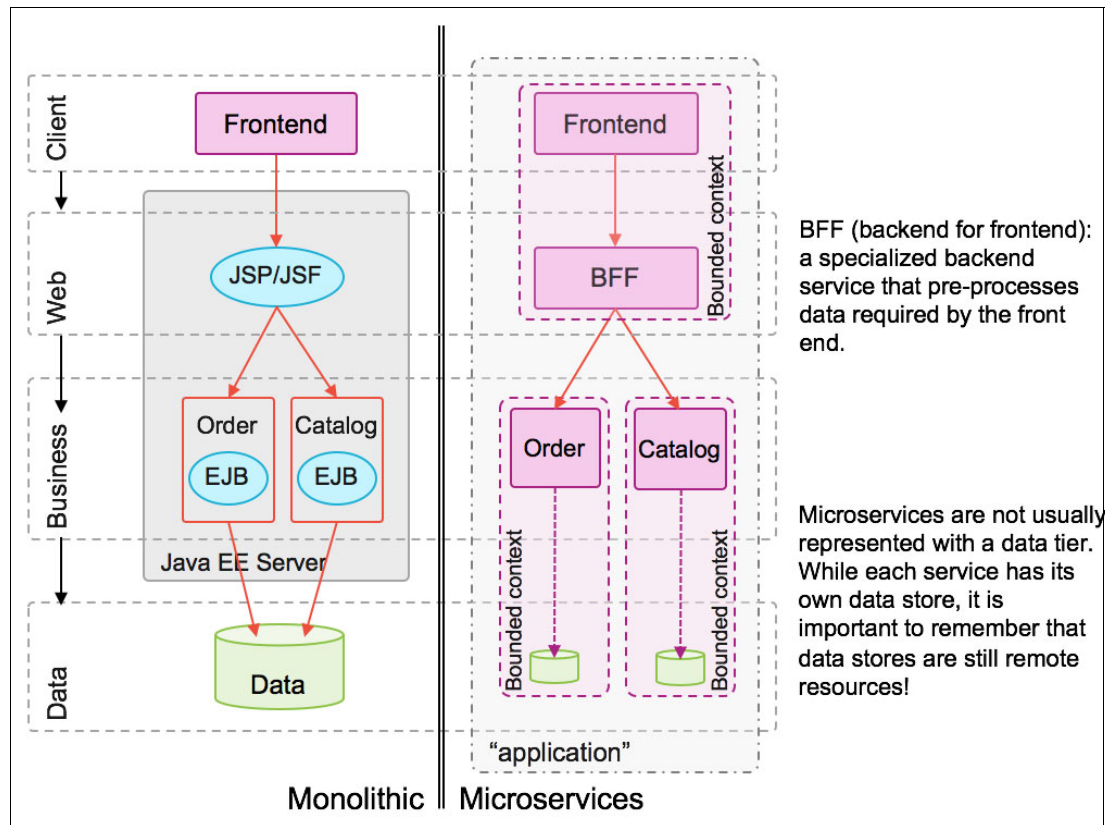


Figure 2-2 Order and Catalog Services as part of monolithic or microservices application

A microservices architecture will likely still have these application tiers, but the elements of each tier will likely be provided by independent services that run in independent processes, as shown in the right side of Figure 2-2. Note these details:

- ▶ The backend-for-frontend pattern is commonly used to support specialized behavior for specific front ends, like optimizations for specific devices or providing additional capabilities in a web application.
- ▶ Although each service owns and maintains its own data store, remember that the data store is itself an independent service.

Not all microservice applications will have all of these elements. For example, web applications that use JSPs or JSF, or interpreted scripts like PHP run in the backend. They could be considered as “backend-for-frontend”, especially if they have a close relationship with JavaScript running in the browser. Game On! however, does not use the backend-for-frontend pattern (at the time of this writing). The JavaScript single page web application is served as a set of static resources from an independent service. The application then runs in the browser, and calls backend APIs directly.

Internal structure

Structure your code inside a service so that you have a clear separation of concerns to facilitate testing. Calls to external services should be separate from the domain logic, which should also be separate from logic specific to marshalling data to/from a backing data service.

Figure 2-3 shows a simplified version of a microservice architecture. This architecture separates the domain logic from any code that interacts with or understands external services. The architecture is similar to the Ports and Adapters (Hexagonal) architecture⁵, and to Toby Clemson’s approach in “Testing Strategies in a Microservice Architecture”⁶.

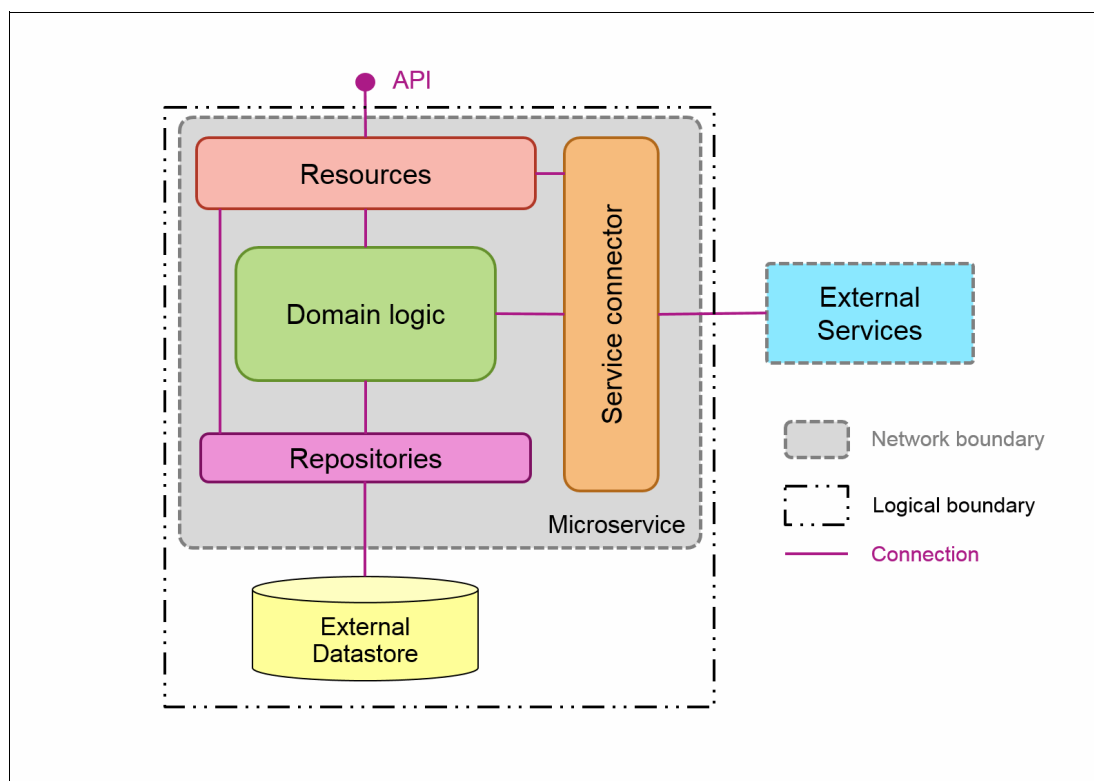


Figure 2-3 The internal structure of a microservice

The example internal structure considers the following elements:

- ▶ Resources expose JAX-RS resources to external clients. This layer handles basic validation of requests and then passes the information into the domain logic layer.
- ▶ Domain logic, as a general concept, takes many forms. In Boundary-Entity-Control patterns, domain logic represents the entity itself, with parameter validation, state change logic, and so on.
- ▶ Repositories are optional, but can provide a useful abstraction between the core application domain logic and the data store when present. This configuration allows the backing data store to be changed or replaced without extensive changes to the domain logic.
- ▶ Service connectors are similar to the Repository abstraction, encapsulating communications with other services. This layer functions as either a façade or an “anti-corruption” layer to protect domain logic from changes to external resource APIs, or to convert between API wire formats and internal domain model constructs.

⁵ <http://alistair.cockburn.us/Hexagonal+architecture>

⁶ <http://martinfowler.com/articles/microservice-testing/#anatomy-modules>

This architecture does require you to follow a few rules when creating classes. For example, each class that you use to implement your service should perform one of these tasks:

- ▶ Perform domain logic
- ▶ Expose resources
- ▶ Make external calls to other services
- ▶ Make external calls to a data store

These are general recommendations for code structure that do not have to be followed strictly. The important characteristic is to reduce the risk of making changes. For example, if you want to change the data store, you only need to update the repository layer. You do not have to search through every class looking for the methods that call the data store. If there are API changes to external services, these changes can similarly be contained within the service connector. This separation further simplifies testing, as described in Chapter 7, “Testing” on page 77.

2.3.4 Shared library or new service?

A common principle in both Agile and Object Oriented methodologies is “don’t repeat yourself” (DRY). Repeating the same code multiple times generally indicates that code should be turned into a reusable “thing.” In traditional OO, this thing is an object, or perhaps a utility class. However, especially in the case of ubiquitous utility classes, strict application of DRY principles is the end of neatly decoupled modules.

Microservices development makes this backsliding into code spaghetti more difficult. Hard process boundaries make oversharing of implementation details much harder, but sadly not impossible. So, what do you do with common code?

- ▶ Accept that there might be some redundancy
- ▶ Wrap common code in a shared, versioned library
- ▶ Create an independent service

Depending on the nature of the code, it might be best to accept redundant code and move on. Given that each service can and should evolve independently, the needs of the service over time might change the shape of that initially common code.

It is especially tempting to create shared libraries for Data Transfer Objects, which are thin classes that serve as vehicles to change to and from the wire transfer format (usually JSON). This technique can eliminate a lot of boiler plate code, but it can leak implementation details and introduce coupling.

Client libraries are often recommended to reduce duplication and make it easier to consume APIs. This technique has long been the norm for data stores, including new NoSQL data stores that provide REST-based APIs. Client libraries can be useful for other services as well, especially those using binary protocols. However, client libraries do introduce coupling between the service consumer and provider. The service may also be harder to consume for platforms or languages that do not have well-maintained libraries.

Shared libraries can also be used to ensure consistent processing for potentially complex algorithms. For Game On!, for example, a shared library is used to encapsulate request signing and signature verification, which is described in more detail in Chapter 6, “Application Security” on page 67. This configuration allows you to simplify services within the system and ensure that values are computed consistently. However, we do not provide a library for all languages, so some services still must do it the hard way.

When should a shared library really be an independent service? An independent service is independent of the language used, can scale separately, and can allow updates to take effect immediately without requiring coordinated updates to consumers. An independent service also adds extra processing, and is subject to all of the concerns with inter-service communication that are mentioned in Chapter 4, “Microservice communication” on page 33. It is a judgment call when to make the trade off, although if a library starts requiring backing services, that is generally a clue that it is a service in its own right.

2.4 Creating REST APIs

So, what does a good microservices API look like? Designing and documenting your APIs is important to ensure that it is useful and usable. Following conventions and using good versioning policies can make the API easier to understand and evolve over time.

2.4.1 Top down or bottom up?

When creating a service from scratch, you can either define the APIs that you need first and write your code to fulfill the APIs, or write the code for the service first and derive your APIs from the implementation.

In principle, the first option is always better. A microservice does not operate in isolation. Either it is calling something, or something is calling it. The interactions between services in a microservices architecture should be well defined and well documented. Concerning domain driven design, APIs are the known intersection points between bounded contexts.

As Alan Perlis says, “Everything should be built from the top down, except the first time⁷.” When doing a proof of concept, you might be unclear about what APIs you are providing. In those early days, write something that works to help refine ideas about what it is that you are trying to build. However, after you have a better understanding of what your service needs to do, you should go back and refine the domain model and define the external APIs to match.

2.4.2 Documenting APIs

Using tools to document your APIs can make it easier to ensure accuracy and correctness of the published documentation, which can then be used in discussions with API consumers. This documentation can also be used for your consumer driven contract tests (see Chapter 7, “Testing” on page 77).

The Open API Initiative (OAI)⁸ is a consortium focused on standardizing how RESTful APIs are described. The OpenAPI specification⁹ is based on Swagger¹⁰, which defines the structure and format of metadata used to create a Swagger representation of a RESTful API. This definition is usually expressed in a single, portable file (`swagger.json` by convention, though YAML is supported by using `swagger.yaml`). The swagger definition can be created by using visual editors or generated based on scanning annotations in the application. It can further be used to generate client or server stubs.

⁷ <http://www.cs.yale.edu/homes/perlis-alan/quotes.html>

⁸ <https://openapis.org/>

⁹ <https://github.com/OAI/OpenAPI-Specification>

¹⁰ <http://swagger.io/>

The apiDiscovery feature¹¹ in WebSphere Application Server Liberty integrates support for Swagger definitions into the run time. It automatically detects and scans JAX-RS applications for annotations that it uses to generate a Swagger definition dynamically. If Swagger annotations are present, information from those annotations is included to create informative documentation that can be maintained inline with the code. This generated documentation is made available by using a system provided endpoint, as shown in Figure 2-4.

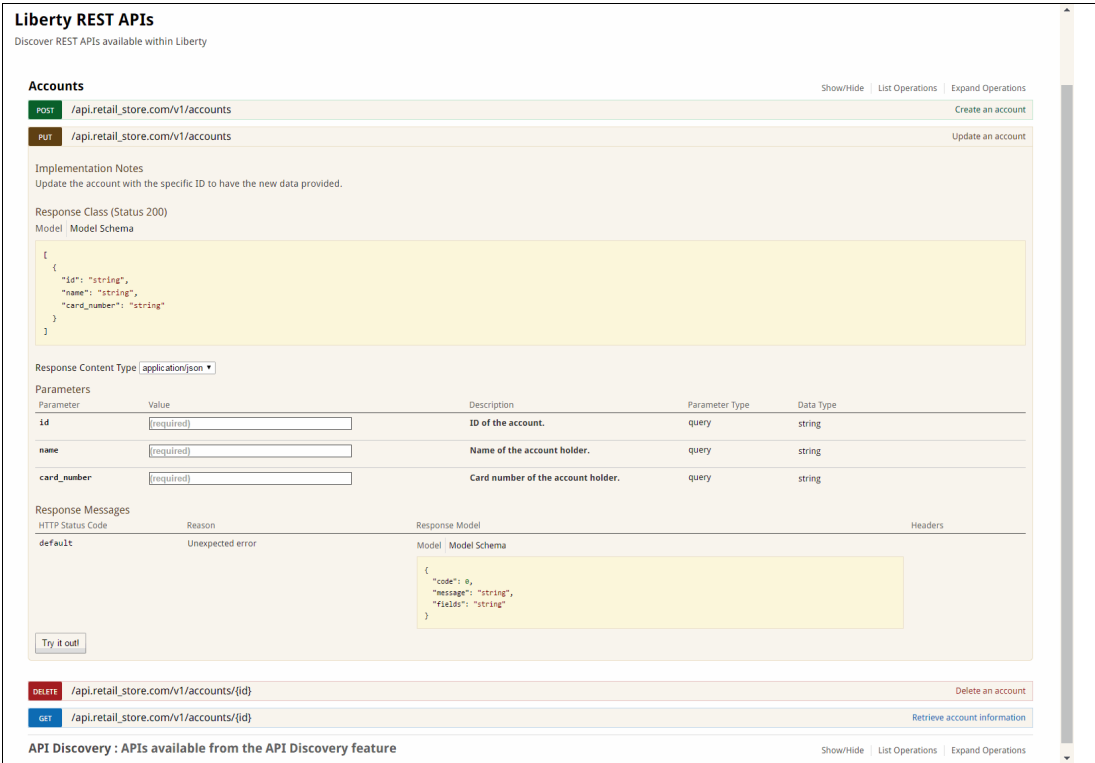


Figure 2-4 The user interface provided by the apiDiscovery-1.0 feature to document APIs

The Swagger API definition, especially as rendered in a web interface, is useful for visualizing the external, consumer view of the API. Developing or designing the API from this external view helps ensure consistency. This external view aligns with Consumer Driven Contracts¹², a pattern that advocates designing APIs with a focus on consumer expectations.

2.4.3 Use the correct HTTP verb

REST APIs should use standard HTTP verbs for create, retrieve, update, and delete operations, with special attention paid to whether the operation is idempotent (safe to repeat multiple times).

POST operations may be used to Create(C) resources. The distinguishing characteristic of a POST operation is that it is not idempotent. For example, if a POST request is used to create resources, and it is started multiple times, a new, unique resource should be created as a result of each invocation.

¹¹ <https://developer.ibm.com/wasdev/blog/2016/02/17/exposing-liberty-rest-apis-swagger/>

¹² <http://martinfowler.com/articles/consumerDrivenContracts.html>

GET operations must be both idempotent and nullipotent. They should cause no side effects, and should only be used to Retrieve(R) information. To be specific, GET requests with query parameters should not be used to change or update information (use POST, PUT, or PATCH instead).

PUT operations can be used to Update(U) resources. PUT operations usually include a complete copy of the resource to be updated, making the operation idempotent.

PATCH operations allow partial Update(U) of resources. They might or might not be idempotent depending on how the delta is specified and then applied to the resource. For example, if a PATCH operation indicates that a value should be changed from A to B, it becomes idempotent. It has no effect if it is started multiple times and the value is already B. Support for PATCH operations is still inconsistent. For example, there is no @PATCH annotation in JAX-RS in Java EE7.

DELETE operations are not surprisingly used to Delete(D) resources. Delete operations are idempotent, as a resource can only be deleted once. However, the return code varies, as the first operation succeeds (200), while subsequent invocations do not find the resource (204).

2.4.4 Create machine-friendly, descriptive results

An expressive REST API should include careful consideration of what is returned from started operations. Given that APIs are started by software instead of by users, care should be taken to communicate information to the caller in the most effective and efficient way possible.

As an example, it used to be common practice to return a 200 (OK) status code along with HTML explaining the error message. Although this technique might work for users viewing web pages (for whom the HTTP status code is hidden anyway), it is terrible for machines determining whether their request succeeded.

The HTTP status code should be relevant and useful. Use a 200 (OK) when everything is fine. When there is no response data, use a 204 (NO CONTENT) instead. Beyond that technique, a 201 (CREATED) should be used for POST requests that result in the creation of a resource, whether there is a response body or not. Use a 409 (CONFLICT) when concurrent changes conflict, or a 400 (BAD REQUEST) when parameters are malformed. The set of HTTP status codes is robust, and should be used to express the result of a request as specifically as possible. For more information, see the following website:

<https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>

You should also consider what data is being returned in your responses to make communication efficient. For example, when a resource is created with a POST request, the response should include the location of the newly created resource in a Location header. You do not need to return additional information about the resource in the response body. However, in practice the created resource is usually included in the response because it eliminates the need for the caller to make an extra GET request to fetch the created resource. The same applies for PUT and PATCH requests.

2.4.5 Resource URIs and versioning

There are varying opinions about some aspects of RESTful resource URIs. In general, resources should be nouns, not verbs, and endpoints should be plural. This technique results in a clear structure for create, retrieve, update, and delete operations:

POST /accounts	Create a new item
GET /accounts	Retrieve a list of items

GET /accounts/16	Retrieve a specific item
PUT /accounts/16	Update a specific item
PATCH /accounts/16	Update a specific item
DELETE /accounts/16	Delete a specific item

The general consensus is to keep things simple with URIs. If you create an application that tracks a specific kind of rodent, the URI would be */mouses/* and not */mice/*, even if it makes you wince.

Relationships are modeled by nesting URIs, for example, something like */accounts/16/credentials* for managing credentials associated with an account.

Where there is less agreement is with what should happen with operations that are associated with the resource, but that do not fit within this usual structure. There is no single correct way to manage these operations. Do what works best for the consumer of the API.

For example, with Game On!, the Player service provides operations that assist with generating user names and favorite colors. It is also possible to retrieve, in bulk, the location of every player within the game. The structure that was chosen reflects operations that span all players, while still allowing you to work with specific players. This techniques has its limitations, but illustrates the point:

GET /players/accounts	Retrieve a list of all players
POST /players/accounts	Create a new player
GET /players/accounts/{id}	Retrieve a specific player
GET /players/accounts/{id}/location	Retrieve a specific player's location
PUT /players/accounts/{id}/location	Update a specific player's location
GET /players/color	Return 10 generated colors
GET /players/name	Return 10 generated names
GET /players/locations	Retrieve the location of all players

Versioning

One of the major benefits of microservices is the ability to allow services to evolve independently. Given that microservices call other services, that independence comes with a giant caveat. You cannot cause breaking changes in your API.

The easiest approach to accommodating change is to never break the API. If the robustness principle is followed, and both sides are conservative in what they send and liberal in what they receive, it can take a long time before a breaking change is required. When that breaking change finally comes, you can opt to build a different service entirely and retire the original over time, perhaps because the domain model has evolved and a better abstraction makes more sense.

If you do need to make breaking API changes for an existing service, decide how to manage those changes:

- ▶ Will the service handle all versions of the API?
- ▶ Will you maintain independent versions of the service to support each version of the API?
- ▶ Will your service support only the newest version of the API and rely on other adaptive layers to convert to and from the older API?

After you decide on the hard part, the much easier problem to solve is how to reflect the version in your API. There are generally three ways to handled versioning a REST resource:

- ▶ Put the version in the URI
- ▶ Use a custom request header
- ▶ Put the version in the HTTP Accept header and rely on content negotiation

Put the version in the URI

Adding the version into the URI is the easiest method for specifying a version. This approach has these advantages:

- ▶ It is easy to understand.
- ▶ It is easy to achieve when building the services in your application.
- ▶ It is compatible with API browsing tools like Swagger and command-line tools like curl.

If you're going to put the version in the URI, the version should apply to your application as a whole, so use, for example, `/api/v1/accounts` instead of `/api/accounts/v1`. Hypermedia as the Engine of Application State (HATEOAS) is one way of providing URIs to API consumers so they are not responsible for constructing URIs themselves. GitHub, for example, provides hypermedia URLs in their responses for this reason.¹³ HATEOAS becomes difficult if not impossible to achieve if different backend services can have independently varying versions in their URIs.

If you decide to place the version in your URI, you can manage it in a few different ways, partly determined by how requests are routed in your system. However, assume for a moment that your gateway proxy converts the external `/api/v1/accounts` URI to `/accounts/v1`, which maps to the JAX-RS endpoint provided by your service. In this case, `/accounts` is a natural context root. After that you can either include the version in the `@ApplicationPath` annotation, as shown in Example 2-1, which works well when the application is supporting only one version of the API.

Example 2-1 Using the `@ApplicationPath` annotation to add the version to the URI

```
package retail_store.accounts.api;
@ApplicationPath("/v1")
public class ApplicationClass extends Application {}
```

Alternately, if the service should support more than one version of the application, you can push the version to `@Path` annotations, as shown in Example 2-2.

Example 2-2 The `@ApplicationPath` defines a rest endpoint without changing the URI

```
package retail_store.accounts.api.v1;
@ApplicationPath("/")
public class ApplicationClass extends Application {}
```

Example 2-3 shows the version added to the annotation.

Example 2-3 The version is added to the API using the `@Path` annotation

```
package retail_store.accounts.api.v1;
@Path("v1/accounts")
public class AccountsAPI {
    @GET
    @Path("{id}")
    @Produces(MediaType.APPLICATION_JSON)
    public Response getAccounts(@PathParam("id") String id) {
        getId(id);
        ...
    }
}
```

¹³ <https://developer.github.com/v3/#hypermedia>

One of the arguments against including the version in the URL is a strict interpretation of the HTML standard that a URL should represent the entity and if the entity being represented didn't change, the URL should not change.

Another concern is that putting versions in the URI requires consumers to update their URI references. This concern can be partly resolved by allowing requests to be made without a version, and then mapping those requests to the latest version. However, this approach is prone to unexpected behavior when the latest version changes.

Add a custom request header

You can add a custom request header to indicate the API version. Custom headers can be taken into consideration by routers and other infrastructure to route traffic to specific backend instances. However, this mechanism is not as easy to use for all of the same reasons that Accept headers are not easy to use. In addition, it is a custom header that only applies to your application, which means consumers need to learn how to use it.

Modify the Accept header to include the version

The Accept header is an obvious place to define a version, but is one of the most difficult to test. URLs are easy to specify and replace, but specifying HTTP headers requires more detailed API and command-line invocations.

Plenty of articles describe the advantages and disadvantages of all three methods. This blog post by Troy Hunt is a good place to start:

<https://www.troyhunt.com/your-api-versioning-is-wrong-which-is/>

The following are the most important things to take away from this chapter:

- ▶ Design your APIs from the consumer point of view.
- ▶ Have a strategy for dealing with API changes.
- ▶ Use a consistent versioning technique across all of the services in your application.



Locating services

Microservices are designed to be easily scaled. This scaling is done horizontally by scaling individual services. With multiple instances of microservices, you need a method for locating services and load balancing over the different instances of the service you are calling. This chapter describes the options available for locating and making requests to microservices in a system.

The following topics are covered:

- ▶ Service registry
- ▶ Service invocation
- ▶ API Gateway

3.1 Service registry

A service registry is a persistent store that holds a list of all the available microservices at any time and the routes that they can be reached on. There are four reasons why a microservice might communicate with a service registry:

- ▶ Registration

After a service has been successfully deployed, the microservice must be registered with the service registry.

- ▶ Heartbeats

The microservice should send regular heartbeats to the registry to show that it is ready to receive requests.

- ▶ Service discovery

To communicate with another service, a microservice must call the service registry to get a list of available instances (see 3.2, “Service invocation” on page 27 for more details).

- ▶ De-registration

When a service is down, it must be removed from the list of available services in the service registry.

3.1.1 Third-party registration versus self-registration

The registration of the microservice with the service registry can be done by the microservice or by a third party. Using a third party requires said party to inspect the microservice to determine the current state and relay that information to the service registry. The third party also handles de-registration. If the microservice itself performs registration and heartbeats, the service registry can unregister the service if it misses a heartbeat.

The advantage of using a third party is that the registration and heartbeat logic is kept separate from the business logic. The disadvantage is that there is an extra piece of software to deploy and the microservice must be exposed to a health endpoint for the third party to poll.

Using self-registration pulls the registration and heartbeat logic into the microservice itself. Using this method requires careful testing considerations. See Chapter 7, “Testing” on page 77 for more information about separation of code. However, many service registry solutions provide a convenience library for registration, reducing the complexity of the code required.

Generally, use a service registry that provides a Java based library for registration and heartbeats. This configuration makes your services quicker to code and provide consistency across every microservice in the system (that is in Java). The following service registry solutions are available:

- ▶ Consul

<https://www.consul.io/>

- ▶ Eureka

<https://github.com/Netflix/eureka>

- ▶ Amalgam8

<https://github.com/amalgam8/>

All of these registries are open source and provide integration for Java. As a part of the Netflix stack, Eureka is well integrated with the other Netflix solutions for load balancing and fault tolerance. Using Eureka as the service registry allows you to easily add the other solutions. The Amalgam8 open source project provides both service registration and load balancing. It also provides routing configuration that can be used during testing. For more information, see Chapter 7, “Testing” on page 77.

3.1.2 Availability versus consistency

Most service registries provide partition tolerance and either consistency or availability. They cannot provide all three due to the CAP theorem¹. For example, Eureka provides availability and both Consul and Apache Zookeeper² provide consistency. Some articles argue that you should always choose one or the other (this blog post³ on the Knewton blog is worth a read), but the decision comes down to the precise needs of your application. If you require all of your microservices to have the same view of the world at a time, then choose consistency. If it is essential that requests are answered quickly rather than waiting for a consistent answer, then choose availability. Solutions are available for both that integrate well with Java.

3.2 Service invocation

When a service needs to communicate with another service, it uses the information stored in the service registry. The call to the actual microservice is then either made on the server side or the client side.

