



# Microservices

FLEXIBLE SOFTWARE ARCHITECTURE

EBERHARD WOLFF

FREE SAMPLE CHAPTER

SHARE WITH OTHERS



# Microservices

*This page intentionally left blank*



# Microservices

Flexible Software Architecture

Eberhard Wolff

◆ Addison-Wesley

Boston • Columbus • Indianapolis • New York • San Francisco  
Amsterdam • Cape Town • Dubai • London • Madrid • Milan  
Munich • Paris • Montreal • Toronto • Delhi • Mexico City  
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at [corpsales@pearsoned.com](mailto:corpsales@pearsoned.com) or (800) 382-3419.

For government sales inquiries, please contact [governmentsales@pearsoned.com](mailto:governmentsales@pearsoned.com).

For questions about sales outside the United States, please contact [intlcs@pearson.com](mailto:intlcs@pearson.com).

Visit us on the Web: [informit.com/aw](http://informit.com/aw)

Library of Congress Control Number: 2016952028

Copyright © 2017 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit [www.pearsoned.com/permissions/](http://www.pearsoned.com/permissions/).

ISBN-13: 978-0-134-60241-7

ISBN-10: 0-134-60241-2

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.

First printing: October 2016

**Editor-in-Chief**

Mark Taub

**Acquisitions Editor**

Chris Guzikowski

**Development Editor**

Chris Zahn

**Managing Editor**

Sandra Schroeder

**Project Editor**

Lori Lyons

**Production Manager**

Dhayanidhi

**Copy Editor**

Warren Hapke

**Indexer**

Erika Millen

**Proofreader**

Sudhakaran

**Editorial Assistant**

Olivia Basegio

**Cover Designer**

Chuti Prasertsith

**Compositor**

codeMantra

*To my family and friends for their support.*

*And to the computing community for all the fun it  
has provided to me.*

*This page intentionally left blank*

# Contents at a Glance

Preface .....	xxv
<b>Part I: Motivation and Basics .....</b>	<b>1</b>
Chapter 1: Preliminaries .....	3
Chapter 2: Microservice Scenarios .....	11
<b>Part II: Microservices: What, Why, and Why Not? .....</b>	<b>25</b>
Chapter 3: What Are Microservices? .....	27
Chapter 4: Reasons for Using Microservices .....	55
Chapter 5: Challenges .....	69
Chapter 6: Microservices and SOA .....	81
<b>Part III: Implementing Microservices .....</b>	<b>95</b>
Chapter 7: Architecture of Microservice-Based Systems .....	99
Chapter 8: Integration and Communication .....	163
Chapter 9: Architecture of Individual Microservices .....	193
Chapter 10: Testing Microservices and Microservice-Based Systems .....	213
Chapter 11: Operations and Continuous Delivery of Microservices .....	237
Chapter 12: Organizational Effects of a Microservices-Based Architecture .....	269
<b>Part IV: Technologies .....</b>	<b>301</b>
Chapter 13: Example of a Microservices-Based Architecture .....	303
Chapter 14: Technologies for Nanoservices .....	343
Chapter 15: Getting Started with Microservices .....	369
Index .....	375



*This page intentionally left blank*

# Contents

Preface .....	xxv
Acknowledgments .....	xxxii
About the Author .....	xxxiii
<b>Part I: Motivation and Basics .....</b>	<b>1</b>
<b>Chapter 1: Preliminaries .....</b>	<b>3</b>
1.1 Overview of Microservice .....	3
Microservice: Preliminary Definition .....	3
Deployment Monoliths .....	4
1.2 Why Microservices? .....	4
Strong Modularization .....	5
Easy Replaceability .....	5
Sustainable Development .....	6
Further Development of Legacy Applications .....	6
Time-to-Market .....	6
Independent Scaling .....	7
Free Choice of Technologies .....	7
Continuous Delivery .....	7
1.3 Challenges .....	8
1.4 Conclusion .....	9
<b>Chapter 2: Microservice Scenarios .....</b>	<b>11</b>
2.1 Modernizing an E-Commerce Legacy Application .....	11
Scenario .....	11
Reasons to Use Microservices .....	12
Slow Continuous Delivery Pipeline .....	12
Parallel Work Is Complicated .....	12
Bottleneck During Testing .....	13
Approach .....	14
Challenges .....	14

Entire Migration Lengthy	15
Testing Remains a Challenge	15
Current Status of Migration	15
Creating Teams	16
Advantages	17
Conclusion	17
Rapid and Independent Development of New Features	17
Influence on the Organization	18
Amazon Has Been Doing It for a Long Time	18
2.2 Developing a New Signaling System	19
Scenario	19
Reasons to Use Microservices	20
Distributed System	20
Technology Stack per Team	21
Integration of Other Systems	21
Challenges	21
High Technological Complexity	21
Advantages	22
Verdict	22
2.3 Conclusion	23
Essential Points	23

## **Part II: Microservices: What, Why, and Why Not? . . . . . 25**

### **Chapter 3: What Are Microservices? . . . . . 27**

3.1 Size of a Microservice	27
Modularization	28
Distributed Communication	28
Sustainable Architecture	29
Refactoring	29
Team Size	30
Infrastructure	30
Replaceability	31
Transactions and Consistency	31
Consistency	32
Compensation Transactions across Microservices	32
Summary	33

3.2 Conway's Law	35
Reasons for the Law	36
The Law as Limitation	36
The Law as Enabler	38
The Law and Microservices	39
3.3 Domain-Driven Design and Bounded Context	40
Ubiquitous Language	41
Building Blocks	41
Bounded Context	42
Collaboration between <i>Bounded Contexts</i>	44
Bounded Context and Microservices	45
Large-Scale Structure	46
3.4 Why You Should Avoid a Canonical Data Model (Stefan Tilkov)	47
3.5 Microservices with a UI?	50
Technical Alternatives	50
Self-Contained System	51
3.6 Conclusion	52
Essential Points	53
<b>Chapter 4: Reasons for Using Microservices</b>	<b>55</b>
4.1 Technical Benefits	55
Replacing Microservices	56
Sustainable Software Development	57
Handling Legacy	57
Continuous Delivery	59
Scaling	61
Robustness	61
Free Technology Choice	62
Independence	63
4.2 Organizational Benefits	63
Smaller Projects	65
4.3 Benefits from a Business Perspective	65
Parallel Work on Stories	65
4.4 Conclusion	67
Essential Points	68

<b>Chapter 5: Challenges</b> .....	<b>69</b>
5.1 Technical Challenges .....	69
Code Dependencies .....	71
Unreliable Communication .....	73
Technology Pluralism .....	73
5.2 Architecture .....	74
Architecture = Organization .....	74
Architecture and Requirements .....	74
Refactoring .....	75
Agile Architecture .....	75
Summary .....	76
5.3 Infrastructure and Operations .....	76
Continuous Delivery Pipelines .....	77
Monitoring .....	77
Version Control .....	77
5.4 Conclusion .....	78
Essential Points .....	78
<b>Chapter 6: Microservices and SOA</b> .....	<b>81</b>
6.1 What Is SOA? .....	81
Introducing SOA .....	84
Services in an SOA .....	84
Interfaces and Versioning .....	85
External Interfaces .....	85
Interfaces Enforce a Coordination of Deployments .....	85
Coordination and Orchestration .....	86
Technologies .....	86
6.2 Differences between SOA and Microservices .....	87
Communication .....	87
Orchestration .....	87
Flexibility .....	87
Microservices: Project Level .....	88
Synergies .....	91
6.3 Conclusion .....	91
Essential Points .....	92

<b>Part III: Implementing Microservices</b> .....	<b>95</b>
<b>Chapter 7: Architecture of Microservice-Based Systems</b> .....	<b>99</b>
7.1 Domain Architecture .....	100
Strategic Design and Domain-Driven Design .....	100
Example Otto Shop .....	101
Managing Dependencies .....	101
Unintended Domain-Based Dependencies .....	102
Cyclic Dependencies .....	103
7.2 Architecture Management .....	104
Tools for Architecture Management .....	104
Cycle-Free Software .....	104
Microservices and Architecture Management .....	106
Tools .....	107
Is Architecture Management Important? .....	107
Context Map .....	108
7.3 Techniques to Adjust the Architecture .....	110
Where Does Bad Architecture Come From? .....	110
Changes in Microservices .....	111
Changes to the Overall Architecture .....	111
Shared Libraries .....	112
Transfer Code .....	113
Reuse or Redundancy? .....	114
Shared Service .....	115
Spawn a New Microservice .....	116
Rewriting .....	117
A Growing Number of Microservices .....	117
Microservice-Based Systems Are Hard to Modify .....	117
7.4 Growing Microservice-Based Systems .....	118
Planning Architecture? .....	118
Start Big .....	119
Start Small? .....	120
Limits of Technology .....	121
Replaceability as a Quality Criterion .....	121
The Gravity of Monoliths .....	121

Keep Splitting	122
Global Architecture?	122
7.5 Don't Miss the Exit Point or How to Avoid the Erosion of a Microservice (Lars Gentsch)	122
Incorporation of New Functionality	123
What Is Happening to the Microservice Here?	123
Criteria Arguing for a New Microservice Instead of Extending an Existing One	124
How to Recognize Whether the Creation of a New Microservice Should Have Occurred Already	125
Conclusion	125
7.6 Microservices and Legacy Applications	126
Breaking Up Code?	126
Supplementing Legacy Applications	127
Enterprise Integration Patterns	127
Limiting Integration	129
Advantages	129
Integration via UI and Data Replication	129
Content Management Systems	130
Conclusion	130
No Big Bang	131
Legacy = Infrastructure	131
Other Qualities	132
7.7 Hidden Dependencies (Oliver Wehrens)	132
The Database	133
7.8 Event-Driven Architecture	134
7.9 Technical Architecture	136
Technical Decisions for the Entire System	136
Sidecar	137
7.10 Configuration and Coordination	139
Consistency as Problem	139
Immutable Server	140
Alternative: Installation Tools	140
7.11 Service Discovery	141
Service Discovery = Configuration?	141
Technologies	142

7.12 Load Balancing	144
REST/HTTP	144
Central Load Balancer	145
A Load Balancer per Microservice	145
Technologies	145
Service Discovery	146
Client-Based Load Balancing	147
Load Balancing and Architecture	148
7.13 Scalability	148
Scaling, Microservices, and Load Balancing	149
Dynamic Scaling	149
Microservices: Advantages for Scaling	150
Sharding	150
Scalability, Throughput, and Response Times	151
7.14 Security	151
Security and Microservices	152
OAuth2	152
Possible Authorization Grants	153
JSON Web Token (JWT)	154
OAuth2, JWT, and Microservices	155
Technologies	155
Additional Security Measures	156
Hashicorp Vault	157
Additional Security Goals	158
7.15 Documentation and Metadata	159
Outdated Documentation	160
Access to Documentation	160
7.16 Conclusion	161
Essential Points	162
<b>Chapter 8: Integration and Communication</b>	<b>163</b>
8.1 Web and UI	164
Multiple Single-Page-Apps	164
SPA per Microservice	165
Asset Server for Uniformity	166
A Single-Page App for All Microservices	167
HTML Applications	168



ROCA .....	168
Easy Routing .....	169
Arrange HTML with JavaScript .....	170
Front-End Server .....	171
Mobile Clients and Rich Clients .....	172
Organizational Level .....	173
Back-End for Each Front-End .....	174
8.2 REST .....	175
Cache and Load Balancer .....	176
HATEOAS .....	177
HAL .....	177
XML .....	177
HTML .....	178
JSON .....	178
Protocol Buffer .....	178
RESTful HTTP Is Synchronous .....	179
8.3 SOAP and RPC .....	179
Flexible Transport .....	179
Thrift .....	180
8.4 Messaging .....	180
Messages and Transactions .....	181
Messaging Technology .....	182
8.5 Data Replication .....	184
Replication .....	185
Problems: Redundancy and Consistency .....	185
Implementation .....	186
Batch .....	186
Event .....	186
8.6 Interfaces: Internal and External .....	187
External Interfaces .....	188
Separating Interfaces .....	188
Implementing External Interfaces .....	188
Semantic Versioning .....	189
Postel's Law or the Robustness Principle .....	189
8.7 Conclusion .....	190
Client .....	190
Logic Layer .....	191

Data Replication .....	191
Interfaces and Versions .....	192
Essential Points .....	192
<b>Chapter 9: Architecture of Individual Microservices .....</b>	<b>193</b>
9.1 Domain Architecture .....	193
Cohesion .....	194
Encapsulation .....	194
Domain-Driven Design .....	194
Transactions .....	194
9.2 CQRS .....	195
CQRS .....	195
Microservices and CQRS .....	196
Advantages .....	196
Challenges .....	197
9.3 Event Sourcing .....	197
9.4 Hexagonal Architecture .....	199
Hexagons or Layers? .....	200
Hexagonal Architectures and Microservices .....	201
An Example .....	201
9.5 Resilience and Stability .....	203
Timeout .....	203
Circuit Breaker .....	203
Bulkhead .....	204
Steady State .....	205
Fail Fast .....	205
Handshaking .....	205
Test Harness .....	206
Uncoupling via Middleware .....	206
Stability and Microservices .....	206
Resilience and Reactive .....	207
Hystrix .....	207
9.6 Technical Architecture .....	208
Process Engines .....	208
Statelessness .....	209
Reactive .....	209
Microservices without Reactive? .....	210