3.2.1 Server side

Server-side communication with the microservice is performed by using a service proxy. The service proxy is either provided as part of the service registry, or as a separate service. When a microservice needs to make a call, it calls the service proxy's well-known endpoint. The service proxy is responsible for getting the location of the microservice from the service registry and forwarding the request. The proxy takes the response and routes it back to the original microservice that made the request. In the example scenario, load balancing is handled completely on the server side.

¹ https://en.wikipedia.org/wiki/CAP_theorem

² <https://zookeeper.apache.org/>

³ <https://tech.knewton.com/blog/2014/12/eureka-shouldnt-use-zookeeper-service-discovery/>

Figure 3-1 shows the flow of requests when using a service proxy.

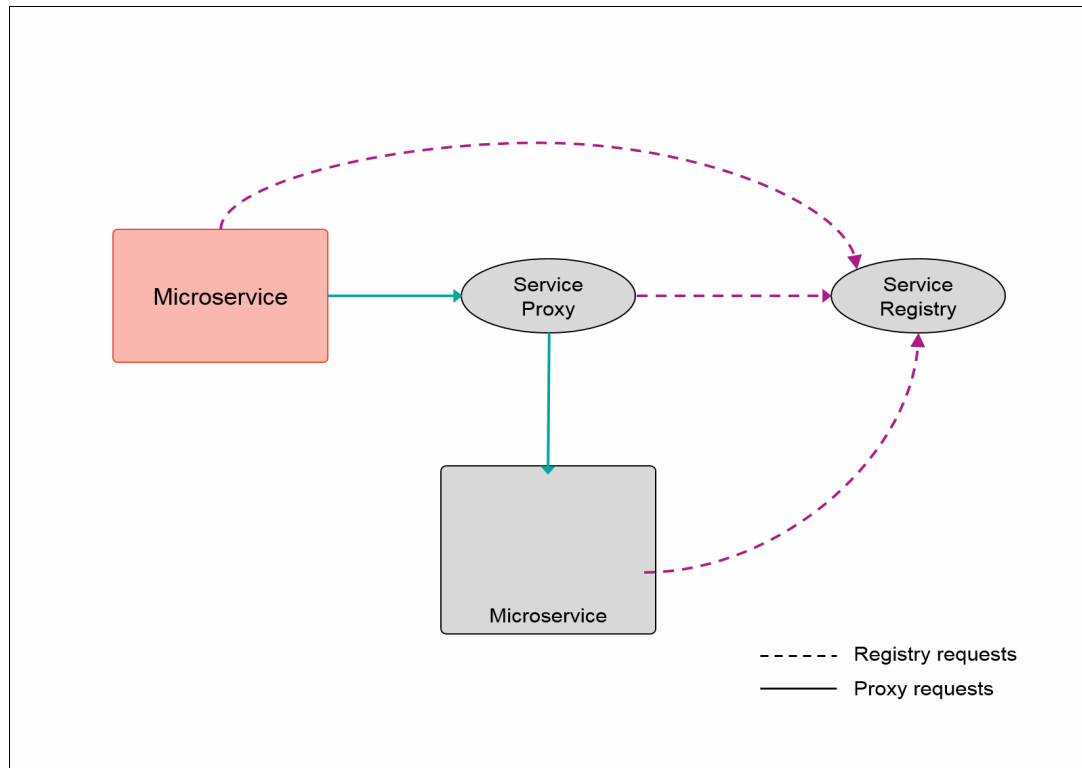


Figure 3-1 Calls to other microservices are made by using the service proxy

Using server-side invocation provides these key advantages:

- Simple requests

The request made by the microservice can be simple because it is calling a well-known endpoint.

- Easier testing

Using a server proxy takes any load balancing or routing logic out of the microservice. This configuration allows you to test the full function of the service using a mocked proxy.

In cloud platforms such as Cloud Foundry and Bluemix, the proxy is provided by the platform. In Bluemix, every deployed application has a well-known endpoint, called a *route*. When another service or application calls this route, Bluemix does load balancing in the background.

This approach has the following key disadvantage:

- Greater number of hops

Adding in a call to the service proxy increases the number of network hops per request. As a result, the number of hops per request is generally higher than for client-side solutions. It is worth noting that the service proxy can hold cached information about the other microservices, so in many cases the request would go through four hops: 1 → 4 → 5 → 6.

Figure 3-2 shows the network hops required.

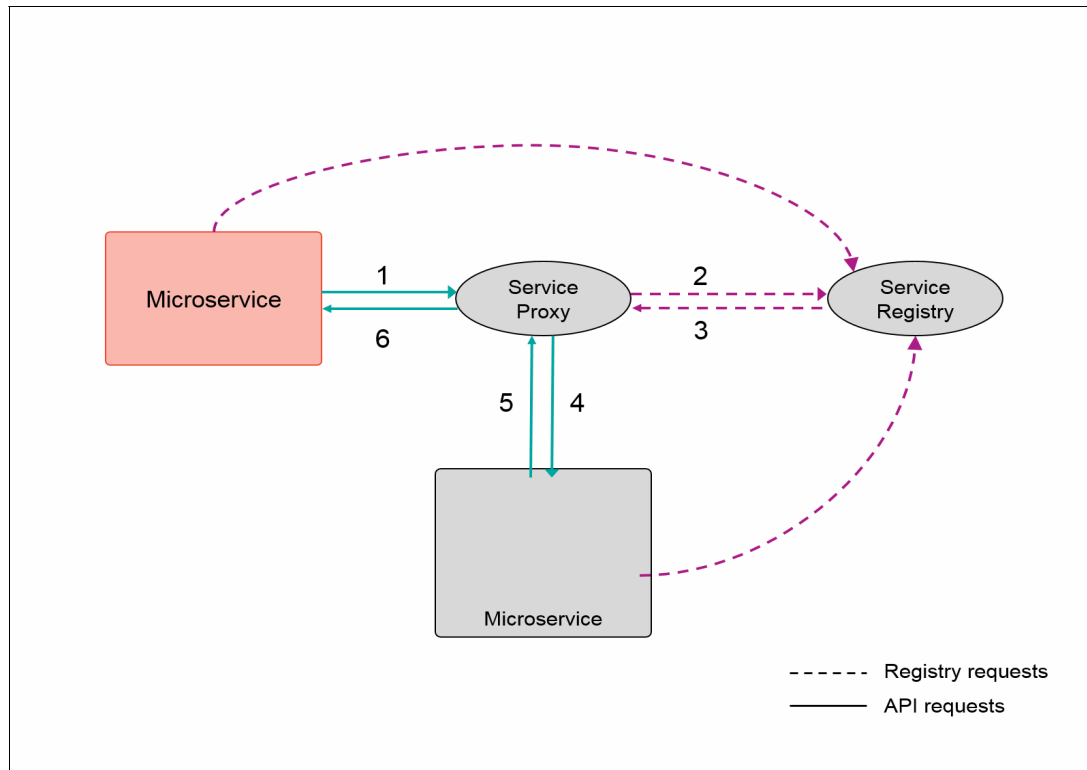


Figure 3-2 Network hops are required to complete a request service when using a service proxy

3.2.2 Client side

The request from one microservice to another can be made directly from the client side. First, the location of one or several instances of a service is requested from the service registry. Second, the request to the microservice is made on the client side. The location of the microservice is usually cached so that, in the future, a request can be made without returning to the service registry. If in the future the request fails, then the client can repeat the call to the service registry. It is a best practice to have the cached microservice location on a timeout. This configuration means that if a new version of a service is deployed, the other microservices do not have to wait for an error from their cached instance before being informed about the new instance.

In comparison to the service proxy approach, this approach requires fewer network hops. Figure 3-3 shows the network hops required. Again, this number of hops is for when the client does not have cached information about the services. In many cases, the request to the service registry is not required, making it just two hops: 3 → 4.

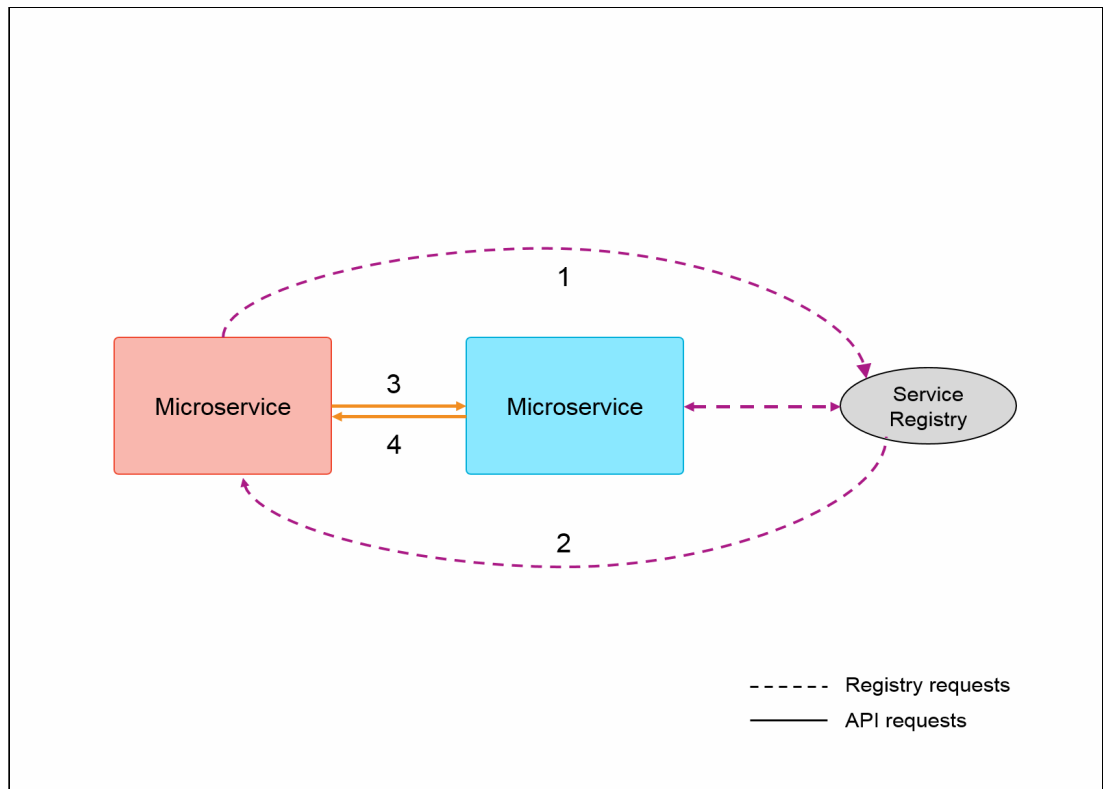


Figure 3-3 Network hops when making a client-side request to the service registry

The request and any load balancing that is required can be handled by one of two mechanisms:

- ▶ Client library
- ▶ Sidecar

Both mechanisms make requests on the client side. However, the client library runs within the microservice and the sidecar is deployed with the microservice, but runs on a separate process. Figure 3-4 shows an example architecture diagram.

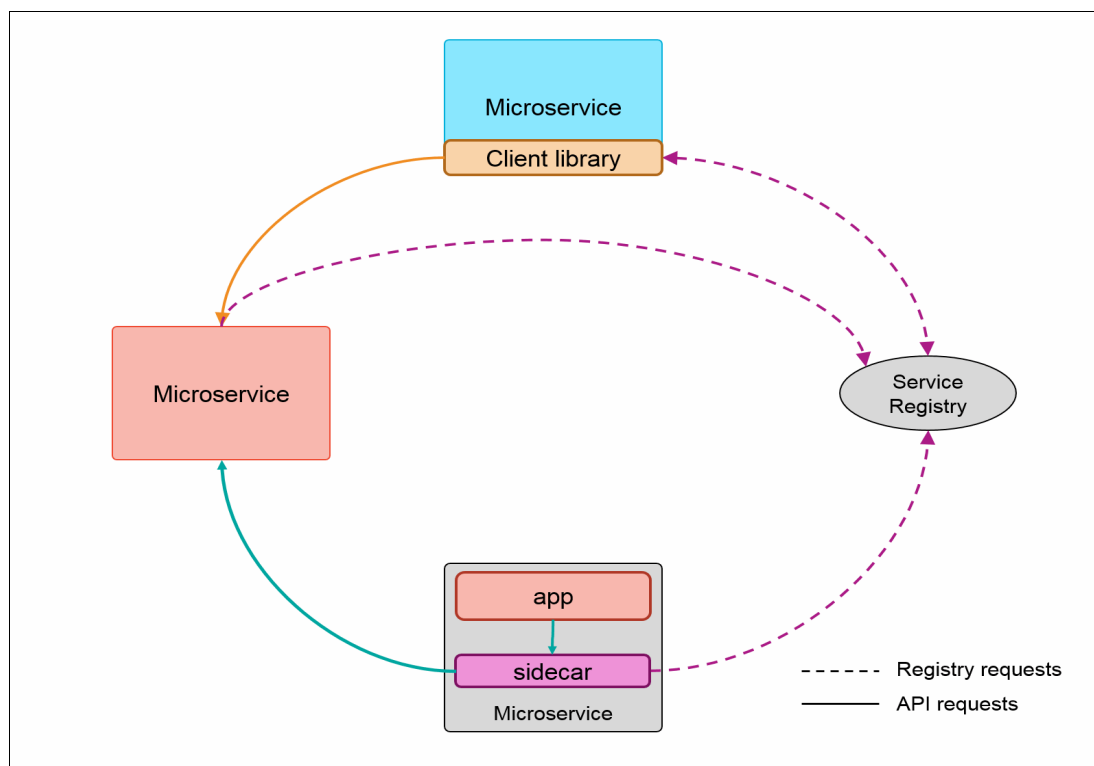


Figure 3-4 Requests are either made by the client library or by the sidecar process

Client library

Client libraries can be useful for many reasons, one of which is isolating the specifics of communicating with a remote resource. For service registries, client libraries like those provided by Consul or Netflix Eureka handle service registration and heart beating. Other libraries, such as Netflix Ribbon, provide client-side load balancing.

Generally start with an existing library, rather than writing your own. If all of the microservices in your system are written in Java, then you should standardize your library. If you have a polyglot system of microservices, then standardize on one for each language. By using the same library everywhere, you maintain the flexibility to move developers from microservice to microservice and avoid over-complicating your infrastructure and build process.

The disadvantage of using a client library is that you have now moved the complex service calls back into your application. This configuration makes it more complicated to test your service. Make sure to keep a good separation between the code that uses the client library and the business logic in your microservice. See Chapter 7, “Testing” on page 77 for more details about code separation to enable easier testing.

Sidecar

A good middle ground between the service proxy and a client library is a sidecar. A sidecar is a separate process that is deployed with your microservice, either as a separate process within a container, or as a separate, but closely related container.

Sidecars maintain the service registration with the service registry, and usually also perform client-side load balancing for outbound calls to other microservices.

Because sidecars run in their own process, they are independent of language. You can use the same sidecar implementation for all of the microservices in a polyglot application, regardless of the language that the service is written in.

Netflix Prana is an open source sidecar that provides the capabilities of the Java based Ribbon and Eureka client libraries to non-Java applications. For more information, see the following website:

<http://github.com/Netflix/Prana>

Kubernetes is an orchestration engine for containers that provides sidecar like service discovery and load balancing capabilities. For more information, see the following website:

<http://kubernetes.io/docs/user-guide/services>

Amalgam8 is a solution that can be used with and without containers. It manages service registration and provides client-side service discovery, routing, and load balancing. For more information, see the following website:

<http://amalgam8.io>

Choosing a solution that handles several of these elements can reduce the complexity of your services and your infrastructure.

3.3 API Gateway

An API Gateway can be used to perform service invocation for both internal and external clients. The API Gateway performs a similar role to a service proxy (see 3.2.1, “Server side” on page 27). Services make requests to the API Gateway that look up the service in the registry, makes the request, and then returns the response. There is a difference between the two, however. Requests to a service proxy use the exact API that the end service provides. Requests to an API Gateway use the API that the gateway provides. This configuration means that the API Gateway can provide different APIs to the ones that the microservice provides.

The internal microservices can use the exact APIs, while API Gateways provide most benefit to external clients. Typically microservices provide fine grained APIs. By contrast, an external client that uses the application probably does not need an API that is as fine grained. It might also need an API that uses information from several microservices. The API Gateway exposes APIs to the external client that are helpful to them and hides the actual implementation of the microservices behind the Gateway. All external clients access the application through this API Gateway.



Microservice communication

In a distributed system, inter-service communication is vital. The microservices that make up your application must work together seamlessly to provide value to your clients.

Chapter 3, “Locating services” on page 25 discussed how you can locate a particular service and the options for load balancing across different instances.

This chapter covers the communication between different services in a microservice architecture after the location of the required service is determined. This description includes synchronous and asynchronous communication, and how to achieve resilience.

The following topics are covered:

- ▶ Synchronous and asynchronous
- ▶ Fault tolerance

4.1 Synchronous and asynchronous

Synchronous communication is a request that requires a response, whether that be immediately or after some amount of time. Asynchronous communication is a message where no response is required.

A strong case can be made for using asynchronous events or messaging in highly distributed systems that are built from independent parts. There are situations where using synchronous calls is more appropriate, which are described in 4.1.1, “Synchronous messaging (REST)”.

For either invocation style, services should document any published APIs using tools like Swagger. Event or message payloads should also be documented to ensure that the system is comprehensible, and enable future consumers. Event subscribers and API consumers should tolerate unrecognized fields, as they might be new. Services should also expect and handle bad data. Assume that everything will fail at some point.

4.1.1 Synchronous messaging (REST)

As stated previously, in distributed systems, asynchronous forms of messaging are highly valuable. Where clear request/response semantics apply, or where one service needs to trigger a specific behavior in another service, synchronous APIs should be used instead.

These are usually RESTful operations that pass JSON formatted data, but other protocols and data formats are possible. In a Java based microservice, having the applications pass JSON data is the best choice. There are libraries to parse JSON to Java objects, and JSON is becoming widely adopted, making it a good choice to make your microservice easily consumable.

Async support for JAX-RS

Although JAX-RS requests will always be synchronous (you always require a response), you can take advantage of asynchronous programming models. The asynchronous support in JAX-RS 2.0 allows threads to perform other requests while waiting for the HTTP response. In a stateless EJB bean, an `AsyncResponse`¹ object instance is injected and bound to the processing of the active request by using the `@Suspended`² annotation. The `@Asynchronous`³ annotation allows work to be offloaded from the active thread. Example 4-1 shows the code that is used to return the response.

Example 4-1 JAX-RS 2.0 includes asynchronous support

```
@Stateless
@Path("/")
public class AccountEJBResource {
    @GET
    @Asynchronous
    @Produces(MediaType.APPLICATION_JSON)
    public void getAccounts(@Suspended final AsyncResponse ar) {
        Collection<Account> accounts = accountService.getAccounts();
        Response response = Response.ok(accounts).build();
        ar.resume(response);
    }
}
```

¹ <http://docs.oracle.com/javaee/7/api/javax/ws/rs/container/AsyncResponse.html>

² <http://docs.oracle.com/javaee/7/api/javax/ws/rs/container/Suspended.html>

³ <http://docs.oracle.com/javaee/6/api/javax/ejb/Asynchronous.html>

To see more examples of using the asynchronous support in JAX-RS 2.0, see the following GitHub project:

<https://github.com/WASdev/sample.async.jaxrs>

Another option is using reactive libraries like RxJava¹. RxJava is a Java implementation of ReactiveX², a library that is designed to aggregate and filter responses from multiple parallel outbound requests. It extends the observer pattern³ and also integrates well with libraries that are designed to handle load balancing and add resilience to your endpoints such as FailSafe⁴ and Netflix Hystrix⁵.

4.1.2 Asynchronous messaging (events)

Asynchronous messaging is used for decoupled coordination. An asynchronous event can only be used if the creator of the event does not need a response.

A request from an external client generally must go through several microservices before returning a response. If each of these calls is made synchronously, then the total time for the call holds up other requests. The more complex the microservice system and the more interactions between microservices required per external request, the bigger the resultant latency in your system. If the requests made in the process can be replaced with asynchronous events, then you should implement the event instead.

Use a reactive style for events. A service should only publish events about its own state or activity. Other services can then subscribe to those events and react accordingly.

Events are an excellent way to compose new interaction patterns without introducing dependencies. They enable the extension of the architecture in ways you had not considered when it was first created.

The microservices pattern requires each microservice to own its own data. This requirement means when a request comes in, there might be several services that need to update their databases. Services should advertise data changes by using events that other interested parties can subscribe to. To learn more about using events for data transactions and consistency, see Chapter 5, “Handling data” on page 41.

To coordinate the messaging of your application, you can use any one of the available message brokers and matching client libraries. Some examples that integrate with Java are the AMQP broker⁶ with the RabbitMQ Java client⁷, Apache Kafka⁸, and MQTT⁹, which is aimed at the Internet of Things space.

Internal messaging

Events can be used within a single microservice. For example, an order request could be processed by creating a set of events that the service connector classes subscribe to which triggers them to publish the external events. Using internal events can increase the speed of your response to synchronous requests because you do not have to wait for other synchronous requests to complete before returning a response.

¹ <https://github.com/ReactiveX/RxJava>

² <http://reactivex.io/>

³ https://en.wikipedia.org/wiki/Observer_pattern

⁴ <https://github.com/jhalterman/failsafe>

⁵ <https://github.com/Netflix/Hystrix>

⁶ <https://www.amqp.org/>

⁷ <https://www.rabbitmq.com/>

⁸ <http://kafka.apache.org/>

⁹ <http://mqtt.org/>

Take advantage of the Java EE specification support for events. Internal asynchronous events can be done by using Contexts and Dependency Injection (CDI)¹. In this model, an event consists of a Java event object and a set of qualifier types. Any methods wanting to react to events fired use the `@Observes` annotation. Events are fired by using the `fire()` method. Example 4-2 shows the methods and annotations in use.

Example 4-2 Events using Context and Dependency Injection

```
public class PaymentEvent {
    public String paymentType;
    public long value;
    public String dateTime;
    public PaymentEvent() {}
}

public class PaymentEventHandler {
    public void credit(@Observes @Credit PaymentEvent event) {
        ...
    }
}

public class Payment {
    @Inject
    @Credit
    Event<PaymentEvent> creditEvent;
    Public String pay() {
        PaymentEvent paymentPayload = new PaymentEvent();
        // populate payload...
        creditEvent.fire(paymentPayload);
        ...
    }
}
```

4.1.3 Examples

To illustrate clearly the scenarios where you should use synchronous and asynchronous communication this section covers two examples: Game On! and an online retail store. See 1.5, “Examples” on page 7 for an introduction to the examples.

Game On!

Consider a user who has authenticated with Game On! and wants to update a room. Figure 4-1 on page 37 shows the communication flow with the requests numbered for reference. When the user sends an update room request to the map service, they expect a confirmation response, which means the request must be synchronous. This synchronous request (1) returns a confirmation that either the update was successful or that the update has been successfully added to a queue (depending on the implementation of the update function).

After the update has occurred, that is the room has been updated in the database using a synchronous request (2), the map service needs to notify the other services about the change. This update does not require a response, so it is sent by using an asynchronous event (3) that other services are subscribed to (4).

¹ <http://docs.oracle.com/javaee/6/tutorial/doc/gkhic.html>

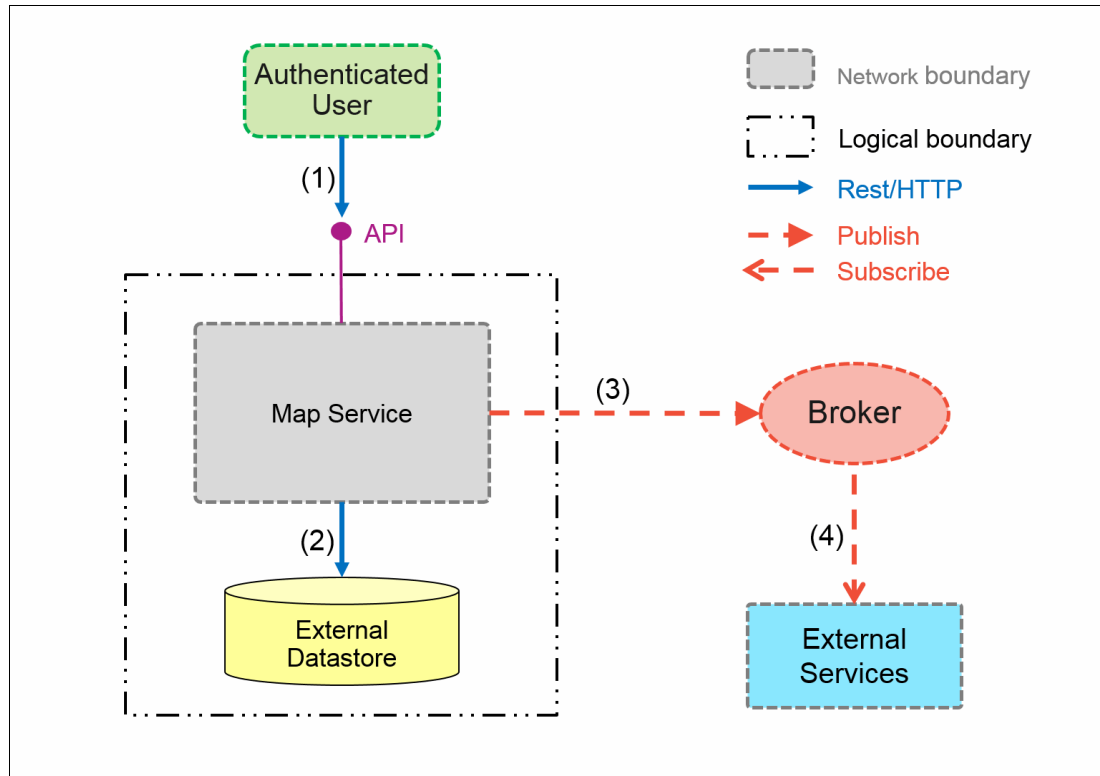


Figure 4-1 Communication pattern used to update a room

Online retail store

Consider a user wanting to place an order on an online retail store. After the user has entered all the details of the order, they click the **Submit** button. The user expects a response to confirm that the order has gone through, so the **Submit** button triggers a synchronous request that returns a response. After the order has been submitted, the application starts processing the order. The service that received the order request sends out asynchronous events that various other services are subscribed to. These events include inventory updates, shipping events, and payment events. Some services trigger other synchronous or asynchronous requests in response to the events. For example, the inventory service uses a synchronous request to update the inventory database.

4.2 Fault tolerance

A strong motivator for moving to a microservice architecture is to get a more fault tolerant and resilient application. Modern applications are required to have near zero downtime and respond to requests within seconds rather than minutes. Each individual service in a microservice must be able to continue functioning even if the other services have gone down. This section focuses on the techniques and tools you should use when communicating with other microservices to make your microservice fault tolerant and resilient.

4.2.1 Resilient against change

When a microservice makes a synchronous request to another microservice, it uses a specific API. The API has defined input attributes that must be included in the request and output attributes that are included in the response. In a microservice environment, this is

usually in the form of JSON data that is passed from service to service. It is unrealistic to assume that these input and output attributes will never change. Even in the best designed application, requirements are constantly changing, causing attributes to be added, removed, and changed. To be resilient against these forms of changes, the producer of a microservice must carefully consider the APIs they produce, and the APIs they consume. For more best practices concerning defining APIs including versioning, see Chapter 2, “Creating Microservices in Java” on page 9.