9.7 Conclusion .....	211
Essential Points .....	212
<b>Chapter 10: Testing Microservices and Microservice-Based Systems .....</b>	<b>213</b>
10.1 Why Tests? .....	213
Tests Minimize Expenditure .....	214
Tests = Documentation .....	214
Test-Driven Development .....	215
10.2 How to Test? .....	215
Unit Tests .....	215
Integration Tests .....	216
UI Tests .....	216
Manual Tests .....	217
Load Tests .....	217
Test Pyramid .....	217
Continuous Delivery Pipeline .....	220
10.3 Mitigate Risks at Deployment .....	220
10.4 Testing the Overall System .....	222
Shared Integration Tests .....	223
Avoiding Integration Tests of the Overall System .....	224
10.5 Testing Legacy Applications and Microservices .....	225
Relocating Tests of the Legacy Application .....	225
Integration Test: Legacy Application and Microservices ...	226
10.6 Testing Individual Microservices .....	227
Reference Environment .....	228
Stubs .....	228
10.7 Consumer-Driven Contract Tests .....	230
Components of the Contract .....	230
Contracts .....	230
Implementation .....	231
Tools .....	232
10.8 Testing Technical Standards .....	233
10.9 Conclusion .....	235
Essential Points .....	236
<b>Chapter 11: Operations and Continuous Delivery of Microservices .....</b>	<b>237</b>
11.1 Challenges Associated with the Operation of Microservices ...	237
Numerous Artifacts .....	238

Delegate into Teams .....	238
Unify Tools .....	239
Specify Behavior .....	239
Micro and Macro Architecture .....	239
Templates .....	240
11.2 Logging .....	241
Logging for Microservices .....	241
Technologies for Logging via the Network .....	242
ELK for Centralized Logging .....	242
Scaling ELK .....	243
Graylog .....	244
Splunk .....	244
Stakeholders for Logs .....	245
Correlation IDs .....	245
Zipkin: Distributed Tracing .....	245
11.3 Monitoring .....	246
Basic Information .....	247
Additional Metrics .....	248
Stakeholders .....	248
Correlate with Events .....	249
Monitoring = Tests? .....	249
Dynamic Environment .....	250
Concrete Technologies .....	250
Enabling Monitoring in Microservices .....	252
Metrics .....	252
StatsD .....	252
collectd .....	252
Technology Stack for Monitoring .....	252
Effects on the Individual Microservice .....	253
11.4 Deployment .....	254
Deployment Automation .....	254
Installation and Configuration .....	256
Risks Associated with Microservice Deployments .....	256
Deployment Strategies .....	256
11.5 Combined or Separate Deployment? (Jörg Müller) .....	258
11.6 Control .....	259
11.7 Infrastructure .....	260

- Virtualization or Cloud ..... 261
- Docker ..... 261
- Docker Container versus Virtualization ..... 263
- Communication between Docker Containers ..... 263
- Docker Registry ..... 264
- Docker and Microservices ..... 264
- Docker and Servers ..... 264
- PaaS ..... 266
- 11.8 Conclusion ..... 266
  - Essential Points ..... 268

**Chapter 12: Organizational Effects of a Microservices-Based Architecture ..... 269**

- 12.1 Organizational Benefits of Microservices ..... 270
  - Technical Independence ..... 270
  - Separate Deployment ..... 270
  - Separate Requirement Streams ..... 271
  - Three Levels of Independence ..... 271
- 12.2 An Alternative Approach to Conway’s Law ..... 273
  - The Challenges Associated with Conway’s Law ..... 273
  - Collective Code Ownership ..... 274
  - Advantages of Collective Code Ownership ..... 274
  - Disadvantages of Collective Code Ownership ..... 275
  - Pull Requests for Coordination ..... 276
- 12.3 Micro and Macro Architecture ..... 277
  - Decision = Responsibility ..... 277
  - Who Creates the Macro Architecture? ..... 278
  - Extent of the Macro Architecture ..... 279
  - Technology: Macro/Micro Architecture ..... 280
  - Operations ..... 281
  - Domain Architecture ..... 282
  - Tests ..... 282
- 12.4 Technical Leadership ..... 284
  - Developer Anarchy ..... 284
- 12.5 DevOps ..... 285
  - DevOps and Microservices ..... 285
  - Do Microservices Necessitate DevOps? ..... 286

12.6	When Microservices Meet Classical IT Organizations (Alexander Heusingfeld)	287
	Pets versus Cattle	287
	Us versus Them	288
	Development versus Test versus Operations: Change of Perspective	288
	For Operations There Is Never an “Entirely Green Field”	289
	Conclusion	290
12.7	Interface to the Customer	290
	Architecture Leads to Departments	291
12.8	Reusable Code	292
	Client Libraries	292
	Reuse Anyway?	292
	Reuse as Open Source	293
12.9	Microservices without Changing the Organization?	295
	Microservices without Changing the Organization	295
	Evaluation	296
	Departments	296
	Operations	296
	Architecture	297
12.10	Conclusion	297
	Essential Points	299
<b>Part IV: Technologies</b>		<b>301</b>
<b>Chapter 13: Example of a Microservices-Based Architecture</b>		<b>303</b>
13.1	Domain Architecture	304
	Separate Data Storages	304
	Lots of Communication	305
	Bounded Context	305
	Don’t Modularize Microservices by Data!	306
13.2	Basic Technologies	306
	HSQL Database	307
	Spring Data REST	307
	Spring Boot	307
	Spring Cloud	308
	Spring Cloud Netflix	310

13.3	Build	311
13.4	Deployment Using Docker	313
13.5	Vagrant	314
	Networking in the Example Application	317
13.6	Docker Machine	320
13.7	Docker Compose	321
13.8	Service Discovery	324
	Eureka Client	324
	Configuration	325
	Eureka Server	326
13.9	Communication	327
	Zuul: Routing	327
13.10	Resilience	329
	Circuit Breaker	329
	Hystrix with Annotations	330
	Monitoring with the Hystrix Dashboard	331
	Turbine	331
13.11	Load Balancing	333
	Ribbon with Spring Cloud	334
13.12	Integrating Other Technologies	335
13.13	Tests	336
	Stubs	336
	Consumer-Driven Contract Test	337
13.14	Experiences with JVM-Based Microservices in the Amazon Cloud (Sascha Möllering)	338
	Conclusion	340
13.15	Conclusion	341
	Essential Points	341
<b>Chapter 14: Technologies for Nanoservices</b>		<b>343</b>
14.1	Why Nanoservices?	344
	Minimum Size of Microservices is Limited	344
	Compromises	345
	Desktop Applications	346
14.2	Nanoservices: Definition	346
14.3	Amazon Lambda	347
	Calling Lambda Functions	348
	Evaluation for Nanoservices	348
	Conclusion	349

14.4 OSGi .....	350
The OSGi Module System .....	350
Handling Bundles in Practice .....	351
Evaluation for Nanoservices .....	353
Conclusion .....	353
14.5 Java EE .....	354
Nanoservices with Java EE .....	355
Microservices with Java EE? .....	355
An Example .....	356
14.6 Vert.x .....	357
Conclusion .....	359
14.7 Erlang .....	360
Evaluation for Nanoservices .....	362
14.8 Seneca .....	363
Evaluation for Nanoservices .....	365
14.9 Conclusion .....	366
Essential Points .....	367
<b>Chapter 15: Getting Started with Microservices .....</b>	<b>369</b>
15.1 Why Microservices? .....	369
15.2 Roads towards Microservices .....	370
15.3 Microservice: Hype or Reality? .....	371
15.4 Conclusion .....	372
<b>Index .....</b>	<b>375</b>



*This page intentionally left blank*

# Preface

Although “microservices” is a new term, the concepts that it represents have been around for long time. In 2006, Werner Vogels (CTO at Amazon) gave a talk at the JAOO conference presenting the Amazon Cloud and Amazon’s partner model. In his talk he mentioned the CAP theorem, today the basis for NoSQL. In addition, he spoke about small teams that develop and run services with their own databases. Today this structure is called DevOps, and the architecture is known as micro services.

Later I was asked to develop a strategy for a client that would enable them to integrate modern technologies into their existing application. After a few attempts to integrate the new technologies directly into the legacy code, we finally built a new application with a completely different modern technology stack alongside the old one. The old and the new application were only coupled via HTML links and via a shared database. Except for the shared database, this is in essence a microservices approach. That happened in 2008.

In 2009, I worked with another client who had divided his complete infrastructure into REST services, each being developed by individual teams. This would also be called microservices today. Many other companies with a web-based business model had already implemented similar architectures at that time. Lately, I have also realized how continuous delivery influences software architecture. This is another area where microservices offer a number of advantages.

This is the reason for writing this book—a number of people have been pursuing a microservices approach for a long time, among them some very experienced architects. Like every other approach to architecture, microservices cannot solve every problem. However, this concept represents an interesting alternative to existing approaches.

---

## Overview of the Book

This book provides a detailed introduction to microservices. Architecture and organization are the main topics. However, technical implementation strategies are not neglected. A complete example of a microservice-based system demonstrates a concrete technical implementation. The discussion of technologies for nanoservices

illustrates that modularization does not stop with microservices. The book provides all the necessary information for readers to start using microservices.

---

## For Whom Is the Book Meant?

The book addresses managers, architects, and developers who want to introduce microservices as an architectural approach.

### Managers

Microservices work best when a business is organized to support a microservices-based architecture. In the introduction, managers understand the basic ideas behind microservices. Afterwards they can focus on the organizational impact of using microservices.

### Developers

Developers are provided with a comprehensive introduction to the technical aspects and can acquire the necessary skills to use microservices. A detailed example of a technical implementation of microservices, as well as numerous additional technologies, for example for nanoservices, helps to convey the basic concepts.

### Architects

Architects get to know microservices from an architectural perspective and can at the same time deepen their understanding of the associated technical and organizational issues.

The book highlights possible areas for experimentation and additional information sources. These will help the interested reader to test their new knowledge practically and delve deeper into subjects that are of relevance to them.

---

## Structure and Coverage

The book is organized into four parts.

### Part I: Motivation and Basics

The first part of the book explains the motivation for using microservices and the foundation of the microservices architecture. Chapter 1, “Preliminaries,” presents

the basic properties as well as the advantages and disadvantages of microservices. Chapter 2, “Microservice Scenarios,” presents two scenarios for the use of microservices: an e-commerce application and a system for signal processing. This section provides some initial insights into microservices and points out contexts for applications.

## **Part II: Microservices—What, Why, and Why Not?**

Part II not only explains microservices in detail but also deals with their advantages and disadvantages:

- Chapter 3, “What Are Microservices,” investigates the definition of the term “microservices” from three perspectives: the size of a microservice, Conway’s Law (which states that organizations can only create specific software architectures), and finally a technical perspective based on domain-driven Design and Bounded Context.
- The reasons for using microservices are detailed in Chapter 4, “Reasons for Using Microservices.” Microservices have not only technical but also organizational advantages, and there are good reasons for turning to microservices from a business perspective.
- The unique challenges posed by microservices are discussed in Chapter 5, “Challenges.” Among these are technical challenges as well as problems related to architecture, infrastructure, and operation.
- Chapter 6, “Microservices and SOA,” aims at defining the differences between microservices and SOA (service-oriented architecture). At first sight both concepts seem to be closely related. However, a closer look reveals plenty of differences.

## **Part III: Implementing Microservices**

Part III deals with the application of microservices and demonstrates how the advantages that were described in Part II can be obtained and how the associated challenges can be solved.

- Chapter 7, “Architecture of Microservice-Based Systems,” describes the architecture of microservices-based systems. In addition to domain architecture, technical challenges are discussed.
- Chapter 8, “Integration and Communication,” presents the different approaches to the integration of and the communication between microservices. This

includes not only communication via REST or messaging but also the integration of UIs and the replication of data.

- Chapter 9, “Architecture of Individual Microservices,” shows possible architectures for microservices such as CQRS, Event Sourcing, or hexagonal architecture. Finally, suitable technologies for typical challenges are addressed.
- Testing is the main focus of Chapter 10, “Testing Microservices and Microservice-Based Systems.” Tests have to be as independent as possible to enable the independent deployment of the different microservices. However, the tests need to not only check the individual microservices, but also the system in its entirety.
- Operation and Continuous Delivery are addressed in Chapter 11, “Operations and Continuous Delivery of Microservices.” Microservices generate a huge number of deployable artifacts and thus increase the demands on the infrastructure. This is a substantial challenge when introducing microservices.
- Chapter 12, “Organizational Effects of a Microservices-Based Architecture,” illustrates how microservices also influence the organization. After all, microservices are an architecture, which is supposed to influence and improve the organization.

## Part IV: Technologies

The last part of the book shows in detail and at the code level how microservices can be implemented technically:

- Chapter 13, “Example of a Microservices-Based Architecture,” contains an exhaustive example for a microservices architecture based on Java, Spring Boot, Docker, and Spring Cloud. This chapter aims at providing an application, which can be easily run, that illustrates the concepts behind microservices in practical terms and offers a starting point for the implementation of a microservices system and experiments.
- Even smaller than microservices are nanoservices, which are presented in Chapter 14, “Technologies for Nanoservices.” Nanoservices require specific technologies and a number of compromises. The chapter discusses different technologies and their related advantages and disadvantages.
- Chapter 15, “Getting Started with Microservices,” demonstrates how microservices can be adopted.

---

## Essays

The book contains essays that were written by experts of various aspects of microservices. The experts were asked to record their main findings about microservices on approximately two pages. Sometimes these essays complement book chapters, sometimes they focus on other topics, and sometimes they contradict passages in the book. This illustrates that there is, in general, no single correct answer when it comes to software architectures, but rather a collection of different opinions and possibilities. The essays offer the unique opportunity to get to know different viewpoints in order to subsequently develop an opinion.

---

## Paths through the Book

The book offers content suitable for each type of audience. Of course, everybody can and should read the chapters that are primarily meant for people with a different type of job. However, the chapters focused on topics that are most relevant for a certain audience are indicated in Table P.1.

**Table P.1** *Paths through the Book*

Chapter	Developer	Architect	Manager
1 - Preliminaries	X	X	X
2 - Microservice Scenarios	X	X	X
3 - What Are Microservices?	X	X	X
4 - Reasons for Using Microservices	X	X	X
5 - Challenges	X	X	X
6 - Microservices and SOA		X	X
7 - Architecture of Microservice-Based Systems		X	
8 - Integration and Communication	X	X	
9 - Architecture of Individual Microservices	X	X	
10 - Testing Microservices and Microservice-Based Systems	X	X	
11 - Operations and Continuous Delivery of Microservices	X	X	

(Continued)

**Table P.1** *Continued*

Chapter	Developer	Architect	Manager
12 - Organizational Effects of a Microservices-Based Architecture			X
13 - Example of a Microservice-Based Architecture	X		
14 - Technologies for Nanoservices	X	X	
15 - Getting Started with Microservices	X	X	X

Readers who only want to obtain an overview are advised to concentrate on the summary section at the end of each chapter. People who want to gain practical knowledge should commence with Chapters 13 and 14, which deal with concrete technologies and code.