Consuming APIs

As the consumer of an API, it is necessary to do validation on the response you receive to check that it contains the information needed for the function being performed. If you are receiving JSON, you also need to parse the JSON data before performing any Java transformations. When doing validation or JSON parsing, you must do these two things:

- ▶ Only validate the request against the variables or attributes that you need
Do not validate against variables just because they are provided. If you are not using them as part of your request, do not rely on them being there.
- ▶ Accept unknown attributes
Do not issue an exception if you receive an unexpected variable. If the response contains the information you need, it does not matter what else is provided alongside.

Select a JSON parsing tool that allows you to configure the way it parses the incoming data. The Jackson Project¹ provides annotations to configure the parser as shown in Example 4-3. The `@JsonInclude` and `@JsonIgnoreProperties` annotations provided by the Jackson library are used to indicate which values should be used during serialization and whether to ignore unknown properties.

Example 4-3 Jackson annotations for JSON parsing

```
@JsonInclude(Include.NON_EMPTY)
@JsonIgnoreProperties(ignoreUnknown = true)
Public class Account {
    // Account ID
    @JsonProperty("id")
    protected String id;
}
```

By following both these rules, your microservice can be resilient against any changes that do not directly affect it. Even if the producer of the microservice removes or adds attributes, if they are not the ones you are using, your microservice continues functioning normally.

Producing APIs

When providing an API to external clients, do these two things when accepting requests and returning responses:

- ▶ Accept unknown attributes as part of the request
If a service has called your API with unnecessary attributes, discard those values. Returning an error in this scenario just causes unnecessary failures.
- ▶ Only return attributes that are relevant for the API being invoked
Leave as much room as possible for the implementation of a service to change over time. Avoid leaking implementation details by oversharing.

¹ <https://github.com/FasterXML/jackson>

As mentioned earlier, these rules make up the Robustness Principle. By following these two rules you put yourself in the best possible position for changing your API in the future and adapting to changing requirements from consumers. For more information about the Robustness Principle, see the following website:

https://en.wikipedia.org/wiki/Robustness_principle

4.2.2 Timeouts

When making a request to another microservice, whether asynchronously or not, the request must have a timeout. Because we expect services to come and go, we should not be waiting indefinitely for a response.

The asynchronous support in JAX-RS 2.0 has a `setTimeout` function on the `AsyncResponse` object (see “Synchronous messaging (REST)” on page 34). Tools like Failsafe and Netflix Hystrix have built-in support for timeouts.

Setting a timeout improves the resilience of your microservice, but can still result in a poor user experience because the end-to-end path will be slow. It is also possible for timeouts to be badly aligned so that the request from service A to service B times out while service B is waiting for a response from service C.

4.2.3 Circuit breakers

Using a timeout prevents a request from waiting indefinitely for a response. However, if a particular request always produces a timeout, you waste a lot of time waiting for the timeout to occur.

A circuit breaker is designed to avoid repeating timeouts. It works in a similar way to the electronic version of the same name. The circuit breaker makes a note every time a specific request fails or produces a timeout. If this count reaches a certain level, it prevents further calls from occurring, instead of returning an error instantly. It also incorporates a mechanism for retrying the call, either after a certain amount of time or in reaction to an event.

Circuit breakers are vital when calling external services. Generally, use one of the existing libraries for circuit breakers in Java.

4.2.4 Bulkheads

In shipping, a bulkhead is a partition that prevents a leak in one compartment from sinking the entire ship. Bulkheads in microservices are a similar concept. You need to make sure that a failure in one part of your application doesn't take down the whole thing. The bulkhead pattern is about how you create microservices rather than the use of any specific tool or library. When creating microservices, always ask yourself how you can isolate the different parts and prevent cascading failures.

The simplest implementation of the bulkhead pattern is to provide fallbacks. Adding a fallback allows the application to continue functioning when non-vital services are down. For example, in an online retail store there might be a service that provides recommendations for the user. If the recommendation service goes down, the user should still be able to search for items and place orders. An interesting example is chained fallbacks, where a failed request for personalized content falls back to a request for more general content, which would in turn fall back to returning cached (and possibly stale) content in preference to an error.

Another strategy for preventing a slow or constrained remote resource from bringing down the whole ship is to limit the resources available for those outbound requests. The most common ways of doing this use queues and Semaphore. For information about Semaphore, see the following website:

<https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Semaphore.html>

A queue has a depth that indicates a maximum amount of pending work. Any requests that are made after the queue is full get a fast failure. Queues are also assigned a finite number of workers, which defines the maximum number of server threads that can be blocked by the remote resource.

A semaphore mechanism works by having a set number of permits. A permit is required to make the remote request. After a request has completed successfully, the permit is released.

The significant difference between the two is in what happens when the allocated resources are all used up. With a semaphore approach, the outbound request is skipped entirely if the permit cannot be obtained, although the queue approach does offer some padding (at least until queue is full).

These approaches can also be used to deal with remote resources that impose rate limits, which are yet another form of unexpected failure to deal with.



Handling data

In an environment with polyglot persistence, which can exist in a system of microservices, it is necessary to keep the data handling manageable. To explain how this goal can be achieved, this chapter provides a short description of characteristics of microservices concerning data handling and then looks how it can be done with Java based microservices.

The following topics are covered:

- ▶ Data-specific characteristics of a microservice
- ▶ Support in Java

5.1 Data-specific characteristics of a microservice

One way to identify the data, which must be stored in the data store of your microservice, is a top down approach. Start at the business level to model your data. The following sections show how to identify this data, how to handle the data, and how it can be shared with other data stores of other microservices. For a description of the top down approach, see the following website:

http://en.wikipedia.org/wiki/Top-down_and_bottom-up_design

5.1.1 Domain-driven design leads to entities

From the approach in domain-driven design, you get the following objects, among others:

- ▶ Entity
“An object that is not defined by its attributes, but rather by a thread of continuity and its identity.”
- ▶ Value Objects
“An object that contains attributes but has no conceptual identity. They should be treated as immutable.”
- ▶ Aggregate
“A collection of objects that are bound together by a root entity, otherwise known as an aggregate root. The aggregate root guarantees the consistency of changes being made within the aggregate by forbidding external objects from holding references to its members.”
- ▶ Repository
“Methods for retrieving domain objects should delegate to a specialized Repository object such that alternative storage implementations may be easily interchanged.”

These quotations are taken from the following source:

http://en.wikipedia.org/wiki/Domain-driven_design

These objects should be mapped to your persistence storage (you should only aggregate entities when they have the same lifecycle). This domain model is the basis for both a logical data model and a physical data model.

For more information about domain-drive design terms, see the following website:

http://ddcommunity.org/resources/ddd_terms/

5.1.2 Separate data store per microservice

Every microservice should have its own data store (Figure 5-1) and is decoupled from the other microservice and its data. The second microservice must not directly access the data store of the first microservice.

Reasons for this characteristic:

- ▶ If two microservices share a data store, they are tightly coupled. Changing the structure of the data store (such as tables) for one microservice can cause problems for the other microservice. If microservices are coupled this way, then deployment of new versions must be coordinated, which must be avoided.
- ▶ Every microservice should use the type of database that best fits its needs (polyglot persistence, see 5.1.3, “Polyglot persistence” on page 45 for more details). You do not need to make a trade off between different microservices when choosing a database system. This configuration leads to a separate data store for each microservice.
- ▶ From a performance aspect, it can be useful for every microservice to have its own data store because scaling can become easier. The data store can be hosted on its own server.

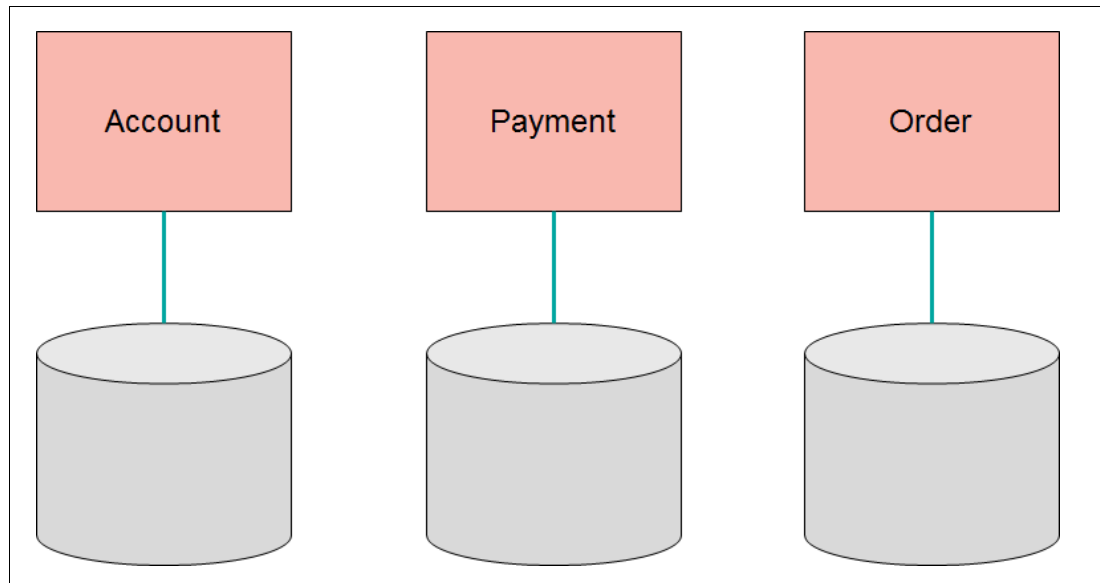


Figure 5-1 Separate data store per microservice

The separation of data stores for a relational database can be achieved in one of the following ways:

- ▶ Schema per microservice

Each service has its own schema in the database (Figure 5-2 on page 44). Other services can use the same database but must use a different schema. This configuration can be enforced with the database access control mechanism (using grants for connected database users) because developers suffering from time pressures tend to use shortcuts and can access the other schema directly.

- ▶ Database per microservice

Each microservice can have its own database but share database server with the other microservices (Figure 5-2 on page 44). With a different database, users can connect to the database server and there is a good separation of the databases.

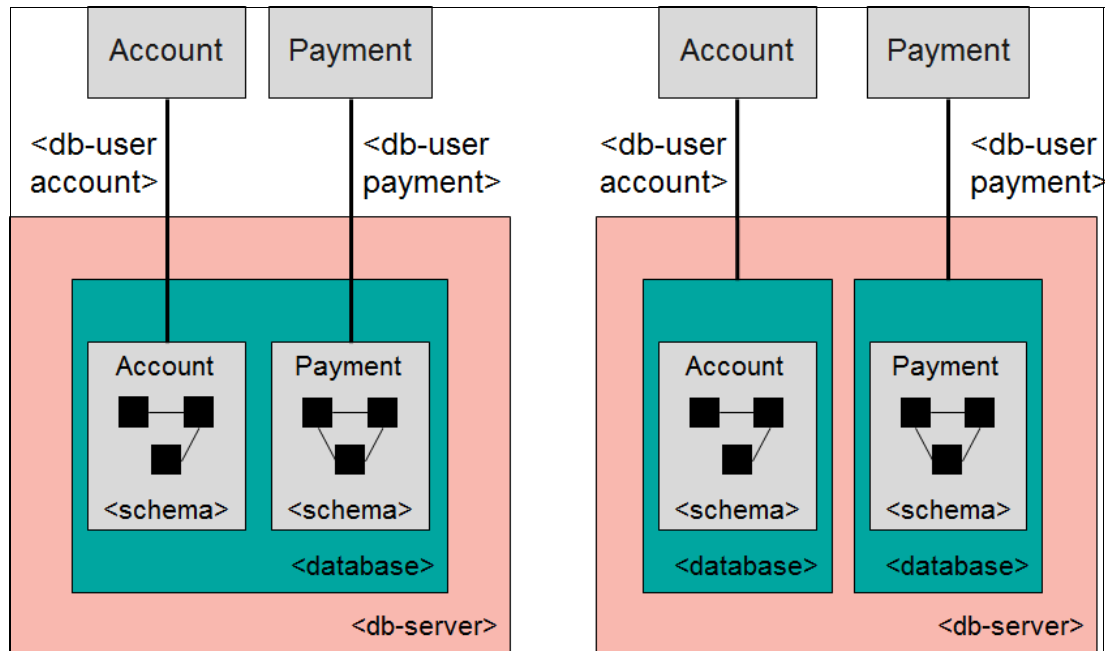


Figure 5-2 Schema per microservice and database per microservice

► Database server per microservice

This is the highest degree of separation. It can be useful, for example, when performance aspects need to be addressed (Figure 5-3).

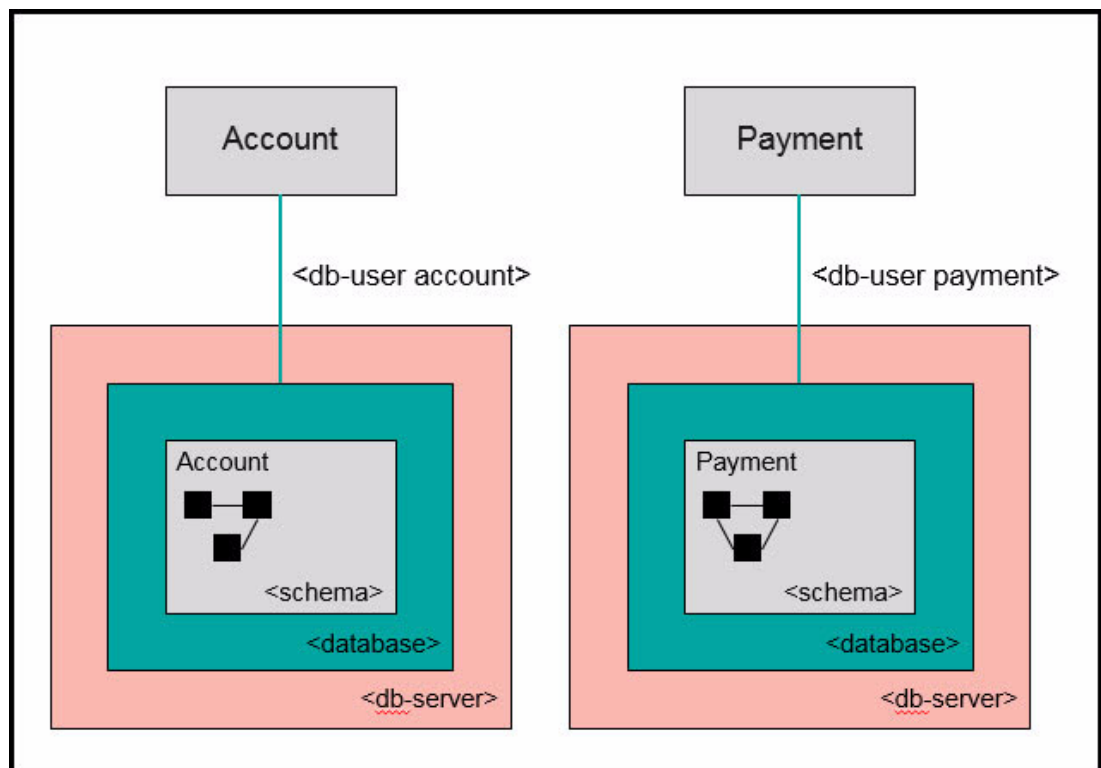


Figure 5-3 Database server per microservice

5.1.3 Polyglot persistence

Every microservice should use its own data store, which means that it can also use a different data storage technology. The NoSQL movement has led to many new data storage technologies, which can be used alongside a traditional relational database. Based on the requirements that an application must implement, it can choose between different types of technologies to store its data. Having a range of data storage technologies in one application is known as polyglot persistence.

For some microservices, it is best to store their data in a relational database. Other services, with a different type of data (such as unstructured, complex, and graph oriented) can store their data in some of the NoSQL databases.

Polyglot programming is a term that means that applications should be written in a programming language that best fits the challenge that an application must deal with.

A more in-depth description of these two terms can be found on the Martin Fowler website at:

<http://martinfowler.com/bliki/PolyglotPersistence.html>

5.1.4 Data sharing across microservices

In some situations, a client of a microservice application might want to request data that is owned by a different service. For example, the client might want to see all his Payments and the corresponding state of his payments. Then, he would have to query the service for his Account and afterward the service for Payment. It is against the best practices for microservices to join the data by using the database. To handle this business case, you must implement an adapter service (Figure 5-4) that queries the Account service and the Payment service, and then returns the collected data to the client. The adapter service is also responsible for doing the necessary data transformations on the data it has received.

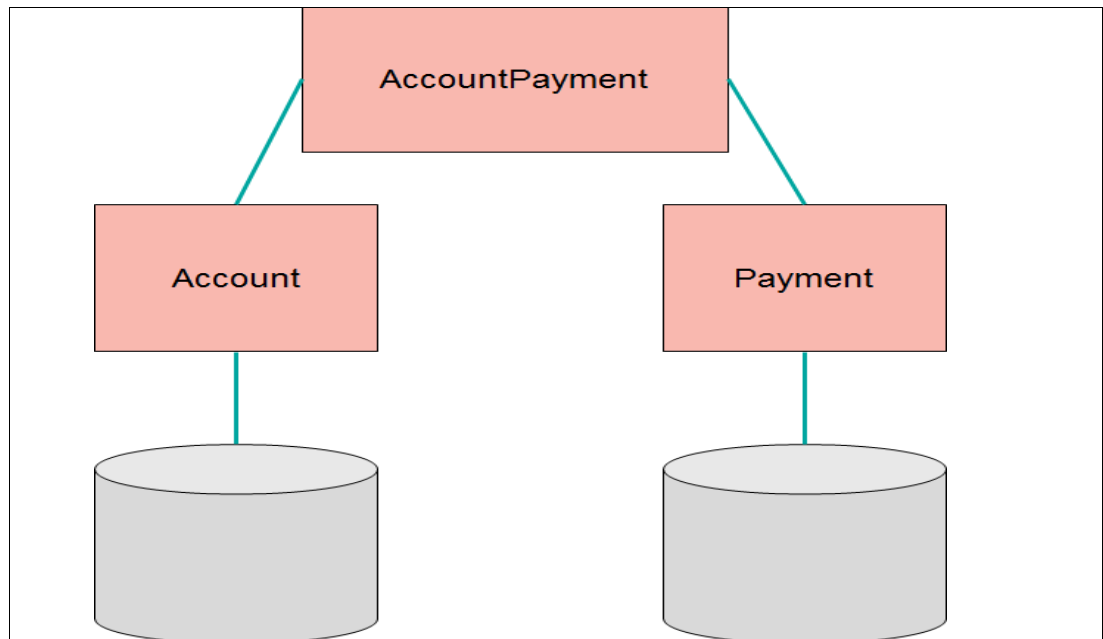


Figure 5-4 Adapter microservice

Changing data can become more complicated. In a system of microservices, some business transactions can span multiple microservices. In a microservice application for a retail store, for example, there might be a service to place an Order and a service to do the Payment. Therefore, if a customer wants to buy something in your shop and pays for it, the business transaction spans the two microservices. Every microservice has its own data store so, with a business transaction spanning two or more microservices, two or more data stores are involved in this business transaction. This section describes how to implement these business transactions.

Event-Driven architecture

You need a method to ensure the consistency of the data involved in a business transaction that spans two or more microservices. One way would be a distributed transaction, but there are many reasons why this should not be done in a microservice application. The main reason is the tight coupling of the microservices that are involved in a distributed transaction. If two microservices are involved in a distributed transaction and one service fails or has a performance problem, the other service must wait for the time out to roll back the transaction.

The best way to span a business transaction across microservices is to use an event-driven architecture. To change data, the first service updates its data and, in the same (inner) transaction, it publishes an event. The second microservice, which has subscribed to this event, receives this event and does the data change on its data. Using a publish/subscribe communication model, the two microservices are loosely coupled. The coupling only exists on the messages that they exchange. This technique enables a system of microservices to maintain data consistency across all microservices without using a distributed transaction.

If there are microservices in the microservice application that send many messages to each other, then they might be good candidates for merging into one service. But be careful because this configuration might break the domain-driven design aspect of your microservices. Adapter services doing complex updates, which span more than one service, can also be implemented by using events.

The programming model of an event-driven architecture is more complex, but Java can help to keep the complexity manageable.

Eventual Consistency

In an event-driven architecture, sending messages to other microservices creates a problem called Eventual Consistency. It is most often a runtime problem resulting from the following situation: microservice A changes data in its data store and sends, in the same inner transaction, a message to microservice B. After a short amount of time, microservice B receives the message and changes the data in its data store. In this normally short period, the data in the two data stores is not consistent. For example: service A updates the Order data in its data store and sends a message to service B to do the Payment. Until the Payment gets processed, there is an Order that has not been paid. Things get worse when the receiver of the message is not able to process the message. In this case, the messaging system or the receiving microservice must implement strategies to cope with this problem¹.

In a microservice application, every microservice has its own database. A business transaction that spans more than one microservice introduces eventual consistency because distributed transactions are discouraged to solve this problem. One way to cope with such business transactions is shown in Figure 5-5 on page 47. The Order microservice saves the Order in its data store and sends an event, for example `OrderCreated`, to the Payment microservice. While the Order microservice has not received the confirmation of the Payment from the Payment microservice, the Order is in a pending state.

¹ "Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions", Gregor Hohpe and Booby Woolf, Addison-Wesley Professional

The Payment service is subscribed to the OrderCreated event so it processes this event and does the Payment in its data store. If the Payment succeeds, then it publishes a PaymentApproved event that is subscribed by the Order microservice. After processing the PaymentApproved event, the state of the Order gets changed from Pending to Approved. If the customer queries the state of his Order he gets one of the following two responses: Order is in a pending state or Order is approved.

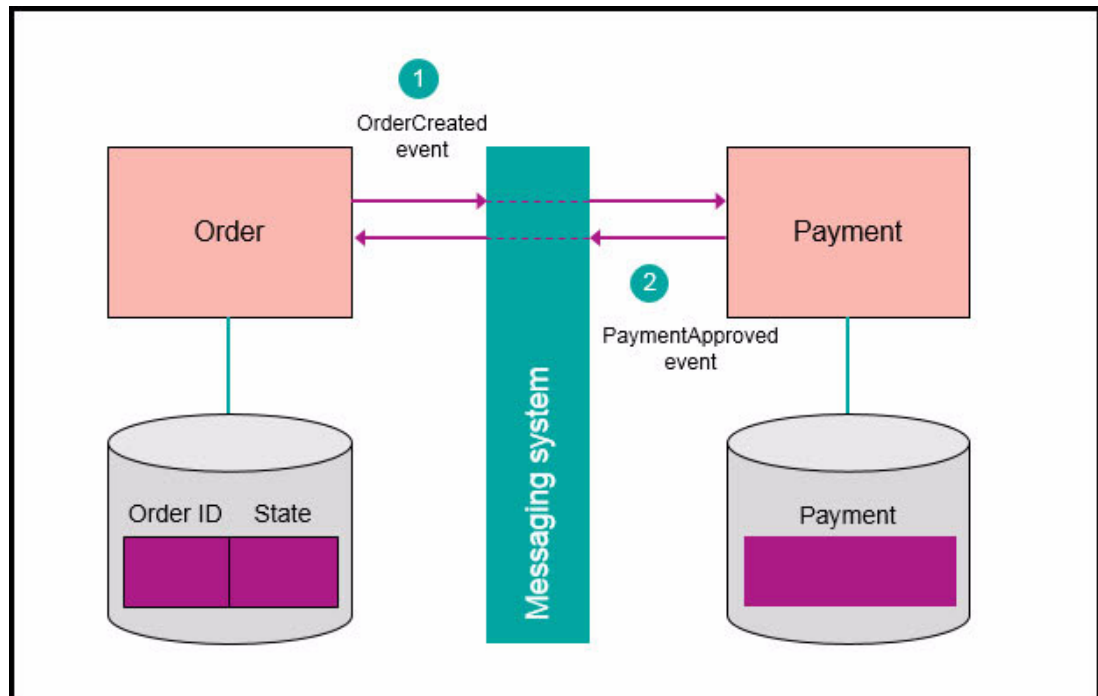


Figure 5-5 Event messaging between microservices

In situations where the data is not available, the service can send out to the client something like the following message: “Sorry, please try again later”.

Data replication

The separation of the data stores and the requirement to get data from different data stores can lead to the idea that using the data replication mechanisms of the database system would solve the problem. Doing this, for example, with a database trigger or a timed stored procedure or other processes, has the same disadvantage as sharing the database with more microservices. Changing the structure of the data on one side of the replication pipe leads to problems with the replication process. The process must be adapted when a new version of a service is being deployed. This is also a form of tight coupling and must be avoided.

As stated, the event-based processing can decouple the two data stores. The services processing the events can do the relevant data transformation, if needed, and store the data in their own data stores.

5.1.5 Event Sourcing and Command Query Responsibility Segregation

In an event-driven architecture, consider Command Query Responsibility Segregation (CQRS) and Event Sourcing. These two architectural patterns can be combined to handle the events flowing through your microservice application.

CQRS splits the access to the data store into two separate parts: One part for the read operations and the other part for the write operations. Read operations do not change the state of a system. They only return the state. Write operations (commands) change the state of the system, but do not return values. Event Sourcing stores sequences of these events that occur as changes happened to your data. Figure 5-6 shows a sample of using CQRS.

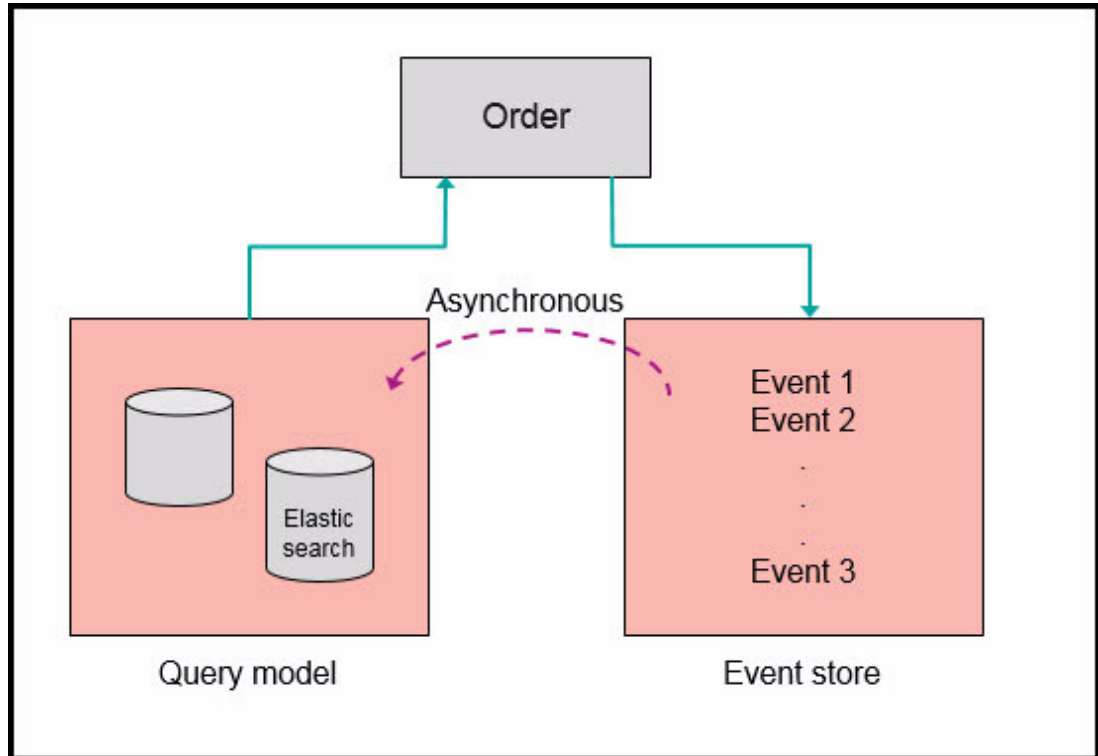


Figure 5-6 Sample of using CQRS

As shown in Figure 5-6, the events are stored sequentially in the event store. The data in the query model gets synchronized with the data from the event store. To support the event store or the query model, you can use specialized systems, for example Elastic Search to support the queries of your microservice. For more information about Elastic Search, see:

<https://www.elastic.co/>

This architecture can also be used to deal with events in your microservice application.