The instructions for experiments, which are given in the sections “Try and Experiment,” help deepen your understanding by providing practical exercises. Whenever a chapter is of particular interest to you, you are encouraged to complete the related exercises to get a better grasp of the topics presented in that chapter.

---

## Supplementary Materials

Errata, links to examples, and additional information can be found at <http://microservices-book.com/>. The example code is available at <https://github.com/ewolff/microservice/>.

Register your copy of *Microservices* at [informit.com](http://informit.com) for convenient access to downloads, updates, and corrections as they become available. To start the registration process, go to [informit.com/register](http://informit.com/register) and log in or create an account. Enter the product ISBN 9780134602417 and click Submit. Once the process is complete, you will find any available bonus content under “Registered Products.”

# Acknowledgments

I would like to thank everybody with whom I have discussed microservices and all the people who asked questions or worked with me—way too many to list them all. The interactions and discussions were very fruitful and fun!

I would like to mention especially Jochen Binder, Matthias Bohlen, Merten Driemeyer, Martin Eigenbrodt, Oliver B. Fischer, Lars Gentsch, Oliver Gierke, Boris Gloger, Alexander Heusingfeld, Christine Koppelt, Andreas Krüger, Tammo van Lessen, Sascha Möllering, André Neubauer, Till Schulte-Coerne, Stefan Tilkov, Kai Tödter, Oliver Wolf, and Stefan Zörner.

As a native speaker, Matt Duckhouse has added some significant improvements to the text and improved its readability.

My employer, innoQ, has also played an important role throughout the writing process. Many of the discussions and suggestions of my innoQ colleagues are reflected in the book.

Finally, I would like to thank my friends and family—especially my wife, whom I have often neglected while working on the book. In addition, I would like to thank her for the English translation of the book.

Of course, my thanks also go to all the people who have been working on the technologies that are mentioned in the book and thus have laid the foundation for the development of microservices. Special thanks also due to the experts who shared their knowledge of and experience with microservices in the essays.

Leanpub has provided me with the technical infrastructure to create the translation. It has been a pleasure to work with it, and it is quite likely that the translation would not exist without Leanpub.

Addison-Wesley enabled me to take the English translation to the next level. Chris Zahn, Chris Guzikowski, Lori Lyons and Dhayanidhi Karunanidhi provided excellent support for that process.

Last but not least, I would like to thank dpunkt.verlag and René Schönfeldt, who supported me very professionally during the genesis of the original German version.



*This page intentionally left blank*

# About the Author



**Eberhard Wolff**, a Fellow at innoQ in Germany, has more than 15 years of experience as an architect and consultant working at the intersection of business and technology. He has given talks and keynote addresses at several international conferences, served on multiple conference program committees, and written more than 100 articles and books. His technological focus is on modern architectures—often involving cloud, continuous delivery, DevOps, microservices, or NoSQL.

*This page intentionally left blank*

## PART II



# Microservices: What, Why, and Why Not?

Part II discusses the different facets of microservice-based architectures to present the diverse possibilities offered by microservices. Advantages as well as disadvantages are addressed so that the reader can evaluate what can be gained by using microservices and which points require special attention and care during the implementation of microservice-based architectures.

**Chapter 3, “What Are Microservices,”** explains the term “**microservice**” in detail. The term is dissected from different perspectives, which is essential for an in-depth understanding of the microservice approach. Important issues are the size of a microservice, Conway’s Law as organizational influence, and domain-driven design particularly with respect to *Bounded Context* from a domain perspective. Furthermore, the chapter addresses the question of whether a microservice should contain a UI.

**Chapter 4, “Reasons for Using Microservices,”** focuses on the advantages of microservices, taking alternately technical, organizational, and business perspectives.

**Chapter 5, “Challenges,”** deals with the associated challenges in the areas of technology, architecture, infrastructure, and operation.

**Chapter 6, “Microservices and SOA,”** distinguishes microservices from service-oriented architecture (SOA). By making this distinction microservices are viewed from a new perspective, which helps to further clarify the microservices approach. Besides, microservices have been frequently compared to SOAs.

*This page intentionally left blank*

## Chapter 3

---

---

# What Are Microservices?

Section 1.1 provided an initial definition of the term microservice. However, there are a number of different ways to define microservices. The different definitions are based on different aspects of microservices. They also show for which reasons the use of microservices is advantageous. At the end of the chapter the reader should have his or her own definition of the term microservice—depending on the individual project scenario.

The chapter discusses the term microservice from different perspectives:

- Section 3.1 focuses on the size of microservices.
- Section 3.2 explains the relationship between microservices, architecture, and organization by using the Conway’s Law.
- Section 3.3 presents a domain architecture of microservices based on domain-driven design (DDD) and bounded context.
- Section 3.5 explains why microservices should contain a user interface (UI).

---

### 3.1 Size of a Microservice

The name “microservices” conveys the fact that the size of the service matters; obviously, microservices are supposed to be small.

One way to define the size of a microservice is to count the lines of code (LOC).<sup>1</sup> However, such an approach has a number of problems:

- It depends on the programming language used. Some languages require more code than others to express the same functionality—and microservices are explicitly not supposed to predetermine the technology stack. Therefore, defining microservices based on this metric is not very useful.
- Finally, microservices represent an architecture approach. Architectures, however, should follow the conditions in the domain rather than adhering to technical metrics such as LOC. Also for this reason attempts to determine size based on code lines should be viewed critically.

In spite of the voiced criticism, LOC can be an indicator for a microservice. Still, the question as to the ideal size of a microservice remains. How many LOC may a microservice have? Even if there are no absolute standard values, there are nevertheless influencing factors, which may argue for larger or smaller microservices.

## Modularization

One factor is modularization. Teams develop software in modules to be better able to deal with its complexity; instead of having to understand the entire software package, developers only need to understand the module(s) they are working on as well as the interplay between the different modules. This is the only way for a team to work productively in spite of the enormous complexity of a typical software system. In daily life there are often problems as modules get larger than originally planned. This makes them hard to understand and hard to maintain, because changes require an understanding of the entire module. Thus it is very sensible to keep microservices as small as possible. On the other hand, microservices, unlike many other approaches to modularization, have an overhead.

## Distributed Communication

Microservices run within independent processes. Therefore, communication between microservices is distributed communication via the network. For this type of system, the “First Rule of Distributed Object Design”<sup>2</sup> applies. This rule states that systems should not be distributed if it can be avoided. The reason for this is that

---

1. <http://yobriefca.se/blog/2013/04/28/micro-service-architecture/>

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

a call on another system via the network is orders of magnitude slower than a direct call within the same process. In addition to the pure latency time, serialization and deserialization of parameters and results are time consuming. These processes not only take a long time, but also cost CPU capacity.

Moreover, distributed calls might fail because the network is temporarily unavailable or the called server cannot be reached—for instance due to a crash. This increases complexity when implementing distributed systems, because the caller has to deal with these errors in a sensible manner.

Experience<sup>3</sup> teaches us that microservice-based architectures work in spite of these problems. When microservices are designed to be especially small, the amount of distributed communication increases and the overall system gets slower. This is an argument for larger microservices. When a microservice contains a UI and fully implements a specific part of the domain, it can operate without calling on other microservices in most cases, because all components of this part of the domain are implemented within one microservice. The desire to limit distributed communication is another reason to build systems according to the domain.

## Sustainable Architecture

Microservices also use distribution to design architecture in a sustainable manner through distribution into individual microservices: it is much more difficult to use a microservice than a class. The developer has to deal with the distribution technology and has to use the microservice interface. In addition, he or she might have to make preparations for tests to include the called microservice or replace it with a stub. Finally, he has to contact the team responsible for the respective microservice.

To use a class within a deployment monolith is much simpler—even if the class belongs to a completely different part of the monolith and falls within the responsibility of another team. However, because it is so simple to implement a dependency between two classes, unintended dependencies tend to accumulate within deployment monoliths. In the case of microservices dependencies are harder to implement, which prevents the creation of unintended dependencies.

## Refactoring

However, the boundaries between microservices also create challenges, for instance during refactoring. If it becomes apparent that a piece of functionality does not fit well within its present microservice, it has to be moved to another microservice. If the target microservice is written in a different programming language, this transfer

---

3. <http://martinfowler.com/articles/distributed-objects-microservices.html>



inevitably leads to a new implementation. Such problems do not arise when functionalities are moved within a microservice. This consideration may argue for larger microservices, and this topic is the focus of section 7.3.

## Team Size

The independent deployment of microservices and the division of the development effort into teams result in an upper limit for the size of an individual microservice. A team should be able to implement features within a microservice and deploy those features into production independently of other teams. By ensuring this, the architecture enables the scaling of development without requiring too much coordination effort between the teams.

A team has to be able to implement features independently of the other teams. Therefore, at first glance it seems like the microservice should be large enough to enable the implementation of different features. When microservices are smaller, a team can be responsible for several microservices, which together enable the implementation of a domain. A lower limit for the microservice size does not result from the independent deployment and the division into teams.

However, an upper limit does result from it: when a microservice has reached a size that prevents its further development by a single team, it is too large. For that matter a team should have a size that is especially well suited for agile processes, which is typically three to nine people. Thus a microservice should never grow so large that a team of three to nine people cannot develop it further by themselves. In addition to the sheer size, the number of features to be implemented in an individual microservice plays an important role. Whenever a large number of changes is necessary within a short time, a team can rapidly become overloaded. Section 12.2 highlights alternatives that enable several teams to work on the same microservice. However, in general a microservice should never grow so large that several teams are necessary to work on it.

## Infrastructure

Another important factor influencing the size of a microservice is the infrastructure. Each microservice has to be able to be deployed independently. It must have a continuous delivery pipeline and an infrastructure for running the microservice, which has to be present not only in production but also during the different test stages. Also databases and application servers might belong to infrastructure. Moreover, there has to be a build system for the microservice. The code for the microservice has to be versioned independently of that for other microservices. Thus a project within version control has to exist for the microservice.

Depending on the effort that is necessary to provide the required infrastructure for a microservice, the sensible size for a microservice can vary. When a small microservice size is chosen, the system is distributed into many microservices, thus requiring more infrastructure. In the case of larger microservices, the system overall contains fewer microservices and consequently requires less infrastructure.

Build and deployment of microservices should anyhow be automated. Nevertheless, it can be laborious to provide all necessary infrastructure components for a microservice. Once setting up the infrastructure for new microservices is automated, the expenditure for providing infrastructures for additional microservices decreases. This automation enables further reduction of the microservice size. Companies that have been working with microservices for some time usually simplify the creation of new microservices by providing the necessary infrastructure in an automated manner.

Additionally, some technologies enable reduction of the infrastructure overhead to such an extent that substantially smaller microservices are possible—however, with a number of limitations in such cases. Such nanoservices are discussed in Chapter 14, “Technologies for Microservices.”

## Replaceability

A microservice should be as easy to replace as possible. Replacing a microservice can be sensible when its technology becomes outdated or if the microservice code is of such bad quality that it cannot be developed any further. The replaceability of microservices is an advantage when compared to monolithic applications, which can hardly be replaced at all. When a monolith cannot be reasonably maintained anymore, its development has either to be continued in spite of the associated high costs or a similarly cost-intensive migration has to take place. The smaller a microservice is, the easier it is to replace it with a new implementation. Above a certain size a microservice may be difficult to replace, for it then poses the same challenges as a monolith. Replaceability thus limits the size of a microservice.

## Transactions and Consistency

Transactions possess the so-called ACID characteristics:

- **Atomicity** indicates that a given transaction is either executed completely or not at all. In case of an error, all changes are reversed.
- **Consistency** means that data is consistent before and after the execution of a transaction—database constraints, for instance, are not violated.

- **Isolation** indicates that the operations of transactions are separated from each other.
- **Durability** indicates permanence: changes to the data are stored and are still available after a crash or other interruption of service.

Within a microservice, changes to a transaction can take place. Moreover, the consistency of data in a microservice can be guaranteed very easily. Beyond an individual microservice, this gets difficult, and overall coordination is necessary. Upon the rollback of a transaction all changes made by all microservices would have to be reversed. This is laborious and hard to implement, for the delivery of the decision that changes have to be reversed has to be guaranteed. However, communication within networks is unreliable. Until it is decided whether a change may take place, further changes to the data are barred. If additional changes have taken place, it might no longer be possible to reverse a certain change. However, when microservices are kept from introducing data changes for some time, system throughput is reduced.

However, when communications occur via messaging systems, transactions are possible (see section 8.4). With this approach, transactions are also possible without a close link between the microservices.

## Consistency

In addition to transactions, data consistency is important. An order, for instance, also has to be recorded as revenue. Only then will revenue and order data be consistent. Data consistency can be achieved only through close coordination. Data consistency can hardly be guaranteed across microservices. This does not mean that the revenue for an order will not be recorded at all. However, it will likely not happen exactly at the same point of time and maybe not even within one minute of order processing because the communication occurs via the network—and is consequently slow and unreliable.

Data changes within a transaction and data consistency are only possible when all data being processed is part of the same microservice. Therefore, data changes determine the lower size limit for a microservice: when transactions are supposed to encompass several microservices and data consistency is required across several microservices, the microservices have been designed too small.