5.1.6 Messaging systems

You can use messaging systems or a message-oriented middleware to support your event-driven architecture.

“Message-oriented middleware (MOM) is software or hardware infrastructure supporting sending and receiving messages between distributed systems. MOM allows application modules to be distributed over heterogeneous platforms and reduces the complexity of developing applications that span multiple operating systems and network protocols. The middleware creates a distributed communications layer that insulates the application developer from the details of the various operating systems and network interfaces. APIs that extend across diverse platforms and networks are typically provided by MOM. MOM provides software elements that reside in all communicating components of a client/server architecture and typically support asynchronous calls between the client and server applications. MOM

reduces the involvement of application developers with the complexity of the master-slave nature of the client/server mechanism.”²

To communicate with message oriented middleware, you can use different protocols. The following are the most common protocols:

- ▶ Advanced Message Queuing Protocol (AMQP)³

AMQP “mandates the behavior of the messaging provider and client to the extent that implementations from different vendors are interoperable, in the same way as SMTP, HTTP, FTP, etc. have created interoperable systems.”⁴

- ▶ MQ Telemetry Transport (MQTT)⁵

MQTT is “publish-subscribe-based “lightweight” messaging protocol for use on top of the TCP/IP protocol. It is designed for connections with remote locations where a “small code footprint” is required or the network bandwidth is limited.”⁶ Mostly it is used in Internet of Things (IoT) environments.

In the Java world there exists an API to communicate with message-oriented middleware: Java Message Service (JMS) which is part of the Java EE Specification. The specific version is JMS 2.0⁷.

Due to the long existence of JMS (since 2001), many JMS Message Brokers are available that can be used as MOM systems⁸. But there also exist messaging systems that implement AMQP:

- RabbitMQ
- Apache Qpid⁹
- Red Hat Enterprise MRG

All of these systems provide Java APIs so they can be used in your Java based microservice.

5.1.7 Distributed transactions

Most, if not all, of the messaging systems support transactions. It is also possible to use distributed transactions when sending messages to the message system and changing data in a transactional data store.

Distributed transactions and two-phase commit can be used between a microservice and its backing store, but should not be used between microservices. Given the independent nature of microservices, there must not be an affinity between specific service instances, which is required for two-phase commit transactions.

For interactions spanning services, compensation or reconciliation logic must be added to ensure consistency is maintained.

² http://en.wikipedia.org/wiki/Message-oriented_middleware

³ <http://www.amqp.org/>

⁴ http://en.wikipedia.org/wiki/Advanced_Message_Queueing_Protocol

⁵ <http://mqtt.org/>

⁶ <http://en.wikipedia.org/wiki/MQTT>

⁷ <http://jcp.org/en/jsr/detail?id=343>

⁸ http://en.wikipedia.org/wiki/Java_Message_Service

⁹ <http://qpid.apache.org/index.html>

5.2 Support in Java

In a polyglot persistence environment, the programming language you use to implement your microservices must deal with the different persistence technologies. Your programming language must be able to support each of the different ways to persist your data. Java as a programming language has many APIs and frameworks that can support the developer to deal with different persistence technologies.

5.2.1 Java Persistence API

“The Java Persistence API (JPA) is a Java specification for accessing, persisting, and managing data between Java objects / classes and a relational database. JPA was defined as part of the EJB 3.0 specification as a replacement for the EJB 2 CMP Entity Beans specification. JPA is now considered the standard industry approach for Object to Relational Mapping (ORM) in the Java Industry.

JPA itself is just a specification, not a product; it cannot perform persistence or anything else by itself. JPA is just a set of interfaces, and requires an implementation. There are open-source and commercial JPA implementations to choose from and any Java EE 5 application server should provide support for its use. JPA also requires a database to persist to.”¹⁰

Java Enterprise Edition 7 (Java EE 7) has included Java Persistence 2.1 (JSR 338)¹¹.

The main focus for inventing JPA was to get an object-relational mapper to persist Java objects in a relational database. The implementation behind the API, the persistence provider, can be implemented by different open source projects or vendors. The other benefit that you get from using JPA is your persistence logic is more portable.

JPA has defined its own query language (Java Persistence Query Language (JPQL)) that can be used to generate queries for different database vendors. Java classes get mapped to tables in the database and it is also possible to use relationships between the classes to correspond to the relationships between the tables. Operations can be cascaded by using these relationships, so an operation on one class can result in operations on the data of the other class. JPA 2.0 has introduced the Criteria API that can help to get a correct query at both run time and compile time. All queries that you implement in your application can get a name and can be addressed with this name. This configuration makes it easier to know after a few weeks of programming what the query does.

The JPA persistence providers implement the database access. The following are the most common:

- ▶ Hibernate
<http://hibernate.org/>
- ▶ EclipseLink
<http://www.eclipse.org/eclipselink/>
- ▶ Apache OpenJPA
<http://openjpa.apache.org/>

The default JPA provider of Liberty for Java EE 7 is EclipseLink.

¹⁰ http://en.wikibooks.org/wiki/Java_Persistence/What_is_JPA%3F

¹¹ <http://jcp.org/en/jsr/detail?id=338>

JPA from a birds eye view

The following short explanations and code snippets show some of the features of JPA. The best way to start with JPA is to create the Entity-Classes to hold the data (Example 5-1).

Example 5-1 JPA Class to hold entity data

```
@Entity
public class Order {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String description;
    @Temporal(TemporalType.DATE)
    private Date orderDate;
    public String getDescription() {
        return description;
    }
    public void setDescription(String description) {
        this.description = description;
    }
    public Date getOrderDate() {
        return orderDate;
    }
    public void setOrderDate(Date orderDate) {
        this.orderDate = orderDate;
    }
    ...
}
```

Every entity class needs an `@Entity` annotation to be managed by the persistence provider. The entity class gets mapped by name to the corresponding table in the database (convention over configuration). Different mappings can also be applied. The attributes of the class get mapped by name to the columns of the underlying table. The automatic mapping of the attributes can also be overwritten (`@Column`). Every entity class must have an identity (see Domain-Driven Design in “Mapping domain elements into services” on page 13). The identity column (or columns), annotated with `@Id` are necessary for the persistence provider to map the values of the object to the data row in the table. The value of the identity column can be generated in different ways by the database or the persistence provider. Some of the attributes of an entity class must be transformed in a special way to be stored in the database. For example, a database column **DATE** must be mapped in the entity class with the annotation `@Temporal`.

The main JPA interface used to query the database is the `EntityManager`¹². It has methods to create, read, update, and delete data from the database. There are different ways to get a reference to the `EntityManager`, depending on the environment that the application is running. In an unmanaged environment (no servlet, EJB, or CDI container) you must use a factory method of the class `EntityManagerFactory` as shown in Example 5-2.

Example 5-2 How to get an EntityManager in a Java SE environment

```
EntityManagerFactory entityManagerFactory =
    Persistence.createEntityManagerFactory("OrderDB");
EntityManager entityManager =
    entityManagerFactory.createEntityManager();
```

¹² <http://docs.oracle.com/javaee/7/api/javax/persistence/EntityManager.html>

The String `OrderDB` is the name of the persistence unit that the `EntityManager` is created for. Persistence units are used to logically group entity classes and related properties to configure the persistence provider (configuration in the `persistence.xml` file).

In a managed environment, things are easier. The `EntityManager` can be injected from the container as shown in Example 5-3.

Example 5-3 How to get an EntityManager In a Java EE environment

```
@PersistenceContext
    EntityManager em;
```

If the persistence context gets injected without using a `unitName`, which is the name of the persistence unit configured in the configuration file (`persistence.xml`), then it uses a default value. If there is only one persistence unit, then this is the default that JPA uses.

The following sections show how to implement some simple create, retrieve, update, and delete operations with the methods from the `EntityManager` as shown in Example 5-4.

Example 5-4 JPA create operation

```
@PersistenceContext
    EntityManager em;
...
public Order createOrder(Order order) {
    em.persist(order);
    return order;
}
```

The `persist` method of the `EntityManager` does two things. First, the `EntityManager` manages the object, which means it holds the object in its persistence context. The persistence context can be seen as a cache holding the objects that are related to the rows in the database. The relation is done by using the database transactions. Second, the object gets persisted. It gets stored in the database. If the `Order` entity class has an ID attributed whose value gets generated by the database, then the attribute of the value will be set by the `EntityManager` after the insertion into the database. That is the reason why the return value of the `createOrder` method is the object itself (Example 5-5).

Example 5-5 JPA read operation by using the find method

```
@PersistenceContext
    EntityManager em;
...
public Order readOrder(Long orderID) {
    Order order = em.find(Order.class, orderID);
    return order;
}
```

The `EntityManager` method `find` searches the table for a row whose primary key is given as a parameter (`orderID`). The result is casted to a Java class of type `Order` (Example 5-6).

Example 5-6 JPA read operation by using JPQL

```
@PersistenceContext
    EntityManager em;
...
public Order readOrder(Long orderID) {
```

```

TypedQuery<Order> query =
    em.createQuery( "Select o from Order o " +
        "where o.id = :id", Order.class );
query.setParameter("id", orderID);
Order order = query.getSingleResult();
return order;
}

```

Example 5-6 on page 52 shows the function of the `find` method by using a JPQL with a parameter. Beginning from the JPQL string `Select o from Order o where o.id = :id`, a `TypedQuery` is generated. With a `TypedQuery`, you can omit the Java cast after generating the result object (see the parameter `Order.class`). The parameter in the JPQL gets addressed by a name (`id`) that makes it more readable for the developer. The method `getSingleResult` makes sure that only one row is found in the database. If there is more than one row corresponding to the SQL, then a `RuntimeException` is thrown.

The `merge` method does the update in the database (Example 5-7). The parameter `order` is a detached object, which means it is not in the persistence context. After the update in the database, an attached (it is now part of the persistence context) `order` object is returned by the `EntityManager`.

Example 5-7 JPA update operation

```

public Order updateOrder(Order order, String newDesc) {
    order.setDescription(newDesc);
    return em.merge(order);
}

```

To delete a row in the database, you need an attached object (Example 5-8). To attach the object to the persistence context, you can do a `find`. If the object is already attached, you do not need the `find` method. The method `remove` with the parameter of the attached object deletes the object in the database.

Example 5-8 JPA delete operation

```

public void removeOrder(Long orderId) {
    Order order = em.find(Order.class, orderId);
    em.remove(order);
}

```

As mentioned before, some configuration must be used to tell the persistence provider where to find the database and how to work with the database. This is done in a configuration file called `persistence.xml`. The configuration file needs to be in the class path of your service. Depending on your environment (Java containers or not), the configuration must be done in one of two ways (Example 5-9).

Example 5-9 Persistence.xml in a Java SE environment

```

<persistence>
    <persistence-unit name="OrderDB"
        transaction-type="RESOURCE_LOCAL">

        <class>com.service.Order</class>

        <properties>
            <!-- Properties to configure the persistence provider -->

```

```

    <property name="javax.persistence.jdbc.url"
      value="<jdbc-url-of-database" />
    <property name="javax.persistence.jdbc.user"
      value="user1" />
    <property name="javax.persistence.jdbc.password"
      value="password1" />
    <property name="javax.persistence.jdbc.driver"
      value="<package>.<DriverClass>" />
  </properties>
</persistence-unit>
</persistence>

```

To configure the persistence provider, the first thing that must be done is to define the persistence unit (OrderDB). One attribute of the persistence unit is the `transaction-type`. Two values can be set: `RESOURCE_LOCAL` and `JTA`. The first option makes the developer responsible for doing the transaction handling in his code. If there is no transaction manager available in your environment, then use this option. The second option is `JTA`, which stands for Java Transaction API and is part of the Java Community Process (JCP). This options tells the persistence provider to delegate transaction handling to a transaction manager that exists in your runtime environment.

Between the XML-tags `<class></class>`, you can enlist the entity classes to be used in this persistence unit.

In the properties section of the file, you can set values to configure the way the persistence provider will work with the database. Property names beginning with `javax.persistence.jdbc` are defined by the JPA standard. Example 5-9 on page 53 shows how to set the database URL (used for database connections) and the user and password. The `javax.persistence.jdbc.driver` property tells the persistence provider which JDBC-driver class to use.

The main difference in the configuration file for the Java EE environment is shown in Example 5-10.

Example 5-10 Persistence.xml in a Java EE environment

```

<persistence>
  <persistence-unit name="OrderDB">
    <jta-data-source>jdbc/OrderDB</jta-data-source>
    <class>com.widgets.Order</class>
    ...
  </persistence-unit>
</persistence>

```

The default transaction handling in JPA is to use `JTA` so that you do not need to set it in the configuration file. The `jta-data-source` property points to the JNDI-Name of the data source configured in the application server of your Java EE environment.

To do the transaction management in non Java EE environments, the `EntityManager` has some methods that can be used as shown in Example 5-11.

Example 5-11 Transaction management in no Java EE environments

```

EntityManagerFactory emf =
    Persistence.createEntityManagerFactory("OrderDB");
EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();

```

```

tx.begin();
try {
    em.persist(yourEntity);
    em.merge(anotherEntity);
    tx.commit();
} finally {
    if (tx.isActive()) {
        tx.rollback();
    }
}
}

```

Example 5-11 on page 54 shows the transaction handling in a non Java EE environment. Avoid doing transaction management on your own in a Java EE environment. There are better ways of doing this management as described in 5.2.2, “Enterprise JavaBeans” on page 57.

To separate the data store for your microservice, you can set the default schema of your relational database by using a configuration file as in Example 5-12.

Example 5-12 Setting default schema in my-orm.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings      xmlns="http://java.sun.com/xml/ns/persistence/orm"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm orm_2_0.xsd"
    version="2.0">
    <persistence-unit-metadata>
        <persistence-unit-defaults>
            <schema>ORDER</schema>
        </persistence-unit-defaults>
    </persistence-unit-metadata>
</entity-mappings>

```

This file must be referenced in the JPA configuration file persistence.xml (Example 5-13).

Example 5-13 Snippet showing reference in persistence.xml to a mapping file

```

<persistence-unit name="OrderDB">
    <mapping-file>custom-orm.xml</mapping-file>

```

This configuration sets the schema name to the name given in the my-orm.xml mapping file, in this case ORDER, to all your JPA classes. It ensures that you can only use tables in this schema.

JPA in combination with NoSQL databases

EclipseLink is one of the JPA providers that started to support NoSQL databases (from version 2.4 forward). From this version forward, they support MongoDB and Oracle NoSQL. Other NoSQL databases are expected to be supported in the future releases.

MongoDB is a NoSQL document-oriented database. The data structure is favorable to JSON like documents with the ability to have dynamic schemas. MongoDB has a specialized version of the JSON format called BSON:

<http://docs.mongodb.com/manual/reference/bson-types/>

The EclipseLink Solution Guide for EclipseLink Release 2.5 shows an example of how to access a MongoDB:

http://www.eclipse.org/eclipselink/documentation/2.5/solutions/nonrelational_db002.htm

Before deciding to use JPA (like with EclipseLink) as the provider for MongoDB, consider the following points:

- ▶ SQL is a specified language that has gone through a number of revisions. The database vendors have implemented this standard, but they have also added some features to the SQL that have not been standardized. JPA has good support for SQL, but does not support all of the features that are exposed by MongoDB. If some of these features are needed in your microservice, then the benefits you get from using JPA are fewer.
- ▶ JPA has numerous features that do not make sense in a document-oriented database, but the `EntityManager` has methods for these features. So you must define which methods to use in your service.

If you are familiar with JPA and only require simple functions for storing your data in a NoSQL database, then it is acceptable to start doing this with your JPA provider. If the data access gets more complicated, it is better to use the Java drivers from your NoSQL database. JPA does not really fit with NoSQL databases, but it can be a good starting point for your implementation. For a sample of how to use the native Java driver of MongoDB, see:

<http://docs.mongodb.com/getting-started/java/>

To get more out of your JPA provider, it can be useful to use Spring Data JPA. In addition to the JPA provider, Spring Data JPA adds an extra layer on top of the JPA provider:

<http://projects.spring.io/spring-data-jpa/>

Suitability of JPA for data handling in microservices

The following list shows some arguments why JPA is useful for data handling in your microservices:

- ▶ Defining your microservices from the perspective of Domain Driven Design leads to the situation where most of the microservices need only simple queries to persist their entities (simple create, retrieve, update, and delete operations). JPA's `EntityManager` has these create, retrieve, update, and delete methods that you need: `persist`, `find`, `merge`, `delete`). Not much programming is needed to call these methods.
- ▶ In some cases, the queries get more complex. These queries can be done with the query language defined in JPA: JPQL. The need for complex queries should be an exceptional case. JSON documents with a hierarchy of entity data should each be stored separately. This leads to simple IDs and simple queries.
- ▶ JPA is standardized and has a great community to support you in developing your microservice. All Java EE servers must support JPA.
- ▶ To implement microservices not running in a Java EE container, you can also use JPA.
- ▶ Generating entity classes from the database (reverse engineering) can reduce the number of lines of code you must implement by yourself.
- ▶ For polyglot persistence, JPA has support for a relational data store and a document-oriented data store (EclipseLink).
- ▶ JPA as an abstraction of the relational database allows you to exchange your relational data store with another relational data store if you need to. This portability prevents your microservice from a vendor lock in.

- ▶ To implement the strategy, every microservice should have its own schema in a relational database, you can set the default schema for your service in the JPA configuration file `persistence.xml`.

5.2.2 Enterprise JavaBeans

Enterprise JavaBeans 3.2 (EJB) (specified in the JSR 345¹³) is part of the Java EE specification. EJBs are more than normal Java classes for these reasons:

- ▶ They have a lifecycle.
- ▶ They are managed by an EJB container (runtime environment for the EJB).
- ▶ They have a lot more features that can be helpful.

EJBs are server-side software components. Since EJB 3.0, it is no longer necessary to use a Deployment Descriptor. All declarations of an EJB can be done with annotations in the EJB class itself. The implementation to handle the EJB inside an EJB container is as lightweight as it is for CDI managed beans. The thread handling of the EJBs is done by the EJB container (similar to thread handling in a servlet container). An additional feature of EJBs is that they can be used in combination with Java EE Security.

EJBs can be divided into the following types:

- ▶ Stateless
- ▶ Stateful
- ▶ Singleton
- ▶ Message-driven beans (MDB)

Stateless EJBs cannot hold any state, but stateful EJBs can. Due to the characteristics of a microservice, stateful EJBs should not be used in a microservice. A Singleton Bean exists only one time in a Java EE server. MDBs are used in the processing of asynchronous messages in combination with a JMS provider.

EJBs can implement several business views, which must be annotated accordingly:

- ▶ Local interface (`@Local`)
Methods in this Interface can only be called by clients in the same Java virtual machine (JVM).
- ▶ Remote interface (`@Remote`)
Methods that are listed in this interface can be called by clients from outside the JVM.
- ▶ No interface (`@LocalBean`)
Nearly the same as local interface, except all public methods of the EJB class are exposed to the client.

In a lightweight architecture, which a microservice should have, it is useful to implement the EJBs as no interface EJBs.

One of the main benefits that EJBs provide is automatic transaction handling. Every time the a business method is called, the transaction manager of the EJB container is invoked (exception: an EJB with explicitly switched off support for transactions). So it is easy to use EJBs in combination with a transactional data store. Integrating EJBs with JPA is also easy.

¹³ <http://jcp.org/en/jsr/detail?id=345>

The code snippet in Example 5-14 shows an example of how to combine the EJBs with the JPA framework.

Example 5-14 Stateless (no-interface) EJB with PersistenceContext

```
@Stateless
@LocalBean
public class OrderEJB {
    @PersistenceContext
    private EntityManager entityManager;
    public void addOrder(Order order) {
        entityManager.persist(order);
    }
    public void deleteOrder(Order order) {
        entityManager.remove(order);
    }
    . . .
}
```

The EntityManager gets injected as described in 5.2.1, “Java Persistence API” on page 50.

An EJB can have one of the following transaction attributes to operate with a transactional data store (must be implemented by the EJB container):

- ▶ REQUIRED (default)
- ▶ MANDATORY
- ▶ NEVER
- ▶ NOT_SUPPORTED
- ▶ REQUIRES_NEW
- ▶ SUPPORTS

For more information about these attributes, see the following website:

<http://docs.oracle.com/javaee/7/api/javax/ejb/TransactionAttributeType.html>

These so called container-managed transactions (CMTs) can be used on every business method of an EJB. Bean-managed transactions (BMT), which can also be used in an EJB, should be avoided. The annotation TransactionAttribute can be set on a class level so every business method of that class has this transaction attribute (Example 5-15). If nothing is set, then all methods have the default transaction level (REQUIRED). A transaction attribute at the method level overwrites the class attribute.

Example 5-15 Set transaction attribute explicit in an EJB

```
@TransactionAttribute(REQUIRED)
@Stateless
@LocalBean
public class OrderEJB {
    ...
    @TransactionAttribute(REQUIRES_NEW)
    public void methodA() {...}
    @TransactionAttribute(REQUIRED)
    public void methodB() {...}
}
```

REST Endpoint not implemented as EJB

Some of the database changes or verifications are done at the point where the transaction manager has decided to commit. In some situations, check constraints in the database for example, are verified as one of the last steps before the transaction is committed. If this verification fails, then the outcome is a `RuntimeException` that the JPA provider throws, because JPA uses runtime exceptions to report errors. If you use EJBs for transaction management, then the place to catch the `RuntimeException` is in the stub code of the EJB where the EJB container does the transaction management. The stub code is generated by the EJB container. Therefore, you cannot react to this `RuntimeException` and the exception is thrown further to the place it will be caught.

If you implement your REST endpoint as an EJB, as some people prefer, then the exception must be caught in your REST provider. REST providers do have exception mappers to do the conversion of exceptions to HTTP error codes. However, these exception mappers do not interfere in the case when the `RuntimeException` is thrown during the database commit. Therefore, the `RuntimeException` is received by the REST client, which should be avoided.

The best way to handle these problems is to implement the REST endpoint as a CDI managed request scoped bean. In this CDI bean, you can use injection the same way as inside an EJB. So it is easy to inject your EJB into the CDI managed bean (Example 5-16).

Example 5-16 REST endpoint implemented as CDI managed bean with EJB injected

```
@RequestScoped
@Path("/Order")
public class OrderREST {
    @EJB
    private OrderEJB orderEJB;
    ...
}
```

It is also possible to integrate EJBs with Spring (Enterprise JavaBeans (EJB) integration - Spring) and if preferred, the transaction management can be done using Transaction Management Spring. However, in a Java EE world it is better to delegate transaction management to the server.

For more information about Spring, see the following website:

<http://docs.spring.io/spring/docs/current/spring-framework-reference/html/ejb.html>

5.2.3 BeanValidation

BeanValidation is also part of the Java EE 7 specification: Bean Validation 1.1 JSR 349¹⁴. The intention of Bean Validation is to easily define and enforce validation constraints on your bean data. In a Java EE environment, Bean Validation is done by the different Java EE containers automatically. The developer only needs to set the constraints, in the form of annotations on attributes, methods, or classes. The validation is done on the invocation of these elements automatically (if configured). The validation can also be done explicitly in your source code.

For more information about Bean Validation, see the following website:

<http://beanvalidation.org/>

¹⁴ <http://jcp.org/en/jsr/detail?id=349>

Examples of built-in constraints in the `javax.validation.constraints` package are shown in Example 5-17.

Example 5-17 Default built-in constraints in Bean Validation

```
private String username; // username must not be null
@Pattern(regexp="\\(\\d{3}\\)\\d{3}-\\d{4}")
private String phoneNumber; // phoneNumber must match the regular expression
@Size(min=2, max=40)
String briefMessage; // briefMessage between 2 and 40 characters
```

Constraints can also be combined as shown in Example 5-18.

Example 5-18 Combination of constraints

```
@NotNull
@Size(min=1, max=16)
private String firstname;
```

It is also possible to extend the constraints (custom constraints). Example 5-19 shows how to do the validation on your own.

Example 5-19 Validating programmatically

```
Order order = new Order( null, "This is a description", null );
ValidatorFactory factory =
    Validation.buildDefaultValidatorFactory();
Validator validator = factory.getValidator();
Set<ConstraintViolation<Order>> constraintViolations = validator.validate(order);
assertEquals( 2, constraintViolations.size() );
assertEquals( "Id may not be null",
    constraintViolations.iterator().next().getMessage() );
assertEquals( "Order date may not be null",
    constraintViolations.iterator().next().getMessage() );
```

JPA can be configured to do the bean validation automatically. The JPA specification mandates that a persistence provider must validate so called managed classes (see Example 5-20). Managed classes in the sense of JPA are, for example, entity classes. All other classes used for the JPA programming must also be validated (for example, embedded classes, super classes). This process must be done in the lifecycle events these managed classes are involved in.

Example 5-20 Persistence.xml with bean validation turned on

```
<persistence>
  <persistence-unit name="OrderDB">
    <provider>
      org.eclipse.persistence.jpa.PersistenceProvider
    </provider>
    <class> . . . </class>
    <properties>
      <property name="javax.persistence.validation.mode"
        value="AUTO" />
    </properties>
  </persistence-unit>
</persistence>
```

All the other frameworks used from the Java EE stack can be used to validate the beans automatically (for example JAX-RS, CDI) and also EJB as shown in Example 5-21.

Example 5-21 Bean validation in an EJB