## Compensation Transactions across Microservices

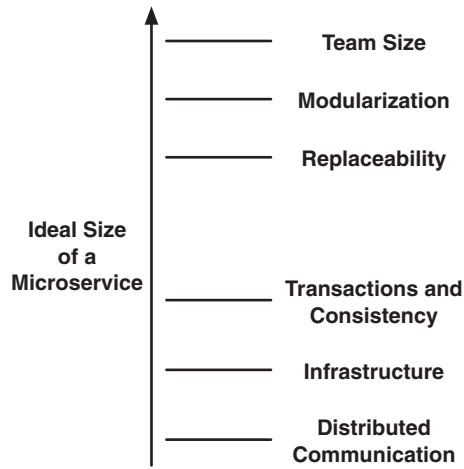
At least in the case of transactions there is an alternative: if a data change has to be rolled back in the end, compensation transactions can be used for that.

A classic example for a distributed transaction is a travel booking, which consists of a hotel, a rental car, and a flight. Either everything has to be booked together or nothing at all. Within real systems and also within microservices, this functionality is divided into three microservices because the three tasks are very different. Inquiries are sent to the different systems whether the desired hotel room, rental car, and flight are available. If all are available, everything is reserved. If, for instance, the hotel room suddenly becomes unavailable, the reservations for the flight and the rental car have to be cancelled. However, in the real world the concerned companies will likely demand a fee for the booking cancellation. Due to that, the cancellation is not only a technical event happening in the background like a transaction rollback but also a business process. This is much easier to represent with a compensation transaction. With this approach, transactions across several elements in microservice environments can also be implemented without the presence of a close technical link. A compensation transaction is just a normal service call. Technical as well as business reasons can lead to the use of mechanisms such as compensation transactions for microservices.

## Summary

In conclusion, the following factors influence the size of a microservice (see Figure 3.1):

- The team size sets an upper limit; a microservice should never be so large that one very large team or several teams are required to work on it. Eventually, the teams are supposed to work and bring software into production independently of each other. This can only be achieved when each team works on a separate deployment unit—that is, a separate microservice. However, one team can work on several microservices.
- Modularization further limits the size of a microservice: The microservice should preferably be of a size that enables a developer to understand all its aspects and further develop it. Even smaller is of course better. This limit is below the team size: whatever one developer can still understand, a team should still be able to develop further.
- Replaceability reduces with the size of the microservice. Therefore, replaceability can influence the upper size limit for a microservice. This limit lies below the one set by modularization: when somebody decides to replace a microservice, this person has first of all to be able to understand the microservice.
- A lower limit is set by infrastructure: if it is too laborious to provide the necessary infrastructure for a microservice, the number of microservices should be kept rather small; consequently the size of each microservice will be larger.



**Figure 3.1** *Factors Influencing the Size of a Microservice*

- Similarly, distributed communication overhead increases with the number of microservices. For this reason, the size of microservices should not be set too small.
- Consistency of data and transactions can only be ensured within a microservice. Therefore, microservices should not be so small that consistency and transactions must be ensured across several microservices.

These factors not only influence the size of microservices but also reflect a certain idea of microservices. According to this idea, the main advantages of microservices are independent deployment and the independent work of the different teams, along with the replaceability of microservices. The optimal size of a microservice can be deduced from these desired features.

However, there are also other reasons for microservices. When microservices are, for instance, introduced because of their independent scaling, a microservice size has to be chosen that ensures that each microservice is a unit, which has to scale independently.

How small or large a microservice can be, cannot be deduced solely from these criteria. This also depends on the technology being used. Especially the effort necessary for providing infrastructure for a microservice and the distributed communication depends on the utilized technology. Chapter 14 looks at technologies, which make the development of very small services possible—denoted as nanoservices. These nanoservices have different advantages and disadvantages to microservices, which, for instance, are implemented using technologies presented in Chapter 13, “Example of a Microservice-based Architecture.”

Thus, there is no ideal size. The actual microservice size will depend on the technology and the use case of an individual microservice.

### Try and Experiment

How great is the effort required for the deployment of a microservice in your language, platform, and infrastructure?

- Is it just a simple process? Or is it a complex infrastructure containing application servers or other infrastructure elements?
- How can the effort for the deployment be reduced so that smaller microservices become possible?

Based on this information you can define a lower limit for the size of a microservice. Upper limits depend on team size and modularization, so you should also think of appropriate limits in those terms.

---

## 3.2 Conway's Law

Conway's Law<sup>4</sup> was coined by the American computer scientist Melvin Edward Conway and indicates the following:

Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.

It is important to know that this law is meant to apply not only to software but to any kind of design. The communication structures that Conway mentions, do not have to be identical to the organization chart. Often there are informal communication structures, which also have to be considered in this context. In addition, the geographical distribution of teams can influence communication. After all it is much simpler to talk to a colleague who works in the same room or at least in the same office than with one working in a different city or even in a different time zone.

---

4. <http://www.melconway.com/research/committees.html>

## Reasons for the Law

Conway's Law derives from the fact that each organizational unit designs a specific part of the architecture. If two architectural parts have an interface, coordination in regards to this interface is required—and, consequently, a communication relationship between the organizational units that are responsible for the respective parts of the architecture.

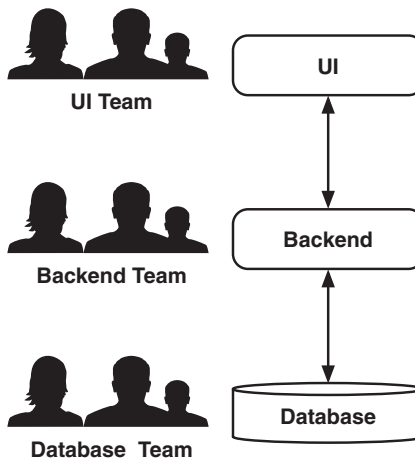
From Conway's Law it can also be deduced that design modularization is sensible. Via such a design, it is possible to ensure that not every team member has to constantly coordinate with every other team member. Instead the developers working on the same module can closely coordinate their efforts, while team members working on different modules only have to coordinate when they develop an interface—and even then only in regards to the specific design of the external features of this interface.

However, the communication relationships extend beyond that. It is much easier to collaborate with a team within the same building than with a team located in another city, another country, or even within a different time zone. Therefore, architectural parts having numerous communication relationships are better implemented by teams that are geographically close to each other, because it is easier for them to communicate with each other. In the end, the Conway's Law focuses not on the organization chart but on the real communication relationships.

By the way, Conway postulated that a large organization has numerous communication relationships. Thus communication becomes more difficult or even impossible in the end. As a consequence, the architecture can be increasingly affected and finally break down. In the end, having too many communication relationships is a real risk for a project.

## The Law as Limitation

Normally Conway's Law is viewed as a limitation, especially from the perspective of software development. Let us assume that a project is modularized according to technical aspects (see Figure 3.2). All developers with a UI focus are grouped into one team, the developers with backend focus are put into a second team, and data bank experts make up the third team. This distribution has the advantage that all three teams consist of experts for the respective technology. This makes it easy and transparent to create this type of organization. Moreover, this distribution also appears logical. Team members can easily support each other, and technical exchange is also facilitated.