```
@Stateless
@LocalBean
public class OrderEJB {
    public String setDescription(@Max(80) String newDescription){
        . . .
    }
}
```

Architectural layering aspects of JPA and BeanValidation

Depending on the layers that are implemented in a microservice, there are some aspects to address.

In a service with just a few layers, it is easier to use the JPA entity classes as a Data Transfer Object (DTO) as well. After the JPA object gets detached from its persistence context, it can be used as a plain old Java object (POJO). This POJO can also be used as a DTO to transfer the data to the REST endpoint. Doing it this way has some disadvantages. The Bean Validation annotations get mixed with the JPA annotations that can lead to Java classes with numerous annotations and your REST endpoints become more closely related to your database.

If the microservice is a little bit larger or has to deal with a more complex data model, then it can be better to use a separate layer to access the database. This extra layer implements all the data access methods based on the JPA classes. The classes in this layer are data access objects (DAOs). You can use the DAO classes to generate the DTO classes for the REST endpoint and focus, on the one side, on the data model (DAO) and, on the other side, on the client (DTO). The disadvantage of this pattern is that you must convert your JPA classes handled in the DAO layer to DTOs and vice versa. To avoid creating a lot of boilerplate code to do this, you can use some frameworks to help you do the converting. The following frameworks are available to convert Java objects:

- ▶ **ModelMapper**
<http://modelmapper.org/>
- ▶ **MapStruct**
<http://mapstruct.org/>

To extend the possibilities you get from Bean Validation, it can be useful to get additional features by using Springs. For more information, see “Validation, Data Binding, and Type Conversion” at:

<http://docs.spring.io/spring/docs/current/spring-framework-reference/html/validation.html>

5.2.4 Contexts and Dependency Injection

If your microservice does not store its data in a transactional data store, then consider using CDI managed beans instead of EJBs. CDI 1.1 is specified in the JSR 346¹⁵ and is part of the Java EE 7 specification. CDI managed beans can be a good alternative in these situations.

¹⁵ <http://jcp.org/en/jsr/detail?id=346>

For more information about CDI managed beans, see the following website:

<http://docs.oracle.com/javaee/6/tutorial/doc/giwh1.html>

In comparison to EJBs, CDI has no Java EE security on its own and does not get the persistence context injected. This process must be done by the developer themselves or by using additional frameworks. Apache DeltaSpike, for example, has many modules that you can use to extend the functions of CDI. For more information about Apache DeltaSpike, see:

<http://deltaspike.apache.org/index.html>

Additional frameworks can be used to extend the functions of CDI managed beans. EJBs have a thread pool that can be managed in an application server. CDI has nothing that corresponds to this function yet. To be able to configure a thread pool can be helpful in an environment with high loads.

To implement a microservice not running in a Java EE application server, CDI and additional modules provide you many functions that are useful in these environments.

5.2.5 Java Message Service API

To implement an event-driven architecture in a Java world, JMS API provides support, which is also specified in Java Message Service 2.0 JSR 343¹⁶. JMS is used to communicate to a so-called message provider that must be implemented by the message-oriented middleware (MOM).

When JMS was updated from version 1.1 to version 2.0, which is part of the Java EE 7 specification, much rework was done to make the API easier to use. JMS 2.0 is compatible with an earlier version, so you can use your existing code or use the new simplified API for your new microservices. The old API will not be deprecated in the next version.

According to the smart endpoints and dumb pipes methodology¹⁷, your Java based microservice must just send your JSON message to an endpoint hosted by the JMS provider. The sender of the messages is called producer and the receiver of the message is the consumer. These endpoints can be of these types:

- Queue

Messages in a queue are consumed by only one consumer. The sequence of the messages in the queue might be consumed in a different order. Queues are used in a point to point semantic.

- Topic

These messages can be consumed by more than one consumer. This is the implementation for publish/subscribe semantics.

In a REST-based microservice, where JSON-based requests are sent by the client, it is a good idea to also use the JSON format for your messaging system. Other messaging applications use XML. If you have only one format in your system of microservice, it is easier to implement JSON.

Example 5-22 Producer sending messages to a JMS queue with EJB

```
@Stateless
@LocalBean
public class OrderEJB {
```

¹⁶ <http://jcp.org/en/jsr/detail?id=343>

¹⁷ <http://martinfowler.com/articles/microservices.html#SmartEndpointsAndDumbPipes>

```

@Inject
@JMSConnectionFactory("jms/myConnectionFactory")
    JMSContext jmsContext;
@Resource(mappedName = "jms/PaymentQueue")
    Queue queue;
public void sendMessage(String message) {

    jmsContext.createProducer().send(queue, message);
}

```

A JMSContext and a Queue are needed to send a message (Example 5-22 on page 62). These objects are injected if the message sender is running in a Java EE application server. Example 5-22 on page 62 uses an EJB so these resources are injected. The configuration for the injected objects must be done in the application server. If an exception occurs, a runtime exception JMSRuntimeException is thrown.

The scope of an injected JMSContext in a JTA transaction is transaction. So if your message producer is an EJB, your message is delivered in the context of a transaction, which avoids losing messages.

To consume the message from a queue, it is easy to use a message driven EJB (MDB) as shown in Example 5-23. Using an MDB also has the advantage that consuming messages is done in a transaction, so no message is lost.

Example 5-23 Consumer - process messages from a JMS queue

```

@MessageDriven(
    name="PaymentMDB",
    activationConfig = {
        @ActivationConfigProperty(
            propertyName="messagingType",
            propertyValue="javax.jms.MessageListener"),
        @ActivationConfigProperty(
            propertyName = "destinationType",
            propertyValue = "javax.jms.Queue"),
        @ActivationConfigProperty(
            propertyName = "destination",
            propertyValue = "PaymentQueue"),
        @ActivationConfigProperty(
            propertyName = "useJNDI",
            propertyValue = "true"),
    }
)
public class PaymentMDB implements MessageListener {
    @TransactionAttribute(
        value = TransactionAttributeType.REQUIRED)
    public void onMessage(Message message) {

        if (message instanceof TextMessage) {
            TextMessage textMessage = (TextMessage) message;
            String text = message.getText();
            . . .
        }
    }
}

```

You must use the `@MessageDriven` annotation to declare the EJB type as message-driven EJB. Within this annotation, you can set the name of the MDB and some activation configurations. The properties of the activation configuration tie the MDB to the JMS messaging system that handles the queues or topics. In an application server environment, where your MDB gets hosted, it is easy to configure these elements. The MDB itself implements a `MessageListener` interface that has only one method: `onMessage`. Every time the JMS provider has a message to process, it calls this method. The method is annotated with a transactional attribute to show that this method is called in a transaction. The default transaction attribute for an MDB is `TransactionAttributeType.REQUIRED`. Inside the method, the message object must be converted and the message can be extracted as a `String`. Other message types are also possible.

It is good practice to use the MDB only for the message handling. Keep the MDB as a technical class. The rest of your business code should be implemented in a Java POJO that gets called by the MDB. This configuration makes the business code easier to test in your JUnits.

As stated before, every MDB runs in a transaction, so no message gets lost. If an error occurs during the processing of the message and the EJB container, which handles the MDB, receives this runtime exception, then the message is redelivered to the MDB (error loop). The retry count, which can be configured in your JMS provider, specifies how often this occurs. After the maximum number of retries, the message is normally put into an error queue. Messages in an error queue must be handled separately.

If a business transaction spans more than one microservice, use an event-driven architecture (see section 5.1.4, “Data sharing across microservices” on page 45). This means that the microservice sending the event must perform the following tasks:

- ▶ Change data in its data store
- ▶ Send the message to the second microservice

The receiving microservice must perform these tasks:

- ▶ Receive the message from the queue
- ▶ Change data in its data store

To be consistent, these two things must be done in one transaction if the data store is transactional. This requirement is also true for the producer and the consumer side of the messaging system. In these cases, you must use a distributed transaction. The transaction partners are the data store and the JMS provider (not the two data stores of the two microservices).

To track the messages produced and consumed, it is useful to use a correlation ID. A correlation ID specified by the producer of the message correlates to the message consumed by the consumer. This correlation ID can also be used in the logging of the microservice to get the complete communication path of your microservices call (see 9.2, “Logging” on page 111).

Java provides a class to generate a unique Id: `UUID`. This class can be used to generate a correlation ID. Example 5-24 shows setting a correlation ID.

Example 5-24 Setting correlation ID in a JMS message

```
// JMSContext injected as before
JMSProducer producer = jmsContext.createProducer();
producer.setJMSCorrelationID(UUID.randomUUID().toString());
producer.send(queue, message);
```

Example 5-25 shows retrieving the correlation ID.

Example 5-25 Retrieving the correlation ID from a JMS message

```
// message received as before
String correlationId = message.getJMSCorrelationID();
```

For more information about UUID, see the following website:

<http://docs.oracle.com/javase/7/docs/api/java/util/UUID.html>

If you are using a non-JMS provider as the message-oriented middleware, JMS might not be the correct way to send messages to this system. Using RabbitMQ (which is an AMQP Broker) can be done with JMS because Pivotal has implemented a JMS Adapter for RabbitMQ. For more information about RabbitMQ, see the following website:

<http://www.rabbitmq.com/>

Apache Qpid also implements a JMS Client for the AMQP Protocol. These are just a few examples to show that it can also be useful to use JMS to communicate with non-JMS providers. But, depending on your requirements, it can be better to use the native Java driver of the messaging system. For more information about Apache Qpid, see the following website:

<http://qpid.apache.org/index.html>

Spring's support for JMS Message Providers is another alternative for handling JMS messages. For more information about Spring, see the following website:

<http://docs.spring.io/spring/docs/current/spring-framework-reference/html/jms.html>

5.2.6 Java and other messaging protocols

Depending on the features that you need from your message-oriented middleware, you might need a non-JMS message provider system. MQTT is a messaging protocol that best fits the needs of Internet of Things (IoT) and AMQP. It was developed to be portable across vendors. JMS cannot be used to communicate with these systems. Using their provided clients allows you to use all special features they provide. For more information about MQTT, see the following website:

<http://mqtt.org/>

Apache Kafka is a non-JMS provider, but provides a Java driver. There is an enhancement request to implement an adapter to let the clients talk JMS with Apache Kafka¹⁸ but this issue is still open. So it is better to use the Java driver that Kafka provides. For more information about Kafka, see the following website:

<http://kafka.apache.org/>

RabbitMQ is a message broker system which implements the AMQP protocol. It provides a Java client¹⁹. For more information about RabbitMQ, see the following website:

<https://www.rabbitmq.com/>

¹⁸ <http://issues.apache.org/jira/browse/KAFKA-1995>

¹⁹ <http://www.rabbitmq.com/java-client.html>

Spring has a library to talk to AMQP based messaging systems. Spring also provides support for the MQTT protocol²⁰. For more information about Spring, see the following website:

<http://projects.spring.io/spring-amqp/>

As you can see, there is support for Java with non JMS messaging systems.

²⁰ <http://docs.spring.io/spring-integration/reference/html/mqtt.html>



Application Security

As always, security is an important consideration. This chapter describes security requirements within the context of a cloud native microservices architecture. The following topics are covered:

- ▶ Securing microservice architectures
- ▶ Identity and Trust

6.1 Securing microservice architectures

The dynamic nature of microservice architectures changes how security should be approached. This consideration is specifically relevant to how application or service boundaries are defined. Figure 6-1 shows simplified representations of how requests flow in monolithic and microservice architectures. Connections between microservices and persistent storage services (RDBMS, NoSQL) have been hidden for general simplicity, but represent yet another dimension of network traffic.

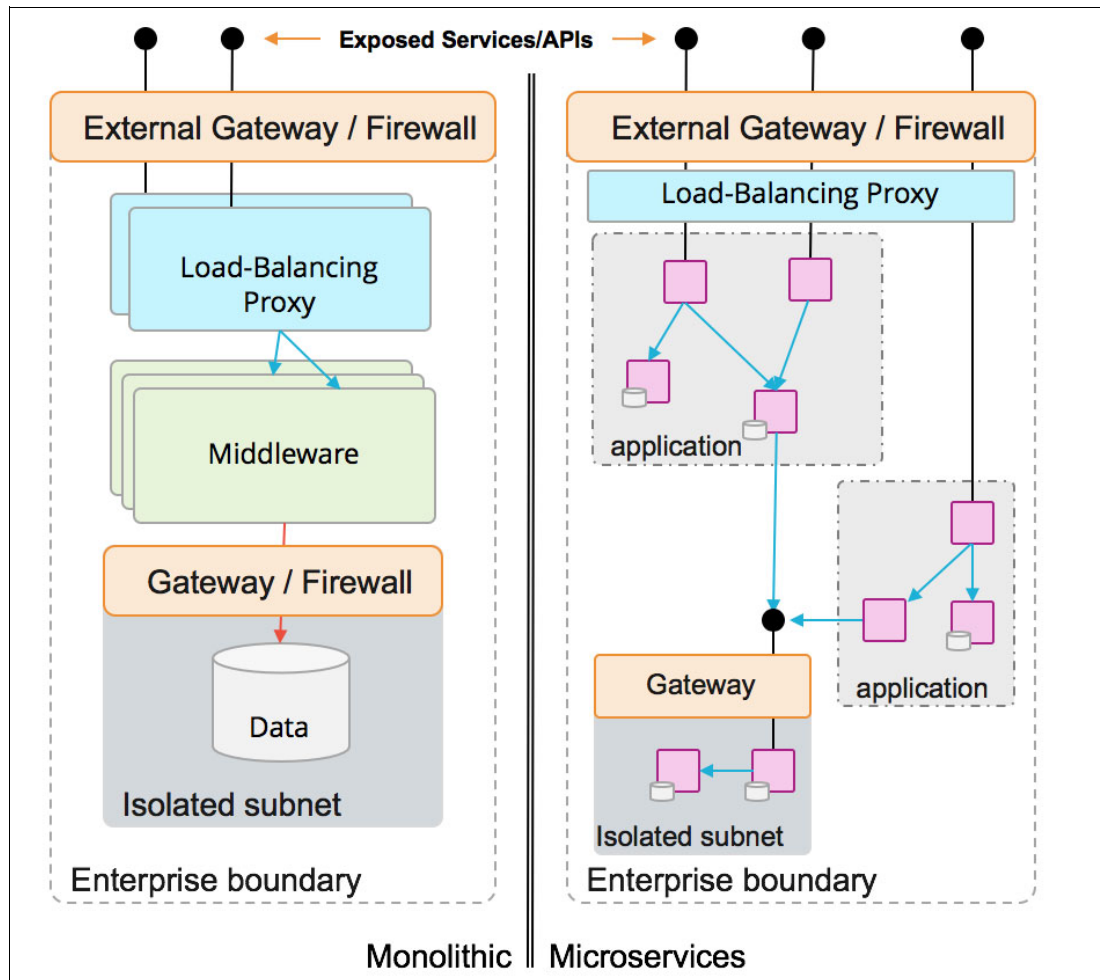


Figure 6-1 Simplified topology for monolithic and microservice architectures

The left side of Figure 6-1 shows a simplification of a traditional monolithic architecture. Requests come through the gateway, go through a load balancer, to business logic in the middleware, and then down to the data tier.

On the right side of Figure 6-1, things are less organized. Collections of microservices are grouped as applications, and surface external APIs as endpoints through a gateway. One obvious difference is the number of moving parts. The other is how many more requests there are.

The biggest difference is that the composition of services shown on the right side are under constant change. Individual service instances come and go based on load. Microservices are continuously updated independently. Overall, there can be hundreds of updates a day across services. Securing the perimeter is not going to be enough. The question is how to secure a rapidly changing infrastructure in an organic way that still allows the individual services to change and grow without requiring central coordination.

6.1.1 Network segmentation

Figure 6-1 on page 68 shows an isolated subnet in both systems. Using an extra firewall or gateway to guard resources that require more levels of protection is a good idea in either environment.

Something else to observe in microservices approach shown in Figure 6-1 on page 68 is that the externally exposed APIs are connected to two different microservices-based applications. Defined application boundaries provide a reasonable amount of isolation between independently varying systems, and are a good way to maintain a reasonable security posture in a dynamic environment.

Network segmentation happens naturally in hosted environments. For example, each microservices application that is shown in Figure 6-1 on page 68 could be a separate tenant in a multi-tenant environment. Where it does not happen naturally, you might want to encourage it. However, describing the best practices for managing networks in microservice architectures is well outside the scope of this publication.

6.1.2 Ensuring data privacy

Different kinds of data require different levels of protection, and that can influence how data is accessed, how it is transmitted, and how it is stored.

When dealing with data, take into account these considerations:

- ▶ Do not transmit plain text passwords.
- ▶ Protect private keys.
- ▶ Use known data encryption technologies rather than inventing your own.
- ▶ Securely store passwords by using salted hashes. For more information about salted hashing, see the following website:
<https://crackstation.net/hashing-security.htm>
- ▶ Further, sensitive data should be encrypted as early as possible, and decrypted as late as possible. If sensitive data must flow between services, only do so while it is encrypted, and should not be decrypted until the data needs to be used. This process can help prevent accidental exposure in logs, for example.

Backing services

As mentioned earlier, connections to backing services are additional sources of network traffic that need to be secured. In many cloud environments, backing services are provided by the platform, relying on multi-tenancy and API keys or access tokens to provide data isolation. It is important to understand the characteristics of backing services, especially how they store data at rest, to ensure regulatory requirements (HIPAA, PCI, and so on) are satisfied.

Log data

Log data is a trade-off between what is required to diagnose problems and what must be protected for regulatory and privacy reasons. When writing to logs, take full advantage of log levels to control how much data is written to logs. For user-provided data, consider whether it belongs in logs at all. For example, do you really need the value of the attribute written to the log, or do you really only care about whether it was null?

6.1.3 Automation

As mentioned way back in Chapter 1, “Overview” on page 1, as much as possible should be automated in microservice environments, which includes general operation. Repeatable automated processes should be used for applying security policies, credentials, and managing SSL certificates and keys across segmented environments to help avoid human error.

Credentials, certificates, and keys must be stored somewhere for automation to use. Remember the following considerations:

- ▶ Do not store your credentials alongside your applications.
- ▶ Do not store your credentials in a public repository.
- ▶ Only store encrypted values.
- ▶ Spring Cloud Config, as an example, uses Git to back up their configuration. It provides APIs to automatically encrypt configuration values on the way into the repository, and to decrypt them on the way out. For more information about Spring Cloud Config, see this website:

<https://cloud.spring.io/spring-cloud-config/>

Hashicorp’s Vault is similar, but adds extra capabilities to allow use of dynamic keys with expiry. For more information about Vault, see the following website:

<https://www.vaultproject.io/>

Other key/value stores, for example etcd, zookeeper, or consul, could also be used, but care should be taken with how credentials are stored in what are otherwise plain text repositories.

6.2 Identity and Trust

A highly distributed, dynamic environment like microservices architectures place some strain on the usual patterns of establishing identity. You must establish and maintain the identity of users without introducing extra latency and contention with frequent calls to a centralized service.

Establishing and maintaining trust throughout this environment is not the easiest either. It is inherently unsafe to assume a secure private network. End-to-end SSL can bring some benefit in that bytes are not flowing around in plain text, but it does not establish a trusted environment on its own, and requires key management.

The following sections outline techniques for performing authentication and authorization and identity propagation to establish and maintain trust for inter-service communications.

6.2.1 Authentication and authorization

There are new considerations for managing authentication and authorization in microservice environments. With a monolithic application, it is common to have fine grained roles, or at least role associated groups in a central user repository. With the emphasis on independent lifecycles for microservices, however, this dependency is an anti-pattern. Development of an independent microservice is then constrained by and coupled with updates to the centralized resource.

It is common to have authentication (establishing the user's identity) performed by a dedicated, centralized service or even an API gateway. This central service can then further delegate user authentication to a third party.

When working with authorization (establishing a user's authority or permission to access a secured resource), in a microservices environment keep group or role definitions coarse grained in common, cross cutting services. Allow individual services to maintain their own fine grained controls. The guiding principle again is independence. A balance must be found between what can be defined in common authorization service to meet requirements for the application as a whole, and what authorization requirements are implementation details for a particular service.

Java EE security

Traditional enterprise access control tokens also have less traction in a microservice environment. The presence of other programming languages and the inability to rely on long-lived sessions undercut the operating assumptions of traditional Java EE security mechanisms.

For Java EE security, each Java EE server must be trusted by all of the other services. The security token (in WebSphere Application Server, this is an LTPA Token) that the user receives after authentication must be valid on the other servers. Establishing trust this way means that either (a) every server needs the certificates of every other server in its keystore, or (b) all of the services need to use the same key. A user authenticating to the system must also be in the security realm defined for your Java EE server. Key management, specifically keystore management for Java applications, in a dynamically provisioned environment is a non-trivial problem to solve. For more information about Java EE Security, see this website:

<https://dzone.com/refcardz/getting-started-java-ee>

If you compare the effort that is required to achieve and maintain this environment to the benefit you get from Java EE security, then it is difficult to recommend using Java EE security.

6.2.2 Delegated Authorization with OAuth 2.0

OAuth provides an open framework for delegating authorization to a third party. Although the framework does define interaction patterns for authorization, it is not a strictly defined programming interface, with some variations between OAuth providers. For more information about the OAuth Authorization Framework, see the following website:

<https://tools.ietf.org/html/rfc6749>

OpenID Connect (OIDC) provides an identity layer on top of OAuth 2.0, making it possible to also delegate authentication to a third party. It defines an interoperable REST-based mechanism for verifying identity based on the authentication steps performed by the OAuth provider, and to obtain basic profile information about the authorized user. For more information about ODIC, see the following website:

<https://openid.net/connect/>

Game On! uses three-legged authentication defined by OAuth and OIDC to delegate to social sign-on providers, specifically Facebook, GitHub, Google, and Twitter. As shown in Figure 6-2, the game has a dedicated Auth service that acts as a proxy for the various social login providers. This Auth service triggers a request to the target OAuth provider including information required to identify the Game On! application.

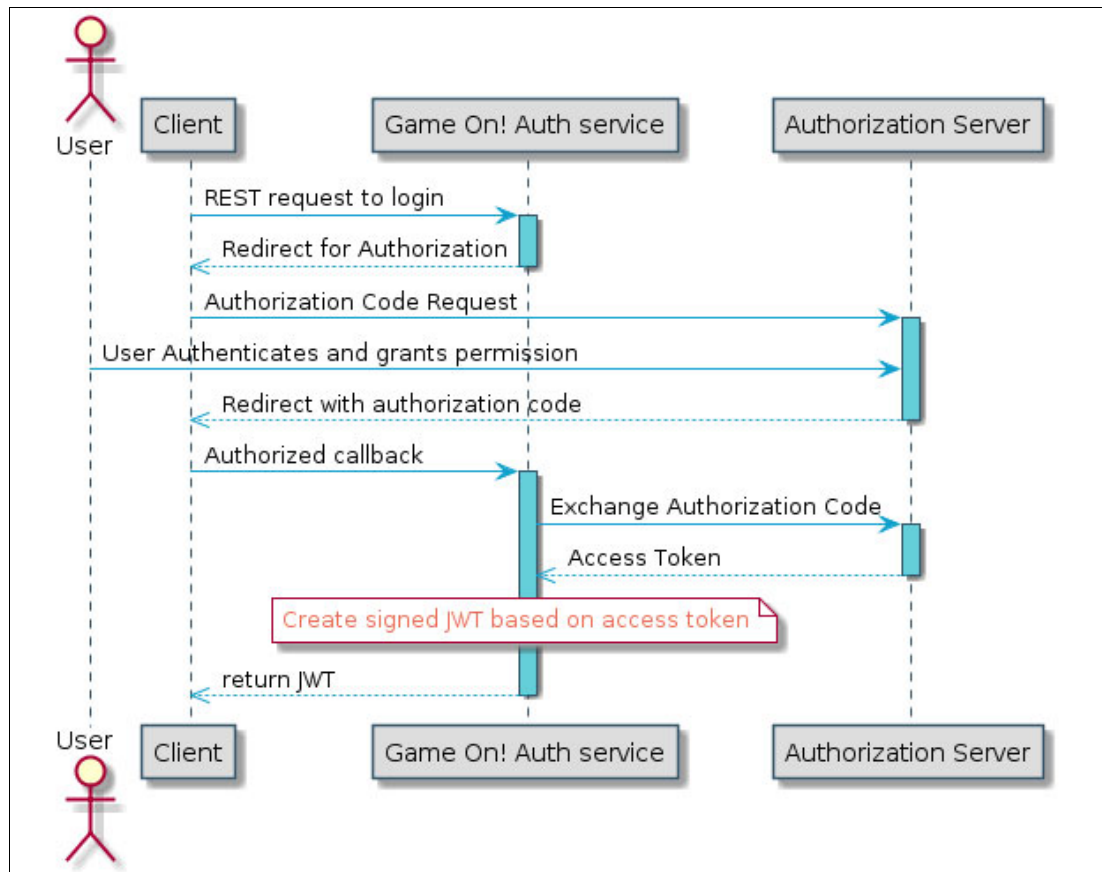


Figure 6-2 OAuth invocation sequence diagram for Game On!

The service interaction goes through these steps:

1. The front end (web client or mobile application) makes a request to the game's Auth service on behalf of the user.
2. The Auth service returns a forwarding response that is automatically handled by the browser to forward the user to the selected OAuth provider with the application identifier.
3. After the user has authenticated and granted permission, the OAuth provider returns an authorization code in another redirect response. The browser automatically handles the redirect to invoke a callback on the Auth service with the authorization code.
4. The Auth service then contacts the OAuth provider to exchange the authorization code for an access token.
5. The Auth service then converts data from that token into a Signed JSON Web Tokens (JWTs), which allows you to verify the identity of the user over subsequent inter-service calls without going back to the OAuth provider.

Some application servers, like WebSphere Liberty, have OpenID Connect features to facilitate communication with OAuth providers. As of this writing, those features do not provide the access token that was returned from the OAuth provider. Also, minute protocol differences can make using individual provider libraries easier than more generic solutions.

A simpler approach might be to perform authentication with an OAuth provider by using a gateway. If you take this approach, it is still a good idea to convert provider-specific tokens into an internal facing token that contains the information that you need.

6.2.3 JSON Web Tokens

Identity propagation is another challenge in a microservices environment. After the user (human or service) has been authenticated, that identity needs to be propagated to the next service in a trusted way. Frequent calls back to a central authentication service to verify identity are inefficient, especially given that direct service-to-service communication is preferred to routing through central gateways whenever possible to minimize latency.

JWTs can be used to carry along a representation of information about the user. In essence, you want to be able to accomplish these tasks:

- ▶ Know that the request was initiated from a user request
- ▶ Know the identity that the request was made on behalf of
- ▶ Know that this request is not a malicious replay of a previous request

JWTs are compact and URL friendly. As you might guess from the name, A JWT (pronounced “jot”) contains a JSON structure. The structure contains some standard attributes (called claims), such as issuer, subject (the user’s identity), and expiration time, which has clear mappings to other established security mechanisms. Room is also available for claims to be customized, allowing additional information to be passed along.

While building your application, you will likely be told that your JWT is not valid for reasons that are not apparent. For security and privacy reasons, it is not a good idea to expose (to the front end) exactly what is wrong with the login. Doing so can leak implementation or user details that could be used maliciously. JWT.IO provides some nice web utilities for working with JWTs, including a quick way to verify whether the encoded JWT that scrolled by in your browser console or your trace log is valid or not. For more information about JWT.IO, see the following website:

<https://jwt.io>

Dealing with time

One of the nice attributes of JWTs is that they allow identity to be propagated, but not indefinitely. JWTs expire, which triggers revalidation of the identity that results in a new JWT. JWTs have three fields that relate to time and expiry, all of which are optional. Generally, include these fields and validate them when they are present as follows:

- ▶ The time the JWT was created (iat) is before the current time and is valid.
- ▶ The “not process before” claim time (nbf) is before the current time and is valid.
- ▶ The expiration time (exp) is after the current time and is valid.

All of these times are expressed as UNIX epoch time stamps.

Signed JWTs

Signing a JWT helps establish trust between services, as the receiver can then verify the identity of the signer, and that the contents of the JWT have not been modified in transit.

JWTs can be signed by using shared secrets, or a public/private key pair (SSL certificates work well with a well known public key). Common JWT libraries all support signing. For more information about JWT libraries, see the following website:

<https://jwt.io/#libraries-io>

Working with JWTs

Common JWT libraries make working with JWTs easy. For example, with JJWT¹, creating a new JWT is straightforward, as shown in Example 6-1.

Example 6-1 Creating and validating a signed JWT using JJWT

```
import java.time.Instant;
import java.time.temporal.ChronoUnit;
import java.util.Date;
import io.jsonwebtoken.Claims;
import io.jsonwebtoken.Jws;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;
public class JwtExample {
    public String createJwt(String secret) throws Exception {
        // create and sign the JWT, including a hint
        // for the key used to sign the request (kid)
        String newJwt = Jwts.builder()
            .setHeaderParam("kid", "meaningfulName")
            .setSubject("user-12345")
            .setAudience("user")
            .setIssuedAt(Date.from(Instant.now()))
            .setExpiration(Date.from(Instant.now().plus(15, ChronoUnit.MINUTES)))
            .signWith(SignatureAlgorithm.HS512, secret)
            .compact();
        return newJwt;
    }
    public void validateJwt(String jwtParameter, String secret) throws Exception {
        // Validate the Signed JWT!
        // Exceptions thrown if not valid
        Jws<Claims> jwt = Jwts.parser()
            .setSigningKey(secret)
            .parseClaimsJws(jwtParameter);
        // Inspect the claims, like make a new JWT
        // (need a signing key for this)
        Claims jwtClaims = jwt.getBody();
        System.out.println(jwtClaims.getAudience());
        System.out.println(jwtClaims.getIssuer());
        System.out.println(jwtClaims.getSubject());
        System.out.println(jwtClaims.getExpiration());
        System.out.println(jwtClaims.getIssuedAt());
        System.out.println(jwtClaims.getNotBefore());
    }
}
```

Example 6-1 is an altered version of what the authentication service in Game On! does to create the JWT from the returned OAuth token². JWTs should be validated in a central place such as a JAX-RS request filter.

A consideration is the longevity of the JWT. If your JWT is long lived, you will have significant reduction in traffic to revalidate JWTs, but might then have a lag in the system reacting to access changes.

¹ <https://github.com/jwtkt/jjwt>

² <https://github.com/gameontext/gameon-auth/>

In the case of Game On!, the user is connected by using long running WebSocket connections to enable 2-way asynchronous communication between third parties. The JWT is passed as part of the query string when the connection is established. Expiring that JWT leads to an unpleasant user experience at the time of writing, so we chose to use a relatively long JWT expiry period.

6.2.4 Hash-Based Messaging Authentication Code

Authentication by using Hash-Based Messaging Authentication Code (HMAC) signatures is superior to HTTP Basic authentication. When using HMAC, request attributes are hashed and signed with a shared secret to create a signature that is then included in the sent request. When the request is received, the request attributes are rehashed by using the shared secret to ensure that the values still match. This process authenticates the user, and verifies that the content of the message has not been manipulated in transit. The hashed signature can also be used to prevent replay attacks.

HMAC validation can be performed by an API Gateway. In this case, the Gateway authenticates the user, and then can either serve cached responses, or produce a JWT or other custom headers to include in the request before forwarding on to backend/internal services. Using a JWT has the advantage of allowing downstream services to consistently use JWTs to work with the caller's identity, independent of the original authentication method (such as OAuth when using a UI, versus HMAC for programmatic invocation).

HMAC is not yet a complete standard, API providers (either gateways or individual services) often require different request attributes (http headers or a hash of the message body) in the generated signature. Minimally, an identifying string and the date are used.

Using shared libraries is encouraged when working with HMAC signatures because HMAC calculations must be performed in the same way by both the API consumer and the provider. Consider providing client-side libraries to help consumers sign their requests correctly.

Game On! does not use an API gateway at the time of writing. We provide a shared library³ to ensure that a common HMAC signing algorithm is used for both REST and WebSocket requests. The shared library provides JAX-RS filters for inbound and outbound requests, and a core utility that can be invoked around the WebSocket handshake between the Mediator⁴ and third-party Room services. See Example 6-2.

For example, the Mediator uses the shared library to sign outbound requests to the Map⁵.

Example 6-2 JAX-RS 2.0 client using SignedClientRequestFilter from shared library

```
Client client = ClientBuilder.newClient()
    .register(JsonProvider.class);
// a filter is added to the JAX-RS client
// the filter will sign the request on the way out
SignedClientRequestFilter apikey = new
    SignedClientRequestFilter(userid, secret);
client.register(apikey);
```

³ <https://github.com/gameontext/signed>

⁴ <https://github.com/gameontext/gameon-mediator>

⁵ <https://github.com/gameontext/gameon-map>

The shared library allows the use of simple annotations to flag server-side endpoints that should have the inbound signature verified. JAX-RS filters and interceptors are provided to verify both request headers and the message body as shown in Example 6-3.

Example 6-3 Map service JAX-RS endpoint: Annotation from shared library to enable signing

```
@POST
@SignedRequest
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public Response createRoom(RoomInfo newRoom) {
    Site mappedRoom = mapRepository.connectRoom(...);
    return Response.created(...).entity(mappedRoom).build();
}
```

Using the shared library in this way keeps the business logic clean. The configuration ensures that the client and server side agree on included attributes and hash calculation as part of producing the library. The shared library simplifies what both the API provider and the consumer need to deal with.

6.2.5 API keys and shared secrets

An API key can be one of a few different things, but the usual usage is to identify the origin of a request. Identifying the origin of a request is important for API management functions like rate limiting and usage tracking. API keys are usually separate from an account's credentials, enabling them to be created or revoked at will.

An API key is sometimes a single string, but unlike a password, these strings are randomly generated and over 40 characters long. However, sometimes the string is used directly in the request as either a bearer token or query parameter (over https). Using an API key is more secure when used as a shared secret for signing tokens (JWTs) or for digest methods of request authentication (HMACs).

If your service creates API keys, ensure that API keys are securely generated with high entropy and are stored correctly. Ensure that these keys can be revoked at any time. If your application has distinct parts, consider using different API keys for each part. This configuration can enable more granular usage tracking, and reduces the impact if a key is compromised.

Asymmetric keys can also be used (such as SSL), but require a public key infrastructure (PKI) to be maintained. Although it is more secure, this configuration is also more labor intensive, and can be especially hard to manage when API consumers are third parties.

Consistent with the 12 factors, secrets should be injected configuration when used by services for intra-service communication. The values should never be hardcoded because that increases the likelihood they will be compromised (such as checked into a repository in plain text). It either makes your code sensitive to the environment, or requires different environments to use the same shared secret, neither of which are good practices.



Testing

It is essential that a microservice application is built with an awareness of how it can be tested. Having good test coverage gives you more confidence in your code and results in a better continuous delivery pipeline.

This chapter covers the tests that are required at three different stages in the build lifecycle:

- ▶ Single service testing
Tests carried out in isolation by a team that owns a particular microservice in a system.
- ▶ Staging environment
Tests that are run on objects in a staging environment. The microservices that form a particular application are deployed into a staging environment for testing.
- ▶ Production environment
Tests carried out on the live production system.

Tests should be automated as part of the build, release, run (delivery) pipeline.

The following topics are covered:

- ▶ Types of tests
- ▶ Application architecture
- ▶ Single service testing
- ▶ Staging environment
- ▶ Production environment

7.1 Types of tests

This chapter focuses on the Java specific testing considerations, assuming some prior knowledge of testing microservices. The following types of tests are covered:

- ▶ **Unit**
Tests a single class or a set of closely coupled classes. These unit tests can either be run using the actual objects that the unit interacts with or by employing the use of test doubles or mocks.
- ▶ **Component**
Tests the full function of a single microservice. During this type of testing, any calls to external services are mocked in some way.
- ▶ **Integration**
Integration tests are used to test the communication between services. These tests are designed to test basic success and error paths over a network boundary.
- ▶ **Contract**
Test the agreed contract for APIs and other resources that are provided by the microservice. For more information, see “Consuming APIs” on page 38.
- ▶ **End-to-end**
Tests a complete flow through the application or microservice. Usually used to test a golden path or to verify that the application meets external requirements.

7.2 Application architecture

The internal structure of your microservice directly affects how easy it is to test. Figure 7-1 shows a simplified version of the microservice architecture described in Chapter 2, “Creating Microservices in Java” on page 9.

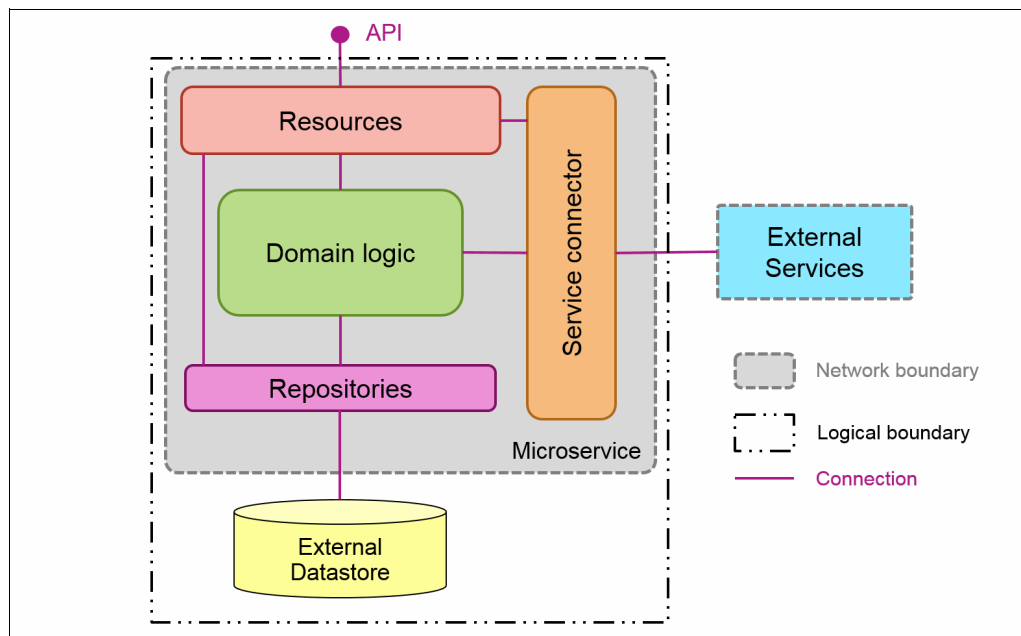


Figure 7-1 The internal microservice architecture.

The different internal sections break down into the following subsections:

- **Resources**

Exposes JAX-RS resources to external clients. This layer can handle basic validation of requests, but any business logic is passed on.

- **Domain logic**

This area is where the bulk of the functional code goes. This code has no understanding of external dependencies to the application.

- **Service connector**

Handles communication with external services. This can be services within the microservice system or third-party services. Responses are validated and then passed to the domain logic layer.

- **Repositories**

Includes data mapping and validation. The repositories layer has the logic that is required to connect to the external data store.

The different sections of the application have different testing requirements. For example, domain logic does not require integration testing, whereas resources do. All of the code in a particular Java class should provide function from the same section. Separating the code in this way results in all of the code in a particular class having the same testing requirements. This technique makes your test suite simpler and better structured. Given the inherent complexity of microservices, it is easy to get into a mess with your testing, so anything that can add structure to your tests is a good idea.

7.3 Single service testing

This section defines single service testing as the tests that the owners of an individual service must create and run. These tests exercise the code that is contained inside the logical boundary in the diagram of the internal microservice architecture as shown in Figure 7-1 on page 78. Three types of tests apply here: Unit tests, component tests, and integration tests. Unit and integration tests are used to test the individual parts of the microservice, and component tests are used to test the service as a whole.

7.3.1 Testing domain or business function

The code in your microservice that performs business function should not make calls to any services external to the application. This code can be tested by using unit tests and a testing framework such as JUnit¹. The unit tests should test for behavior and either use the actual objects (if no external calls are needed) or mock the objects involved in any operations.

When writing tests using the actual objects, a simple JUnit test suffices. For creating mocks of objects, you can either use the built-in capabilities of Java EE or use a mocking framework. The `@Alternatives` annotation² in the Context and Dependency Injection (CDI) specification enables injection of mock objects instead of the actual beans. Plenty of mocking frameworks are available for Java. For example, JMockit³ is designed to work with JUnit to allow you to mock objects during testing. In the most basic test using JMockit, you can create mocked objects by using the `@Mocked` annotation and define behavior when this object is called by using the `Expectations()` function.

¹ <http://junit.org/junit4/>

² <http://docs.oracle.com/javaee/6/tutorial/doc/gjsdf.html>

³ <http://jmockit.org/>

Consider an online payment application that has an “Order” microservice. Suppose that a class called `OrderFormatter` takes in an account ID and returns an order history in a particular format as in Example 7-1. The `DatabaseClient` class handles requests to the database to get the list of orders. The `OrderFormatter` is `@ApplicationScoped` so that it can be injected into any REST endpoints that use it.

Example 7-1 The `OrderFormatter` class formats the response from the `Orders` class

```
@ApplicationScoped
public class OrderFormatter {
    @Inject
    DatabaseClient client;
    public FormattedHistory getOrderHistory(String id) {
        History orderHistory = client.getHistory(id);
        FormattedHistory formatted = format(orderHistory);
        return formatted;
    }
    private FormattedHistory format(History history) {
        ...
    }
}
```

To test this class, make the assumption that the `DatabaseClient` class will return the correct orders from the `getHistory()` function. The behavior that you want to test is the actual formatting. However, you cannot call the `format` object directly because it is a private method. You want to avoid altering the code to enable testing, such as by changing the `format()` method to `public`. Instead, use a mocked version of the `DatabaseClient` class. Example 7-2 shows a simple test of the `OrderFormatter` class using `JMockit` and `JUnit`. The `@Tested` annotation indicates that the class that is being tested. This technique allows test objects to be injected into the class when the test is running. The `@Injectable` annotation is used to inject a mocked version of the `DatabaseClient` into the `OrderFormatter` object when the test runs. The `Expectations()` function defines the behavior of the mocked object.

Example 7-2 `JMockit` allows you to define what particular methods will return

```
public class TestOrderFormatter {
    @Tested OrderFormatter formatter;
    @Injectable DatabaseClient databaseClient;
    @Test
    public void testFormatter() {
        String id = "Test ID";
        new Expectations() {{
            History history = // set history to be formatted
            databaseClient.getHistory(id); returns(history);
        }};
        FormattedHistory expected = // set expected object
        FormattedHistory actual = formatter.getOrderHistory(id);
        assertEquals(expected.toString(), actual.toString());
    }
}
```

7.3.2 Testing resources

Classes that expose JAX-RS endpoints or receive events should be tested by using two types of tests: Integration tests and contract tests.

Integration tests

Integration tests are used to verify the communication across network boundaries. They should test the basic success and failure paths in an exchange. Integration tests can either be run in the same way as unit tests, or by standing up the application on a running server. To run integration tests without starting the server, call the methods that carry JAX-RS annotations directly. During the tests, create mocks for the objects that the resource classes call when a request comes in.

To run integration tests on a running server, you can use one of the methods described in “Tests on a running server” on page 82. To drive the code under test, use the JAX-RS client provided by JAX-RS 2.0. to send requests.

Integration tests should validate the basic success and error paths of the application. Incorrect requests should return useful responses with the appropriate error code.

Consumer driven contract

A consumer of a particular service has a set of input and output attributes that it expects the service to adhere to. This set can include data structures, performance, and conversations. The contract is documented by using a tool like Swagger. See Chapter 2, “Creating Microservices in Java” on page 9 for more information. Generally, have the consumers of a service drive the definition of the contract, which is the origin of the term *consumer driven contract*.

Consumer driven contract tests are a set of tests to determine whether the contract is being upheld. These tests should validate that the resources expect the input attributes defined in the contract, but also accepts unknown attributes (it should just ignore them). They should also validate that the resources return only those attributes that are defined in the documentation. To isolate the code under test, use mocks for the domain logic.

Maintaining consumer driven contract tests introduces some organizational complexity. If the tests do not accurately test the contract defined, they are useless. In addition, if the contract is out of date, then even the best tests will not result in a useful resource for the consumer. Therefore, it is vital that the consumer driven contract is kept up to date with current consumer needs and that the tests always accurately test the contract.

Contract tests require the actual API to be implemented. This technique requires the application be deployed onto a server. See “Tests on a running server” on page 82 for more information. Use tools such as the Swagger editor⁴ to create these tests. The Swagger editor can take the API documentation and produce implementations in many different languages.

Another dimension to contract testing is the tests that are run by the consumer. These tests must be run in an environment where the consumer has access to a live version of the service, which is the *staging environment*. See section 7.4.3, “Contract” on page 86 for more details.

⁴ <http://editor.swagger.io/>

Tests on a running server

A few different methods are available for starting and stopping the application server as part of your automated tests. There are Maven and Gradle plug-ins for application servers such as WebSphere Application Server Liberty that allow you to add server startup into your build lifecycle. This method keeps complexity out of your application and contains it in the build code. For more information about these plug-ins, see the following websites:

- Maven
<https://github.com/WASdev/ci.maven>
- Gradle
<https://github.com/WASdev/ci.gradle>

Another solution is to use Arquillian. Arquillian can be used to manage the applications server during tests. It allows you to start and stop the server mid-test, or start multiple servers. Arquillian is also not affected by the container, so if you write Arquillian tests, they can be used on any application server. This feature is useful for contract testing because the consumers do not have to understand the application server or container that is used by the producer. For more information about Arquillian, see the following website:

<http://arquillian.org>

For a more detailed comparison of these different methods, see “Writing functional tests for Liberty” at:

<https://developer.ibm.com/wasdev/docs/writing-functional-tests-liberty/>

7.3.3 Testing external service requests

Inevitably, your microservice must make calls to external services to complete a request, such as calls to other microservices in the application or services external to the application. The classes to do this construct clients that make the requests and handle any failures. The code can be tested by using two sets of integration tests: One at the single service level and one in the staging environment. Both sets test the basic success and error handling of the client. More information about the tests in the staging environment is available in 7.4.2, “Integration” on page 86.

The integration tests at the single service level do not require the service under test or the external services to be deployed. To perform the integration tests, mock the response from the external services. If you are using the JAX-RS 2.0 client to make the external requests, this process can be done easily by using the JMockit framework.

Consider the online store example introduced in 1.5.1, “Online retail store” on page 7. Suppose the Account service must make a call to the Order service to get the list of recent orders for a particular account. The logic to make the request should be encapsulated in a class similar to Example 7-3. The `OrderServiceClient` is an `@ApplicationScoped` bean that is injected into the classes that use it. We have chosen to use CDI to inject the URL for the requests, rather than using a library that gets the service list from the registry as part of the request. In this case, we are just returning an empty `History` object if the response from the Order service is bad. Take advantage of the `@PostConstruct` methods to initialize the JAX-RS client because you can then call the method directly in the test if required.

Example 7-3 OrderServiceClient handles all requests to the Order service

```
@ApplicationScoped
public class OrderServiceClient {
    @Resource(lookup = "orderServiceUrl")
```



```

String orderServiceUrl;

private Client client;

@PostConstruct
public void initClient() {
    this.client = ClientBuilder.newClient();
}

public History getHistory(String id) {
    String orderUrl = orderServiceUrl + "/" + id;
    WebTarget target = client.target(orderUrl);
    Response response = callEndpoint(target, "GET");
    History history = getHistoryFromResponse(response);
    return history;
}

private History getHistoryFromResponse(Response response) {
    if (response.getStatus() == Response.Status.OK.getStatusCode()) {
        return response.readEntity(History.class);
    } else {
        return new History();
    }
}

private Response callEndpoint(WebTarget target, String callType) {
    Invocation.Builder invoBuild = target.request();
    Response response = invoBuild.build(callType).invoke();
    return response;
}
}

```

To test this class, mock the response from the Order service as shown in Example 7-4. By putting the logic that interprets the response in a separate method to the one making the call, you can test this logic by directly invoking that method. JMockit provides a *Deencapsulation* class to allow invocation of private classes from tests. This technique provides test coverage for the handling of response codes from external services. It is also possible to mock the JAX-RS *WebTarget* or *InvocationBuilder* if your class structure requires this. Higher-level coverage of external requests is done in the staging environment.

Example 7-4 The response from the Order service is mocked to enable isolated testing

```

public class OrderServiceClientTest {

    @Mocked
    Response response;

    OrderServiceClient orderHistoryClient = new OrderServiceClient();

    @Test
    public void test200() {
        new Expectations() {{
            History user = new History();
            user.addOrder("Test");
            response.getStatus(); returns(200);
            response.readEntity(History.class); returns(user);
        }};
        History history = Deencapsulation.invoke(orderHistoryClient,
            "getHistoryFromResponse", response);
    }
}

```

```

        assertTrue("Expected string Test", history.getOrders().contains("Test"));
    }
    @Test
    public void test500() {
        new Expectations() {{
            response.getStatus(); returns(500);
        }};
        History history = Deencapsulation.invoke(orderHistoryClient,
            "getHistoryFromResponse", response);
        assertTrue("Expected empty order list", history.getOrders().isEmpty());
    }
}

```

7.3.4 Testing data requests

In a microservice architecture, each microservice owns its own data. See Chapter 5, “Handling data” on page 41 for more information. If you follow this guideline, the developers of a microservice are also responsible for any external data stores used. As described in 7.2, “Application architecture” on page 78, the code that makes requests to the external data store and performs data mapping and validation is contained in the repositories layer. When testing the domain logic, this layer should be mocked. Tests for data requests, data mapping, and validation are done by using integration tests with the microservice and a test data store deployed locally or on a private cloud. The tests check the basic success and error paths for data requests. If the data mapping and validation for your application requires extensive testing, consider separating out this code and testing it using a mocked database client class.

Test data

The local version of the data store must be populated with data for testing. Think carefully about what data you put in the data store. The data should be structured in the same way as production data but should not be unnecessarily complicated. It must serve the specific purpose of enabling data request tests.

7.3.5 Component testing

Component tests are designed to test an individual microservice as one piece. The component is everything inside the network boundary, so calls to external services are either mocked or are replaced with a “test-service.” There are advantages and disadvantages to both scenarios.

Using mocks

By mocking the calls to external services, you have fewer test objects to configure. You can easily define the behavior of the mocked system by using frameworks like JMockit, and no tests will fail due to network problems. The disadvantage of this approach is that it does not fully exercise the component because you are intercepting some of the calls, increasing the risk of bugs slipping through.

Test services

To fully exercise the communication boundaries of your microservice, you can create test services to mimic the external services that are called in production. These test services can also include a test database. The test services can also be used as a reference for consumers of your microservice. The disadvantage of this system is that it requires you to maintain your test services. This technique requires more processor cycles than maintaining

a mocking system as you must fully test the test microservice and create a deployment pipeline.

After you are using a mocking framework for other levels of testing, it makes sense to reuse those skills. However, if you do take the mocking approach, you must make sure that the tests in your staging environment exercise inter-service communications effectively.

7.3.6 Security verification

Security is important in a distributed system. You can no longer put your application behind a firewall and assume that nothing will break through. For more details about securing your microservices, see Chapter 6, “Application Security” on page 67.

Testing the security of your microservice is slightly different depending on how you implement security. If the individual services are just doing token validation, then test at the individual service level. If you are using another service or a library to validate the tokens, then that service should be tested in isolation and the interactions should be tested in the staging environment.

The final type of tests to run is security scanners such as IBM Security AppScan® to highlight security holes. AppScan scans your code and highlights any vulnerabilities. It also provides recommendations for fixing the vulnerabilities. For more information about Security AppScan, see the following website:

<http://www.ibm.com/software/products/en/appscan>

7.4 Staging environment

This section defines a staging environment as a test environment that is identical (where possible) to the production environment. The build pipeline deploys successfully tested microservices to the staging environment where tests are run to verify the communication across logical boundaries, that is, between microservices.

7.4.1 Test data

The staging environment should include any data stores that will be in your production system. As described in 7.3.4, “Testing data requests” on page 84, the test data should not be unnecessarily complicated. The data in this data store will be more complete than at the individual microservice level, as these tests are testing more complicated interactions.

Use tools to inject data into tests for you. Tools allow you to have more control over the flow of data around the system, enabling you to test what happens if bad data is introduced. The microservice orchestration and composition solution Amalgam8 offers this feature using its controller component. The controller passes information between the service registry and the microservice side-car. It can inject data dynamically to simulate the introduction of bad data into the system.

For more information about Amalgam8, see the following website:

<https://www.amalgam8.io/>

7.4.2 Integration

Integration tests are used to test the interactions between all the services in the system. The in-depth behavior of the individual services has already been tested at this stage. The consumer driven contract tests should have ensured that the services interact successfully, but these tests identify bugs that have been missed. The tests should check the basic success and error paths of service communication with the application deployed. Use the test data as discussed in the previous section.

Rather than testing all of the services at once, it might still be necessary to mock out some of the services during testing. Test the interaction of two specific services, or a small set of services, adding in mocked behavior when calls are made to outside the set. Mocking the calls to the services outside the group under test is done in the same way as the unit tests on the APIs. The same techniques that are used to start and stop the server or container for contract testing also apply here (see 7.3.2, “Testing resources” on page 81).

7.4.3 Contract

Every service that consumes another service or resource should have a set of contract tests that are run against that resource (especially in staging environments). Given that services evolve independently over time, it is important to ensure that the consumer's contract continues to be satisfied.

These tests are specifically written by the consumer (the client side), and are run and managed as part of the test suite for the consuming service. By contrast, the tests that are discussed in 7.3.2, “Testing resources” on page 81 are written by the provider to verify the contract as a whole.

7.4.4 End-to-end

End-to-end testing is essential to find the bugs that were missed previously. End-to-end tests should exercise certain “golden paths” through the application. It is unrealistic to test every path through the application with an end-to-end test, so identify the key ones and test those. A good way to identify these paths through the environment is to review the key external requirements of an application. For example, if your application is an online retail store you might test the following paths:

- ▶ User logs in
- ▶ User purchases an item
- ▶ User views the summary of the order
- ▶ User cancels the order

End-to-end testing should also include the user interface. Tools such as SeleniumHQ can be used to automate interactions with a web browser for testing. For more information about SeleniumHQ, see the following website:

<http://www.seleniumhq.org/>

7.4.5 Fault tolerance and resilience

A microservice should be designed in a fault tolerant way as described in 4.2, “Fault tolerance” on page 37. Various techniques can be used to increase the resilience and fault tolerance of individual microservices, but you should still test how fault tolerant your system is as a whole. Tests for fault tolerance should be performed with all of the services deployed. Do not use any mocks.

Taking down individual microservices

In a microservice system, you must not rely on all of the microservices being available at any one time. During the testing phase, make requests to the system while taking down and redeploying individual services. This process should include the microservices in the system and backend data stores. Monitor the time for requests to return and identify an acceptable response time for your application. If the response times are too long, consider reducing the timeout values on your services or altering the circuit breaker configurations.

There are tools that can automate this process, the best known of which within the Java community is Netflix Chaos Monkey, which terminates instances of virtual machines to allow testing of fault tolerance. For more information about Netflix Chaos Monkey, see the following website:

<https://github.com/Netflix/SimianArmy/wiki/Chaos-Monkey>

Other tools, such as Gremlin, which is used by Amalgam8, take the approach of intercepting and manipulating calls to microservices rather than shutting down instances. For more information about Gremlin, see this website:

<https://github.com/ResilienceTesting/gremlinsdk-python>

Injecting bad data

As in the integration tests, tools such as Amalgam8 can be used to automate the injection of bad data during testing. If you have successfully used the bulk head pattern, then bad data in one microservice should not propagate to other microservices.

Stress testing

Microservices should be able to handle unexpected loads. Stress testing should be used to test the bulk heads in your microservices. If a particular microservice is holding up requests, look at the configured bulk heads.

Recoverability testing

Individual microservices should also be resilient to problems in their own deployment. If a container or virtual machine goes down, the microservice needs to be able to recover quickly. Orchestration tools such as Amalgam8 or Kubernetes⁵ (for container orchestration) will spin up new instances of any application if one goes down. A microservice must have fast startup time and, when a shutdown is required, it should shut down gracefully. Using an application server that provides a fast startup time is also essential.

Couple your recoverability testing with taking down service instances. Allow your orchestration system to create the new instances and monitor how long it takes to start the new service.

7.5 Production environment

To really test your system, you should consider running tests on the production environment. With so many moving parts and services constantly coming and going, you should not expect the system to continue working perfectly indefinitely. It is important to add monitoring and analytics to your application to warn of any bad behavior. For more information, see Chapter 9, “Management and Operations” on page 107.

⁵ <http://kubernetes.io/>

Repeat these specific tests from the staging environment in the production environment:

- ▶ Injecting test or bad data
- ▶ Taking down services, both to test the fault tolerance of other services and the recoverability of the service you took down
- ▶ Security verification

Use the same tools and techniques as in the staging environment. Be careful about when you run these tests. Do not run them during peak hours. Instead, select a timeslot that has the smallest impact on your clients as possible, just in case something goes down.

7.5.1 Synthetic monitoring

To determine the health of your production system, create synthetic transactions to run through your application. Use a browser emulator to instigate the synthetic transactions and monitor the responses. Look for invalid responses or large response times. You can also run the automated build system periodically. If you are committing new code frequently, the builds will also be running frequently, but it is likely that particular services might go through a time where fewer commits are made. During this time, if you continue running builds, you will discover any changes to the application dependencies that might cause problems.

7.5.2 Canary testing

Canary tests are named after the proverbial canary taken into the coal mine to serve as an early indicator of unsafe operating conditions for miners. They are minimal tests that verify all dependencies and basic operating requirements for a service are present and functioning. If the environment is unhealthy, canary tests fail quickly, providing an early indicator for a botched deployment.



From development to production

This chapter covers how individual microservices are built and deployed. To that end, review the characteristics that microservices should have that facilitate a smooth transition from development to production:

- ▶ A microservice must be independently deployable.
- ▶ A new version of a microservice should be deployable in minutes rather than in hours.
- ▶ A microservice must be fault tolerant, and should prevent cascading failures.
- ▶ A microservice should not require any code changes as it is deployed across target environments.

As mentioned earlier in the book, microservice architectures require good automated deployment tools to help manage the deployment, testing, and promotion of services across target environments.

The following topics are covered in this chapter:

- ▶ Deployment patterns
- ▶ Deployment pipelines and tools
- ▶ Packaging options
- ▶ Configuration of the applications across stages

8.1 Deployment patterns

A traditional monolith is built and deployed as one unit. A deployment monolith is built of many parts, many individual Java Platform, Enterprise Edition archives, for example, that are combined into one deployed artifact. Microservices, by contrast, are many different pieces that are all deployed independently. This configuration provides more flexibility in how consumers are exposed to service changes:

- Blue/Green Deployment

Blue/Green deployments are a common pattern for monoliths. The key aspect of this approach is that you are maintaining two working versions of your environment such that one can be used as a rollback to the other if a deployment failure occurs. The process goes something like this:

- a. You have a Blue system that is active and running.
- b. You set up the Green system, test it, verify it is working properly, and then switch traffic to the Green system. The Blue system is left alone as a rollback should something fail as traffic moves to the Green system.
- c. After the Green system is stable, the Blue system is prepared with the next outgoing updates, and the process is repeated to switch from the Green system back to the Blue system.

In the microservices context, this replicated environment still applies. Each system (Blue and Green) contains all of your microservices running with stable versions in a configuration that works. Periodic, coordinated updates of otherwise uncoupled services can be managed this way, with updates gathered and stabilized with each other into a known functioning system that is enabled (or disabled) as a set.

- Canary Releasing

Canary releases are also known as phased or incremental rollouts. A canary release aims to discover and contain failure when deploying a new service. Just as with Blue/Green deployments, canary releases start with a duplicated environment. After the new function is introduced into one of the two systems, traffic is gradually (and selectively) routed to the new function to ensure it behaves and performs as expected.

- Toggling

Feature toggles are flags that are used inside the service to enable or disable new function. This process can effectively be used to allow service consumers to opt in to testing new function (perhaps using custom HTTP headers) without having to maintain different versions of the service or engineer coordinated rollouts. Toggling is common in monolithic environments, too, as a way to protect behavior changes from impacting existing consumers.

Toggles can be set in the environment such that a service either has the function enabled or it does not. They can be dynamic configuration settings that act as a binary mode for the service, or they can be combined with per request mechanisms like HTTP headers to turn new functions on or off per request.

Toggles can be used to make routing decisions, which allows them to be a selector for service-centric canary releases. They can also be used for A/B testing, where you are trying to make a side-by-side comparison of alternative implementations.

8.2 Deployment pipelines and tools

One of the benefits of microservice architectures is agility. Because each individual service has a focused scope and an independent lifecycle, it takes less time to implement a new feature as a microservice, test it, and deploy it into production. Microservices are also often built iteratively, which means supporting the notion of continuous integration and deployment.

Deployment artifacts often run on different systems, so many systems must be configured to make the services work. The more manual steps there are for deploying a microservice into production, the more likely it is that unexpected problems will arise due to bad typing or forgotten steps. Human intervention should be reduced to a minimum. For example, manual approvals might be required before a new version of a service can be released into production environment for regulatory reasons. However, everything leading up to, and following from, that approval can be automated).

Deployment pipelines as described by Martin Fowler¹ are used as a part of a Continuous Delivery environment². They describe or define a set of steps to follow to get microservices into production, usually by breaking the process into stages that cover building, deploying, and then testing the service in each stage to verify that all is well. It is helpful to use DevOps technologies to simplify the creation of a Deployment pipeline and related tool chain to support moving applications from development to production. For more information about DevOps technologies, see this website:

<http://blog.xebialabs.com/2015/07/20/xebialabs-launches-the-periodic-table-of-devops-tools/>

As mentioned earlier, the source for your microservices should be in a Source Code Management (SCM) tool like Subversion³ or GIT⁴. It should be built with a build tool that includes dependency management like Maven⁵ or Gradle⁶. Choose one of the repository management tools (such as Nexus⁷ or Artifactory⁸) to store versioned binary files of your shared libraries. Build your services using one of the CI-Tools like Jenkins⁹ or Bamboo¹⁰.

After the automated build of the microservice testing must be done, and for some of that testing you need to deploy the microservice, which means you need the correct configurations placed on your automatically provisioned system. You can use classic scripting to configure the target stages correctly, but you must then manage platform or other system differences in the scripts. Scripting frameworks/infrastructures like Chef¹¹, Puppet¹², Ansible¹³, and Salt¹⁴ can handle this situation much better. They all can be used the same way for initial installation and for modification of an existing system. You describe the target content of the configuration files, and these tools deal with the difference if there are changes on the target system.

¹ <http://martinfowler.com/bliki/DeploymentPipeline.html>

² <http://martinfowler.com/bliki/ContinuousDelivery.html>

³ <http://subversion.apache.org/>

⁴ <http://git-scm.com/>

⁵ <http://maven.apache.org/>

⁶ <http://gradle.org/>

⁷ <http://www.sonatype.org/nexus/>

⁸ <http://www.jfrog.com/artifactory/>

⁹ <http://jenkins.io/>

¹⁰ <http://www.atlassian.com/software/bamboo>

¹¹ <http://www.chef.io/>

¹² <http://puppet.com/>

¹³ <http://www.ansible.com/>

¹⁴ <http://saltstack.com/>

Containerization can also help you to get a correctly configured system. For dynamic provisioning, you can use Cloud hosting services infrastructure as a service (IaaS) or platform as a service (PaaS). Due to the high number of microservices in your system, it is good to have a tool to help you with Release Management. All of these deployed systems must now be monitored (see Chapter 9, “Management and Operations” on page 107). Logs must also be gathered for auditing and diagnostic purposes (see 9.2, “Logging” on page 111). Last, but not least, the security holes in your microservice should be found, as detailed in Chapter 6, “Application Security” on page 67.

It is essential that the build steps to create the microservice are done with a build tool that can also manage dependencies of JAR files. Maven and Gradle are the best choice to do the job. In both build systems, you can list the needed JAR files to build your microservice to ensure that every build of a service leads to an immutable software artifact. Generally, have a distinct version or revision indicator for every build result of a microservice to make it clear which version of the service is deployed. This goal can be achieved with Maven or Gradle. This version of the service should not be confused with a semantic API version: it just determines which version of the codebase is running,

A sample window of a build pipeline “The Pipeline View” used in Jenkins (CI server) is shown at:

<http://wiki.jenkins-ci.org/display/JENKINS/Build+Pipeline+Plugin>

As you can see in the sample window, versions of the same application are running in parallel through the pipeline. The different rectangles show the deployment state of this software version on the stages.

Templates of microservices (9.3, “Templating” on page 113) can be helpful to develop microservices that need the same implementations for the supporting systems they access. Logging, for example, is necessary in a system of microservices, and is also helpful to track the requests between microservices. If every development team of a service does its own logging and also uses its own logging format, it can be more difficult to join the log files of the different services. Security is also a requirement that must be developed to central specifications. Templates can codify these common concerns, and turn them into conventions that are easy to follow. However, if developers feel restricted by these templates, they will ignore or get rid of them. Balance the effort spent building templates with the value they bring to the developers who are writing services.

8.3 Packaging options

The next question, after building the microservice, is which packaging format should be used to deploy the application. A recurring theme in both the twelve factors and the definition of cloud native applications is the creation of immutable artifacts that can be run without change across different deployment environments.

The more things change from deployment to deployment, the greater the risk that something will break in unexpected ways, due to misconfiguration, missing dependencies, and so on. Building immutable, self-contained artifacts introduces more consistency between development and production environments, and reduces the likelihood of deployment time surprises.

The following packaging options are available, among others:

- ▶ WAR/EAR file deployment on preinstalled middleware
- ▶ Executable JAR file
- ▶ Containers
- ▶ Every microservice on its own virtualized server

In general, you want to minimize differences between the environment in which the service is developed and tested, and the environment in which it runs in production. Some packaging and deployment choices make it easier to achieve parity between development and production environments than others. The following sections explore the pros and cons of each packaging option.

8.3.1 JAR, WAR, and EAR file deployment on preinstalled middleware

This variant of packaging is the default way for deploying Java EE applications, where WAR or EAR files are deployed to a dedicated system with a preinstalled application server. In traditional environments, many EARs and WARs are deployed to the same long running server, with deployments batched per week, month, or even longer time frames.

On the way from development to production, the application gets hosted in different application servers in different staging environments (such as quality assurance, performance test, and user acceptance test). Although every environment should have the same configuration, this is not always the case. For example, a production system might use a more conservative approach to upgrades than development or test systems do.

The following problems can arise from this approach:

- ▶ Frequent deployments in a shorter time period (day, hour) can be hard to achieve. For example, some application deployments need a restart of the application server and as a consequence interrupt the other applications on the same application server. This interruption often cannot be done daily or even hourly.
- ▶ If different applications are hosted together on the same application server, the load from one application can interfere with the performance of the other application.
- ▶ High variations in the load of an application cannot easily be adopted in the infrastructure.
- ▶ Different stages might not have the same configuration, which can lead to application problems in one stage that cannot be reproduced in the other stage.
- ▶ Separated responsibilities often lead to a discussion between developer and administrator to find a “guilty” person when deployment problems arise.

If this pattern is used with microservices, the relationship should be one-to-one between the service and the preinstalled middleware server. This configuration is common, for example, in PaaS environments, where you push only the WAR or EAR to a freshly built, predefined server. It can also apply to cases where application server instances are provisioned by using tools like Chef or Puppet, with the WAR or EAR added afterward. The isolation of services into their own middleware instances is important to allow services to scale independently, and to reduce the risk of service updates interfering with each other.

As mentioned, this approach has a deployment risk even if services are isolated because the service still relies on the preinstalled middleware to supply all of the dependencies that it requires. However, because it mirrors much of the traditional monolith application server experience, it can be an easy first step when starting a new microservices application.

8.3.2 Executable JAR file

One way to avoid depending on the deployment environment to provide dependencies is to pack everything that is needed to run the service inside an executable JAR file. These fat or über JAR files contain the service class files and supporting shared libraries, including servlet containers or other Java EE middleware. This technique eliminates problems introduced by dependency mismatches between development and test or production deployment environments.

These self-contained JAR files are generally easy to handle, and help create a consistent, reproducible environment for local development and test in addition to target deployment environments.

As mentioned in 2.1, “Java platforms and programming models” on page 10, there are various programming models available that produce stand-alone executable JAR files.

Spring Boot

The executable JAR files produced by Spring Boot¹⁵ contain a Spring application, its dependent JAR files, and an underlying web application run time (Tomcat¹⁶, Jetty¹⁷, or Undertow¹⁸). Most of the application configuration is automatically done by Spring Boot itself and can be changed from outside the application. Additional functions for production support, like metrics and health checks, are also included in this package.

There is a quick and easy way to get started. First, you need a build script, like the maven file shown in Example 8-1.

Example 8-1 Spring Boot Maven Quick Start

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.4.1.RELEASE</version>
</parent>
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
</dependencies>
```

You must implement a Controller to respond to the HTTP requests that is started by SpringApplication (Example 8-2).

Example 8-2 Spring Boot Controller

```
@Controller
@EnableAutoConfiguration
public class MyController {
    @RequestMapping("/myservice")
    @ResponseBody
    public String sayHello() {
        return "Greetings from Spring Boot!";
    }
}
```

¹⁵ <http://projects.spring.io/spring-boot/>

¹⁶ <http://tomcat.apache.org/>

¹⁷ <http://www.eclipse.org/jetty/>

¹⁸ <http://undertow.io/>

```

    }
    public static void main(String[] args) throws Exception {
        SpringApplication.run(MyController.class, args);
    }
}

```

The dependency management of the JAR files of the application can be done with Maven or Gradle. Configuration follows the “Convention over Configuration¹⁹” method. An auto-configuration feature scans for included JAR files in the application and provides a default behavior to these Spring modules. This process can also be done by hand to overwrite the default behavior. The application should be implemented with frameworks from the Spring ecosystem to get the most from Spring Boot.

Microservices built and packaged by using Spring Boot have everything that they need to run as 12-factor applications across different deployment environments. Spring Cloud, another set of packages from the Spring ecosystem, provides an additional collection of integrations between third-party technologies and the Spring programming model to support by using Spring Boot applications in cloud environments. For example, Spring Cloud Configuration supports injecting application configuration from external configuration sources in support of immutable application artifacts and Spring Cloud Netflix provides language bindings and libraries to easily build microservice applications by using the popular Netflix OpenSource Software framework.

Dropwizard

Dropwizard also produces stand-alone executable JAR files by using a preselected stack that enhances an embedded Jetty servlet container with extra capabilities.

Wildfly Swarm

Wildfly Swarm²⁰ packages the application and the subset of Wildfly server²¹ the application needs into the executable JAR file. Microservices produced with Wildfly swarm can use a wider set of pre-packaged Java EE technologies, rather than starting from a servlet engine and having to gather and assemble the rest.

WebSphere Liberty

IBM WebSphere Liberty can also produce immutable executable JAR files containing only the technologies the application needs. The packaging tool of Liberty can combine the following elements:

- ▶ A customized (minified) Liberty run time
- ▶ Java EE applications packaged as a WAR file or EAR file (usually only one)
- ▶ Supporting configuration like shared libraries, JNDI values, or data source configurations

Liberty has sensible defaults, requiring configuration only for customized elements with ample support for injected configuration. Liberty is able to directly interpret environment variables, which can eliminate startup scripts that pre-process configuration files to insert environment-specific values. For example, a data source can be defined in `server.xml` as shown in Example 8-3.

Example 8-3 Example 3. Configuring a data store by using environment variables

```

<couchdb id="couchdb" jndiName="couchdb/connector"
    libraryRef="couchdb-lib"

```

¹⁹ https://en.wikipedia.org/wiki/Convention_over_configuration

²⁰ <http://wildfly-swarm.io>

²¹ <http://wildfly.org/>

```
password="${env.COUCHDB_PASSWORD}"  
url="${env.COUCHDB_SERVICE_URL}"  
username="${env.COUCHDB_USER}"/>
```

8.3.3 Containerization

The previous strategies are all running on an operating system that must be installed and configured to work with the executable JAR file. A strategy to avoid the problems resulting from inconsistently configured operating systems is to bring more of this operating system with you. This strategy leads to the situation that a microservice runs on an operating system with the same configuration on every stage.

Containers are lightweight virtualization artifacts that are hosted on an existing operating system. Docker provides an infrastructure capability to build a complete file system that contains all elements (tools, libraries) of the operating system that are needed to run the application, including the application itself. Docker Engine is a lightweight run time that provides both the abstraction and isolation between Docker containers and the host, and the supporting tools to work with hosted containers. For more information about Docker, see this website:

<http://www.docker.com>

Docker containers isolate applications from other applications (residing in peer containers) and also separates applications from the host's operating system. The isolation of these containers and their lightweight attributes make it possible to run more than one Docker container on one operating system (see Figure 8-1). This configuration means that you do not need as many virtualized servers to host your system of microservices.

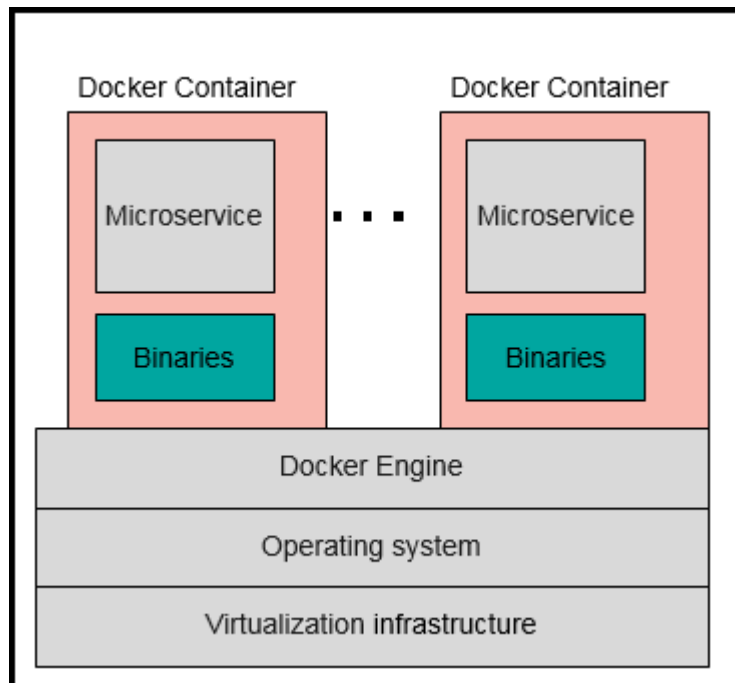


Figure 8-1 Multiple Docker containers on one server

Docker images are immutable artifacts. When applications are built into Docker images, those images can then be used to start containers in different deployment environments, ensuring that the same stack is used across all environments. Using Docker containers makes the exact packaging format (executable JAR, WAR, EAR, ZIP file) irrelevant because the image configuration determines how a container is started. Configuration values for the microservice hosted inside a container can be provided either by using environment variables or mounted volumes.

Data stores and all other backends needed by the microservice should not be hosted in the same container as the microservice itself, as is mentioned in the 12 factors. These backends should be treated as services, which is an essential characteristic of a microservice. Data stores, for example, can also be managed as Docker containers. For failover reasons, you should not host multiple instances of the same microservice on the same host.

8.3.4 Every microservice on its own server

Another alternative to keep the operating system as identical as possible across all stages is to use the same virtualized server as the basis for every stage. To achieve this goal, install the operating system and other tools that you need, and then create a template of that server. Use this template as the basis for all servers that host your system of microservices, which is another form of immutable server.

To manage these servers and server templates, you can use tools provided by your virtualization provider, cloud provider or other third-party options. Some of the third-party options available now include Vagrant²², VMware²³, and VirtualBox²⁴. When paired with software configuration and automation technologies like Salt or Ansible, the virtualization of microservices onto individual servers can become flexible and agile.

This configuration leads mainly to two variants:

- ▶ Having one microservice packaged on a single virtualized server.
- ▶ Having multiple microservice packages on a single virtualized server (see Figure 8-2).

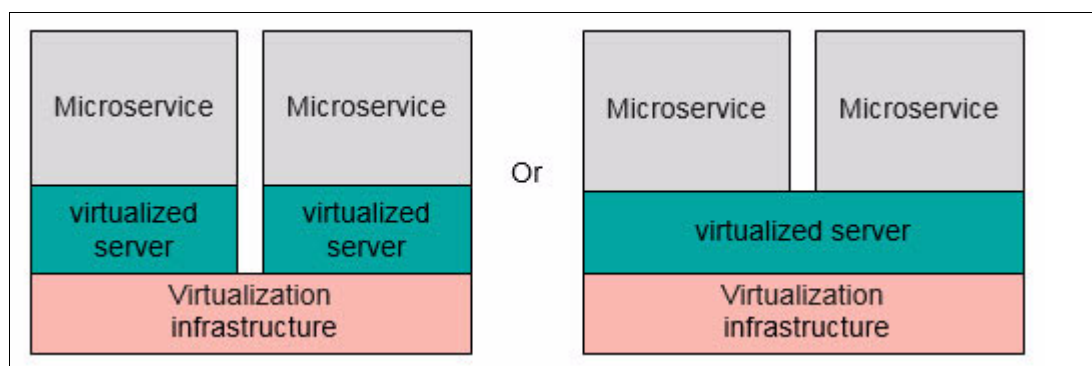


Figure 8-2 Microservices on virtualized servers or microservices together on one server

When deploying only one microservice per virtualized server, you get good isolation from all the other virtualized servers that host the other microservices. The operating system can be configured corresponding to the needs of the specific microservice it is hosting. When errors occur, problem determination is not complicated by having to separate out unrelated information or side effects produced by other applications in the same server. The lack of side effects from other unrelated applications is also good on the performance front. One

²² <http://www.vagrantup.com/>

²³ <http://www.vmware.com>

²⁴ <http://www.virtualbox.org/>

application cannot (as easily) starve other applications of resources when they are isolated from each other in different virtualized servers.

The disadvantage of this alternative is you end up with many virtualized servers (one per microservice instance) that must be managed and coordinated across your infrastructure. The overall resource consumption is also higher because every virtualized server needs its own operating system. Some of the virtualization solutions have optimizations for this problem, which can help to reduce the processor usage caused by multiple operating systems.

Having multiple microservices on one virtualized server should only be done when they have similar resource usage patterns, so that one microservice does not adversely affect, or starve, the others.

The configuration of the operating system must also be compatible with hosting multiple microservices on the same virtualized server. Different configurations must not disturb each other or require conflicting operating system configurations between stages. This pattern would generally reduce the number of virtualized systems that are needed to host the microservice application, at the cost of sharing system configuration. In this setting, you can also use the shared class cache of the JVM, which reduces start time of the microservice.

8.3.5 Aspects of running multiple microservices on one server

The following remarks are suitable for deployments where more than one microservice gets hosted on a server instance, regardless of whether this configuration is done with a container or on a virtualized system.

If you host more than one microservice on your server, then pay attention to the thread pool optimization of your Java run time. The JVM queries the operating system to get the information about the number of CPUs on this server, and uses this information to define a strategy for managing thread pooling. With virtualized hardware, the JVM often gets the wrong information about available resources because every JVM on a server sees the same information about available CPU, even though the amount available to any specific process is considerably less. This configuration can cause unexpected side effects in some thread pool tuning algorithms because the wrong optimization might be triggered because it does not get as many CPUs as expected.

Another aspect that you should consider is how microservices on the same server would share or contend for server resources. Know the resource requirements for each microservice upfront, or gather the data to figure it out, to avoid problems that are related to hosting shared resources. For example, caching inside the memory of a microservice can cause problems as the cache grows. Therefore, generally perform caching on an external system. The external caching server should be treated as an attached service just like a data store.

8.3.6 Preferred practices on packaging

On the first glance, running multiple Docker containers containing microservices on the same host server seems the same as running multiple microservice packages on the same server. But upon further review, there are notable differences between the two deployment patterns. Running multiple microservices on one server without Docker containers can only be done when configurations of the operating system are the same for all microservices. This configuration must be checked in advance. However, Docker provides a much greater level of isolation while making effective use of host resources to the common underlying operating

system. Microservices that require different or conflicting operating system configurations can run happily side by side in Docker containers.

The following is a summary of preferred practices for packaging:

- ▶ Virtualization or cloud services are the preferred basis for infrastructure because dynamic provisioning can help a system of microservices deal with changing capacity requirements.
- ▶ System level configuration should either be as similar as possible across all stages, or should be isolated from the application by using technology like containers.
- ▶ An immutable artifact should be produced that is promoted unchanged between stages.
- ▶ Immutable artifacts for microservices should be stand alone and self-contained. This item could be an executable JAR file or a Docker container.
- ▶ Each immutable artifact should contain only one microservice application and its supporting run time and libraries.
- ▶ Resource requirements should be considered when hosting shared resources to prevent disruption due to resource contention.
- ▶ Shared libraries should be built as part of your application, rather than being shared or used as a dependency provided by the hosting environment.

Figure 8-3 shows a Java EE-based microservice that is packaged together with its Java EE run time running inside a Docker container as a virtualized infrastructure. This is one of the preferred practices for hosting a microservice.

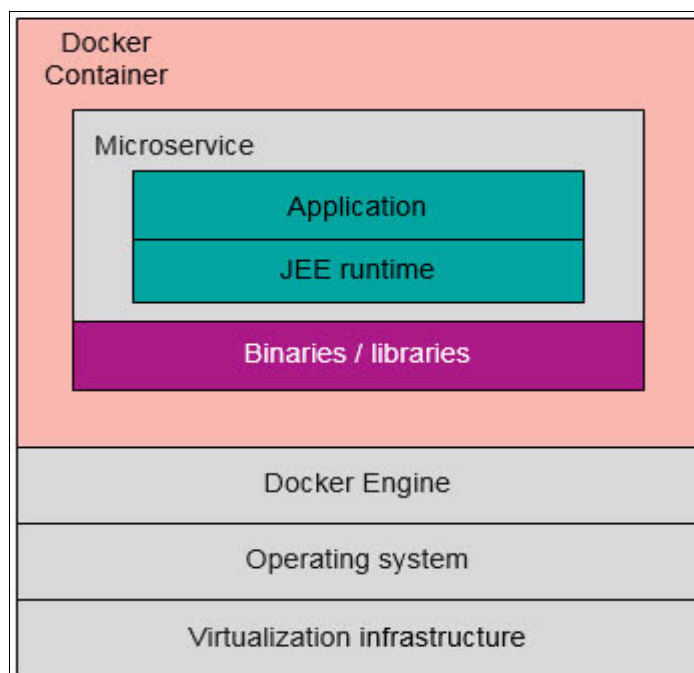


Figure 8-3 Java Platform, Enterprise Edition based microservice in a Docker Container hosted on virtualized infrastructure

8.4 Configuration of the applications across stages

Every microservice should be an immutable deployment artifact. After it is built, it should not be changed for the deployments on different stages. Therefore, environment variables, which can change from stage to stage, need to be injected from outside of the deployment artifact. In addition to using an immutable server for deployments, the application configuration should be immutable as far as possible. Only change values that are necessary on different server (such as port, IP addresses, database users, thread pool size). The remainder of the application configuration should be the same for every stage. The more values that you change, the higher the risk of getting failures on the deployment on the next stage. If values need to be changed on the subsequent stage, then you should test the behavior of your service in the presence to these changes.

Configuring the system of microservices can be done in three different ways:

- ▶ Environment variables
- ▶ Configuration files
- ▶ Configuration systems

The following sections describe these three variants.

8.4.1 Environment variables

One way to get the configuration from outside to your microservice is to use environment variables. This is a common way to pass values to a running process.

Environment variables are character-based key value pairs that are both mapped to the Java data type `java.lang.String`. The way to set these values varies between operating systems. Example 8-4 shows how to set these variables in a scripting environment to start the microservice.

Example 8-4 Setting environment variables in a Linux Bash script

```
#!/bin/bash
export VARIABLE1="Value 1"
export VARIABLE2="Value 2"
```

The Java sample shown in Example 8-5 shows how to parse these environment variables.

Example 8-5 Parsing environment variables

```
Map<String, String> envMap = System.getenv();
for (String envVariable : envMap.keySet()) {
    System.out.format("%s=%s\n",
        envVariable, envMap.get(envVariable));
}
```

Using environment variables is the preferred method compared to passing system properties (Java command line option `-D`) to your microservice because in a Cloud environment, you normally have no access to the command line of your process. Cloud environments normally use environment variables.

Cloud Foundry for example uses an environment variable called `VCAP_SERVICES`. It is a JSON document that contains information about service instances bound to the application. For more information about `VCAP_SERVICES`, see:

<http://docs.cloudfoundry.org/devguide/deploy-apps/environment-variable.html#VCAP-SERVICES>

Example 8-6 shows `VCAP_SERVICES` addressing an IBM Cloudant®²⁵ database.

Example 8-6 VCAP_SERVICES

```
VCAP_SERVICES=
{
  "cloudantNoSQLDB": [
    {
      "name": "myMicroservicesCloudant",
      "label": "cloudantNoSQLDB",
      "plan": "Shared",
      "credentials": {
        "username": "xyz",
        "password": "password",
        "host": "yourhost.yourdomain.com ",
        "port": 443,
        "url": "https://mycloudant.yourhost.yourdomain.com"
      }
    }
  ]
}
```

Connectors available to parse the `VCAP_SERVICES` variable. For example, Spring Boot has a connector to use the `VCAP_SERVICES` in its configuration service. For more information, see this website:

<http://spring.io/blog/2015/04/27/binding-to-data-services-with-spring-boot-in-cloud-foundry>

8.4.2 Configuration files

Another alternative to get configuration values from your environment into Java is to use Java Naming and Directory Interface (JNDI²⁶) which is a Java API to query the directory service. JNDI must be provided by every Java EE compatible server. Elements that are defined in the JNDI namespace can be injected with Java CDI, so accessing these values is easy. To get elements into the JNDI namespace, they must be added to a configuration file.

In Liberty, this process involves the following steps:

1. Activate the JNDI-Feature in the *server.xml*, which is the main configuration file in Liberty, Example 8-7. After this you can define your JNDI variable.

Example 8-7 Activate JNDI feature in Liberty and set JNDI value in server.xml

```
<server>
  <featureManager>
    <feature>jndi-1.0</feature>
```

²⁵ <http://cloudant.com/>

²⁶ http://de.wikipedia.org/wiki/Java_Naming_and_Directory_Interface

```
</featureManager>
<jndiEntry jndiName="myFirstJNDIEntry" value="'myFirstValue'"/>
</server>
```

2. Looking up the value for this JNDI entry can be done in two ways. You can do it on your own as shown in Example 8-8.

Example 8-8 Parsing JNDI value

```
InitialContext ctx = new InitialContext();
String jndiValue = (String) ctx.lookup("myFirstJNDIEntry");
```

Or you can use CDI by using a field-based injection (Example 8-9).

Example 8-9 Field-based injection of JNDI with CDI

```
Public class MyClass {
    @Resource(name="myFirstJNDIEntry")
    private String jndiValue;
    ...
}
```

Using this technique requires that the configuration files must exist on every stage and must have the correct stage values. This configuration can be done with tools from the DevOps universe to ensure that the needed file with the correct values exists on the target stage.

Loading these files into the application within a Liberty `server.xml` can be done with a URL resource that points to where the configuration file is stored in the file system (see Example 8-10).

Example 8-10 JNDI URL Entry in Liberty

```
<server>
  <featureManager>
    <feature>jndi-1.0</feature>
  </featureManager>
  <jndiURLEntry jndiName="myFirstJNDIEntry"
    value="'file:///myDirectory/myConfigFile'" />
</server>
```

In Liberty, you can also use more than one server configuration file. You just have to include other XML files into `server.xml` as shown in Example 8-11.

Example 8-11 Liberty include of configuration files

```
<server>
  <featureManager>
    <feature> . . . </feature>
  </featureManager>
  <include location="mySecondConfigFile.xml"/>
</server>
```

The included file gets parsed and the values are set in the application server. This process gives you the opportunity to separate the configuration files. Configuration files that have values that must be changed on every stage should be included in `server.xml`. The main configuration file (`server.xml`) should only have values that are constant across all stages.

Setting your configuration this way keeps the main configuration file (`server.xml`) the same on every stage.

In a Docker based environment, it is normal to do the configuration with files placed outside the Docker container. The Docker container itself stays unchanged and the configuration file must be provided by DevOps tools.

Spring Boot also offers the opportunity to configure the application from outside. It is possible to use properties files, YAML files, environment variables, and command line arguments. These values get injected by Spring into Spring beans and can be used inside your application. For more information, see the following website:

<http://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-external-config.html>

8.4.3 Configuration systems

The third alternative to configure the system of microservices is to use special servers. These configuration systems are all some kind of key value stores to persist the configuration for the different microservices.

According to the CAP theorem²⁷ from Eric Brewer, you must choose two of the three guarantees between consistency, availability, and partition tolerance. As the microservice system will not work without the configuration system, the configuration system must be partitioned. This limitation means that your configuration system will either be always available with inconsistent values, or have limited availability but with consistent values.

The following are examples of configuration systems used in microservice environments:

- ▶ Apache ZooKeeper
<http://zookeeper.apache.org/>
- ▶ etcd
<http://coreos.com/etcd/>
- ▶ Netflix archaius
<http://github.com/Netflix/archaius>
- ▶ Consul
<http://www.consul.io/>

ZooKeeper and etcd are CP systems with regard to the CAP theorem. Archaius and Consul follow the AP attributes. All of these systems can be used in a Java based application. Which one to use, with regard to the CAP theorem, depends on the requirements of your microservice application.

The advantage of using configuration systems is that configuration data can be changed at run time. This change cannot be easily achieved with environment variables or configuration files. If a bound JNDI value changes, the Liberty server would have to be restarted to get the new values into the application. You should track the changes in the configuration system to be able to reproduce the values later to help you to find the reasons for any failures.

²⁷ http://en.wikipedia.org/wiki/CAP_theorem

8.4.4 Programming considerations

Configuring your microservices should not be done inside the business logic of your source code. It is better to do this configuration in a central place. In most cases, accessing the configuration values is based on technical decisions and is not affected by your business logic.

If you configure your system using CDI, then you must deal with changing values at run time. In such situations, annotate the CDI beans with `@RequestScoped` (or `@Dependent`) to make sure that the values are refreshed on every request. Some of the configuration systems provide callbacks to inform the client when values change. In combination with CDI producers (`@Produces`), you can easily attach your configuration system to your application.

The following examples show a CDI way to set the configuration values in your application. First, you must define a CDI qualifier that can be used to annotate the injection point (`@Configuration`) as shown in Example 8-12.

Example 8-12 CDI qualifier

```
@Qualifier
@Retention(RUNTIME)
@Target({ METHOD, FIELD, PARAMETER, TYPE })
public @interface Configuration {

    // Values without binding
    @Nonbinding
    String value() default "";
}
```

The generic producer class (`MyConfigProducer`) (Example 8-13) checks the name of the attribute inside the qualifier and returns the corresponding value from its configuration cache. The main thing to address is the `InjectionPoint` object that holds the information that is needed by the producer that is set at the injection point in the EJB as shown in Example 8-14 on page 105.

Example 8-13 CDI producer

```
public class MyConfigProducer {
    private Properties environmentProps;
    @Produces
    @Configuration
    public String getConfigValue(InjectionPoint ip) {
        Configuration config = ip.getAnnotated().
            getAnnotation(Configuration.class);
        String configKey = config.value();

        if (configKey.isEmpty()) {
            throw new IllegalArgumentException(
                "Property key missing!");
        }
        String myValue = environmentProps.getProperty(configKey);
        if (myValue == null) {
            throw new RuntimeException("Property value for key " +
                configKey + " could not be found!");
        }
        return myValue;
    }
}
```

```
}  
.  
.  
.  
}
```

The default CDI scope for an EJB is `@RequestScoped` and the default CDI scope when it is not explicit set is `@Default`. The producer injects the value on every request that calls the EJB. Other CDI scopes can also be used depending on your requirements (Example 8-14).

Example 8-14 EJB with injected configuration value

```
@Stateless  
public class MyEJB {  
    @Inject  
    @Configuration("myConfigValue")  
    private String myConfigValue;  
    . . .  
}
```

8.4.5 Preferred practices on configuration

In summary, the following are the preferred practices when doing configuration:

- ▶ Configuration should live outside of the immutable deployment artifact and can be supplied by three variations of configuration (environment variables, configuration files, and configuration systems). The choice depends more on the runtime environment that your system of microservices is being hosted in.
- ▶ Dynamic configuration values should be managed with configuration systems, but your application must handle the changing of configuration values.
- ▶ Keep the system as unchanged as you can. This guideline is also true for configuration values otherwise to minimize the risk of unexpected errors.
- ▶ Change only configuration values that must be changed and use a defined way to manage those changes (for example, track these changes).
- ▶ The business logic of the microservice should not deal with the configuration values. This tracking should be done in a central helper class or package.



Management and Operations

Running a system of microservices has some aspects that differ from running a monolith application. With microservices, a single application becomes a distributed system that is made up of a number of microservices that communicate with each other. Therefore, you must manage a highly distributed system and worry about the communication path between each service.

The number of running services is normally higher than the services needed for a monolith application. When load characteristics vary, a microservice application should react by adjusting the number of running services. Therefore, you must be able to automatically deploy new and remove existing service instances to ensure that overall application is running at optimum state.

In every production system, it is necessary to do health checks to get information about the state of the running applications. This requirement is also true for a system of microservices where you must worry about the health of each service instance.

This chapter covers the following topics:

- ▶ Metrics and health checks
- ▶ Logging
- ▶ Templating

9.1 Metrics and health checks

Metrics and health checks are necessary to get information about the state of a running service. As mentioned in 8.3, “Packaging options” on page 92 each Java EE-based microservice has its own run time packaged together with the microservice code itself. These Java based runtimes have tools to help the administrators to check the health of the running microservice.

9.1.1 Dynamic provisioning

With dynamic provisioning, the aim is to automatically keep the system of microservices in its optimum state. This state is where the system is properly balanced to match the load demands of the application users. It can be reached with enough service instances being available to manage throughput and keep application latency at a minimum. For example, an instance of a microservice becoming overloaded can be compensated by scaling up and starting another instance of a service. This change should be done automatically without the need for administrator intervention. Similarly, unused microservice instances with low load can be stopped to return the unused resources to your virtualization system.

However, in some situations it is necessary to inform the administration when some threshold is reached so they can take action to prevent failures in the system. However, this should be an exceptional case.

Service discovery as described in Chapter 3, “Locating services” on page 25 is a key service that is used by microservices. The service discovery service tracks available microservice instances and the current health state. A new instance of a service should start to take requests to reduce load from the other instances. With this information, load balancing can be dynamically performed to distribute load across available microservice instances for optimal performance. A microservice that is shutting down should complete running requests before it stops, and should not receive any further requests to process.

A system of microservices needs a high degree of automation to achieve this behavior. You can use every tool of your administration infrastructure to help you. But a packaged microservice as described in 8.3, “Packaging options” on page 92 gives you an alternative for this task. The runtime environment in the microservice package has some features to support your automation. Using Liberty as a runtime environment, for example, has some features that offer you support in the areas of auto scaling and dynamic routing.

Auto scaling provides an autonomic scaling capability of Liberty servers. The auto scaling function is enabled by two Liberty features: Scaling controller and scaling member. Auto scaling dynamically adjusts the number of Java virtual machines (JVMs) used to service your workload. This feature provides operational agility and decreases administrative requirements to enhance the resiliency of your middleware environment. The conditions for auto scaling are defined by scaling policies. These conditions include the minimum or maximum number of server instances and the threshold values for each of the server resources.

A common pattern with microservices operation is to keep a specific number of microservice instances running at all times, or more often a minimum number and some level of permissible scaling up. A microservice application with these settings is running within its designed bounds and can scale up an acceptable amount for unexpected loads. With the auto scaling feature of Liberty, you can define the minimum number of service instances that should be running for a healthy system. You can then have system management keep that number alive. Another Liberty feature to check for the health of a service is described in 9.1.2, “Health check” on page 111.

The following metrics that can be used to do auto scaling.

The scaling member monitors the use of the following resources within the server process:

- ▶ CPU
- ▶ Heap
- ▶ Memory

The scaling member monitors the use of the following resources at the host level:

- ▶ CPU
- ▶ Memory

For more information, see “Setting up auto scaling for Liberty collectives” at:

http://www.ibm.com/support/knowledgecenter/SSAW57_9.0.0/com.ibm.websphere.wlp.nd.doc/ae/twlp_wve_autoscaling.html

When you start and stop instances of microservices, your service discovery must also be informed. One easy way to do this is to inform the HTTP server in front of your system of microservices about started or stopped microservices. Liberty has a feature that can be combined with the auto scaling feature. The Dynamic Routing feature enables routing of HTTP requests to members of Liberty collectives without having to regenerate the WebSphere plug-in configuration file when the environment changes. When servers, cluster members, applications, or virtual hosts are added, removed, started, stopped, or modified, the new information is dynamically delivered to the WebSphere plug-in. Requests are routed based on up-to-date information. The feature provides the Dynamic Routing service, which dynamically retrieves routing information from the collective repository and delivers this information to the WebSphere plug-in.

Liberty servers that take part in this functionality must be joined in a collective. The set of Liberty servers in a single management domain is called a collective. A collective consists of at least one server with the `collectiveController-1.0` feature enabled that is called a collective controller. Optionally, a collective can have many servers with the `collectiveMember-1.0` feature enabled that are called collective members and a collective can be configured to have many collective controllers.

Figure 9-1 shows an example of a Liberty collective.

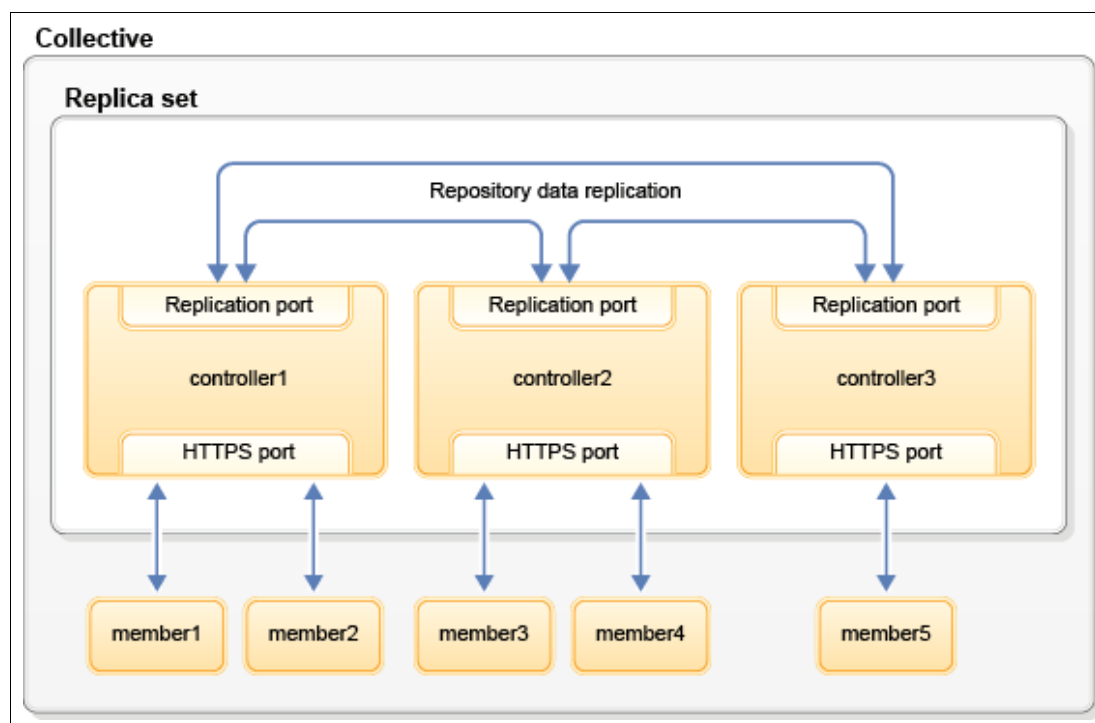


Figure 9-1 Liberty collective architecture¹

A sample for a topology showing the dynamic routing feature is shown in Figure 9-2.

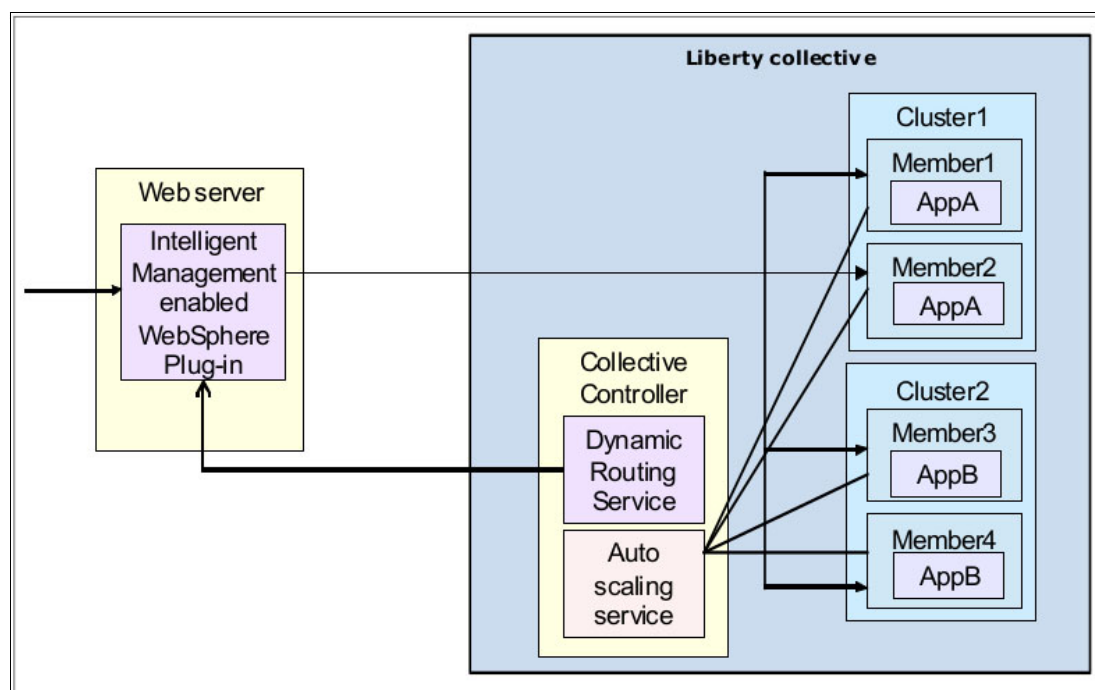


Figure 9-2 Auto scaling and dynamic routing topology

¹ http://www.ibm.com/support/knowledgecenter/en/SSAW57_9.0.0/com.ibm.websphere.wlp.nd.doc/ae/cwlp_coll_ective_arch.html

9.1.2 Health check

The instances of the microservice application experiencing problems must be recognized. This function helps to keep the whole system in its optimum state. To get more information out of your system of microservices, Liberty has another feature to support this function. With the health management feature in Liberty, you can take a policy driven approach to monitoring the application server environment and take action when unhealthy criteria is discovered. You can define the health policies, which include the health conditions to be monitored in your environment and the health actions to take if these conditions are met.

Health management in a Liberty collective can prevent the disruption of service by detecting common problems and generating diagnostic actions based on configured health policies. The health management functions are enabled by two Liberty features: Health manager and health analyzer.

The following predefined health conditions can be used:

- ▶ Excessive request time out condition
- ▶ Excessive response time condition
- ▶ Memory condition: Excessive memory usage
- ▶ Memory condition: Memory leak

When one of these health conditions is raised, a predefined action is executed. The following actions are available:

- ▶ Restart server
- ▶ Take thread memory dumps
- ▶ Take JVM heap memory dumps
- ▶ Enter server into maintenance mode
- ▶ Exit server out of maintenance mode

Some of these actions produce more information (thread and heap memory dumps) to get a better understanding of the problem. A server in maintenance mode is still running, but does not accept new requests.

To get more information about internals of the running server, the monitor feature of Liberty can help you. This feature enables the Performance Monitoring Infrastructure (PMI) for other features that the server is running. The values collected in the MXBeans can be seen in the AdminCenter of the Liberty server (feature `adminCenter-1.0` must be activated).

Other tools already in the infrastructure of your production system (for example Nagios²) can also be used to help administrators keep the system of microservices running.

9.2 Logging

Logging is essential for applications. The same is true for a system of microservices. But there are some special points of interest.

In a system of microservices, a business transaction can span multiple microservices. If your microservices use a message-oriented middleware, this can also be part of your business transaction. Therefore, you need something to track the request from entering your system until the response leaves the system. A good way to track the request is to use a correlation ID (see 5.2.5, “Java Message Service API” on page 62). The same correlation ID you use for your message-oriented middleware can be used for your microservice requests.

² <http://www.nagios.org/>

You must create a correlation ID on the first request that enters your system of microservices, then send the same correlation ID to other microservices when you call them. The other services only have to check whether there is a correlation ID and propagate it in their calls. The correlation ID can be placed into the HTTP header using a special field (X-header fields, for example) to hold the value. The handling of the header fields can be done with a REST filter to keep this logic in a central place.

Using a correlation ID provides these benefits:

- ▶ It is easier to find cascading errors.
- ▶ You can track user requests to get information about typical behavior of the customer.
- ▶ You can find unused or seldom used microservices that can be eliminated, or find heavily used microservices where you can get more business value if you extend them.

There are some benefits with your logging framework to logging the value of the correlation ID in your log files. Some of these frameworks use logging context. SLF4J³ or Log4j 2⁴ implements a Mapped Diagnostic Context (MDC) or a Nested Diagnostic Context (NDC) that you can use to put the correlation ID into it. Therefore, in every log statement written to the log files, the NDC or MDC is included. Another benefit that you get from the NDC/MDC is that if you are using multiple threads to get the request done, then each of these threads logs the same NDC/MDC.

The layout of the log format should be defined in a central manner, so it is easier to aggregate the logs produced in the different microservices. The log levels (for example, FATAL, ERROR, WARN, INFO, DEBUG, TRACE, ALL) should be clearly defined and used corresponding in every microservice. Otherwise, you might get misunderstandings between the microservice teams.

In a system of microservice with dynamic provisioning, you must ensure that the server that hosted the microservices and the log files get deleted after the shutdown of the system. In such cases, it is better to use a logging infrastructure that is always running and gathering the log output from your microservices. This logging system can also help you in situations where log files are distributed over different systems. It can be time consuming to search in log files on different servers. This limitation can also be complicated in a production environment where you normally have no access rights or minimal rights on the systems.

Therefore, the preferred approach for a logging infrastructure in a microservice application is the ELK stack or Elastic Stack⁵, which is a combination of three tools:

- ▶ Elasticsearch (search and analytics engine)
- ▶ Logstash (data collection and transportation pipeline)
- ▶ Kibana (visualizing collected data)

If your system of microservices uses Liberty as the Java EE provider, then there is a feature to use the Elastic Stack: `logstashCollector-1.0`. Use the Logstash collector feature in Liberty to collect log and other events from your Liberty servers and send them to a remote Logstash server.

³ <http://www.slf4j.org/>

⁴ <http://logging.apache.org/log4j/2.x/>

⁵ <http://www.elastic.co/products>

Example 9-1 shows the usage of `logstashCollector` in Liberty.

Example 9-1 Configuring `logstashCollector` in Liberty⁶

```
<featureManager>
  <feature>logstashCollector-1.0</feature>
</featureManager>
<keyStore id="defaultKeyStore" password="Liberty" />
<keyStore id="defaultTrustStore" location="trust.jks" password="Liberty" />
<ssl id="mySSLConfig" trustStoreRef="defaultTrustStore"
keyStoreRef="defaultKeyStore" />
<logstashCollector
  source="message,trace,garbageCollection,ffdc,accessLog"
  hostName="localhost"
  port="5043"
  sslRef="mySSLConfig"
/>
```

Splunk⁷ is a commercial product that can also be used to gather and analyze your log data.

9.3 Templating

From a management point of view, it can be valuable to do templating for your development team, which is implementing the system of microservices. Templating in this sense is defining a skeleton of code fragments that can be used to generate the needed parts of a microservice.

A microservice template typically includes these common capabilities:

- ▶ Communication with Service Discovery to register a service instance
- ▶ Communication with other microservices
- ▶ Communication with a messaging system
- ▶ Logging
- ▶ Security

You get these benefits from a template having all these capabilities:

- ▶ Quicker start for the development teams
- ▶ All teams use the same cross functional services

The following are available generators:

- ▶ WildFly Swarm Project Generator⁸
- ▶ Spring Initializr⁹
- ▶ Liberty app accelerator¹⁰

If all teams use Java to implement the microservices, and do not have a polyglot programming environment, then you must only define one template that can be evolved occasionally to best fit your requirements.

⁶ http://www.ibm.com/support/knowledgecenter/SSAW57_9.0.0/com.ibm.websphere.wlp.nd.doc/ae/twlp_analytics_logstash.html

⁷ <http://www.splunk.com/>

⁸ <http://wildfly-swarm.io/generator/>

⁹ <http://start.spring.io/>

¹⁰ <http://liberty-app-accelerator.wasdev.developer.ibm.com/start/#1>

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this book.

IBM Redbooks

The following IBM Redbooks publications provide additional information about the topic in this document. Note that some publications referenced in this list might be available in softcopy only.

- ▶ *IBM WebSphere Application Server V8.5 Administration and Configuration Guide for Liberty Profile*, SG24-8170-00

You can search for, view, download or order these documents and other Redbooks, Redpapers, Web Docs, draft and additional materials, at the following website:

ibm.com/redbooks

Other publications

These publications are also relevant as further information sources:

- ▶ “Understanding the Differences between AMQP & JMS”, Marc Richards
- ▶ “Building Microservices”, Sam Newman, February 2015
- ▶ “Release It!: Design and Deploy Production-Ready Software”, Michael T. Nygard, 2007
- ▶ “Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation”, Jez Humble, David Farley, 2011
- ▶ “Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions”, Gregor Hohpe and Booby Woolf, Addison-Wesley Professional

Online resources

These websites are also relevant as further information sources:

- ▶ “1.2.2 Robustness Principle”, RFC 1122
<https://tools.ietf.org/html/rfc1122#page-12>
- ▶ Apache Avro
<https://avro.apache.org/>
- ▶ Amalgam8
<https://github.com/amalgam8/>
- ▶ Apache Thrift
<https://thrift.apache.org/>

- ▶ Blog post by Troy Hunt
<https://www.troyhunt.com/your-api-versioning-is-wrong-which-is/>
- ▶ “Chapter 5: Representational State Transfer (REST)”, Fielding, Roy Thomas (2000)
http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
- ▶ Cloud Native Computing Foundation (“CNCF”) Charter
<https://cncf.io/about/charter>
- ▶ Consul
<https://www.consul.io/>
- ▶ Eureka
<https://github.com/Netflix/eureka>
- ▶ Fallacies of Distributed Computing Explained
<http://www.rgoarchitects.com/Files/fallacies.pdf>
- ▶ Game On! Player API
<https://game-on.org/swagger/#!/players>
- ▶ “Microservices: A definition of this new architectural term”, Martin Fowler
<http://martinfowler.com/articles/microservices.html>
- ▶ Oracle Java Platform, Enterprise Edition (Java EE) 7
<http://docs.oracle.com/javaee/7/>
- ▶ Protocol Buffers:
<https://developers.google.com/protocol-buffers/>
- ▶ “The Twelve-Factor App”, Adam Wiggins, 2012
<http://12factor.net>

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

Redbooks

Microservices Best Practices for Java

(0.2"spine)
0.17"<->0.473"
90<->249 pages



SG24-8357-00

ISBN 0738442275

Printed in U.S.A.

Get connected