**Figure 3.2** *Technical Project Distribution*

According to Conway's Law, it follows from such a distribution that the three teams will implement three technical layers: a UI, a backend, and a database. The chosen distribution corresponds to the organization, which is in fact sensibly built. However, this distribution has a decisive disadvantage: a typical feature requires changes to UI, backend, and database. The UI has to render the new features for the clients, the backend has to implement the logic, and the database has to create structures for the storage of the respective data. This results in the following disadvantages:

- The person wishing to have a feature implemented has to talk to all three teams.
- The teams have to coordinate their work and create new interfaces.
- The work of the different teams has to be coordinated in a manner that ensures that their efforts temporally fit together. The backend, for instance, cannot really work without getting input from the database, and the UI cannot work without input from the backend.
- When the teams work in sprints, these dependencies cause time delays: The database team generates in its first sprint the necessary changes, within the second sprint the backend team implements the logic, and in the third sprint the UI is dealt with. Therefore, it takes three sprints to implement a single feature.



In the end this approach creates a large number of dependencies as well as a high communication and coordination overhead. Thus this type of organization does not make much sense if the main goal is to implement new features as rapidly as possible.

Many teams following this approach do not realize its impact on architecture and do not consider this aspect further. This type of organization focuses instead on the notion that developers with similar skills should be grouped together within the organization. This organization becomes an obstacle to a design driven by the domain like microservices, whose development is not compatible with the division of teams into technical layers.

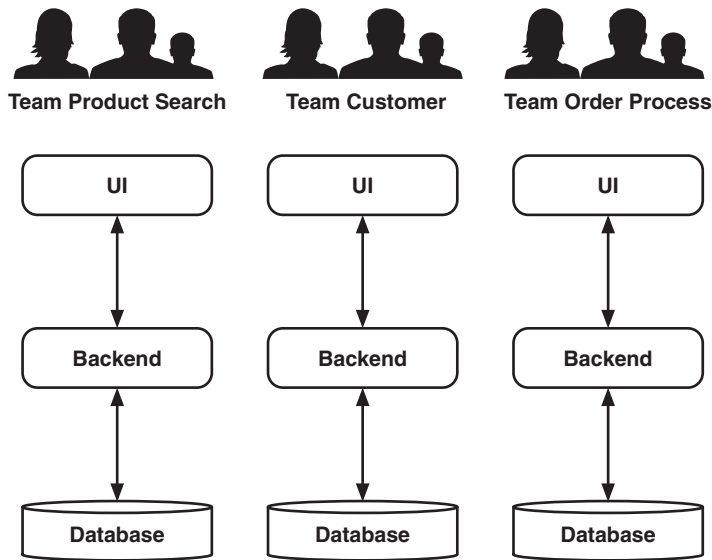
## The Law as Enabler

However, Conway's Law can also be used to support approaches like microservices. If the goal is to develop individual components as independently of each other as possible, the system can be distributed into domain components. Based on these domain components, teams can be created. Figure 3.3 illustrates this principle: There are individual teams for product search, clients, and the order process. These teams work on their respective components, which can be technically divided into UI, back-end, and database. By the way, the domain components are not explicitly named in the figure, for they are identical to the team names. Components and teams are synonymous. This approach corresponds to the idea of so-called cross-functional teams, as proposed by methods such as Scrum. These teams should encompass different roles so that they can cover a large range of tasks. Only a team designed along such principles can be in charge of a component—from engineering requirements via implementation through to operation.

The division into technical artifacts and the interface between the artifacts can then be settled within the teams. In the easiest case, developers only have to talk to developers sitting next to them to do so. Between teams, coordination is more complex. However, inter-team coordination is not required very often, since features are ideally implemented by independent teams. Moreover, this approach creates thin interfaces between the components. This avoids laborious coordination across teams to define the interface.

Ultimately, the key message to be taken from Conway's Law is that architecture and organization are just two sides of the same coin. When this insight is cleverly put to use, the system will have a clear and useful architecture for the project. Architecture and organization have the common goal to ensure that teams can work in an unobstructed manner and with as little coordination overhead as possible.

The clean separation of functionality into components also facilitates maintenance. Since an individual team is responsible for individual functionality and component, this distribution will have long-term stability, and consequently the system will remain maintainable.



**Figure 3.3** *Project by Domains*

The teams need requirements to work upon. This means that the teams need to contact people who define the requirements. This affects the organization beyond the projects, for the requirements come from the departments of the enterprise, and these also according to Conway's Law have to correspond to the team structures within the project and the domain architecture. Conway's Law can be expanded beyond software development to the communication structures of the entire organization, including the users. To put it the other way round: the team structure within the project and consequently the architecture of a microservice system can follow from the organization of the departments of the enterprise.

## The Law and Microservices

The previous discussion highlighted the relationship between architecture and organization of a project only in a general manner. It would be perfectly conceivable to align the architecture along functionalities and devise teams, each of which are in charge for a separate functionality without using microservices. In this case the project would develop a deployment monolith within which all functionalities are implemented. However, microservices support this approach. Section 3.1 already discussed that microservices offer technical independence. In conjunction with the division by domains, the teams become even more independent of each other and have even less need to coordinate their work. The technical coordination as well as the coordination concerning the domains can be reduced to the absolute minimum. This makes it far easier to work in parallel on numerous features and also to bring the features in production.

Microservices as a technical architecture are especially well suited to support the approach to devise a Conway’s Law–based distribution of functionalities. In fact, exactly this aspect is an essential characteristic of a microservices-based architecture.

However, orienting the architecture according to the communication structures entails that a change to the one also requires a change of the other. This makes architectural changes between microservices more difficult and makes the overall process less flexible. Whenever a piece of functionality is moved from one microservice to another, this might have the consequence that another team has to take care of this functionality from that point on. This type of organizational change renders software changes more complex.

As a next step this chapter will address how the distribution by domain can best be implemented. Domain-driven design (DDD) is helpful for that.

### Try and Experiment

Have a look at a project you know:

- What does the team structure look like?
  - Is it technically motivated, or is it divided by domain?
  - Would the structure have to be changed to implement a microservices-based approach?
  - How would it have to be changed?
- Is there a sensible way to distribute the architecture onto different teams? Eventually each team should be in charge of independent domain components and be able to implement features relating to them.
  - Which architectural changes would be necessary?
  - How laborious would the changes be?

---

## 3.3 Domain-Driven Design and Bounded Context

In his book of the same title, Eric Evans formulated domain-driven design (DDD)<sup>5</sup> as pattern language. It is a collection of connected design patterns and supposed to support software development especially in complex domains. In the following text, the names of design patterns from Evan’s book are written in *italics*.

---

5. Eric Evans. 2003. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Boston: Addison-Wesley.

Domain-driven design is important for understanding microservices, for it supports the structuring of larger systems according to domains. Exactly such a model is necessary for the division of a system into microservices. Each microservice is meant to constitute a domain, which is designed in such a way that only one microservice has to be changed in order to implement changes or to introduce new features. Only then is the maximal benefit to be derived from independent development in different teams, as several features can be implemented in parallel without the need for extended coordination.

## Ubiquitous Language

DDD defines a basis for how a model for a domain can be designed. An essential foundation of DDD is *Ubiquitous Language*. This expression denotes that the software should use exactly the same terms as the domain experts. This applies on all levels: in regards to code and variable names as well as for database schemas. This practice ensures that the software really encompasses and implements the critical domain elements. Let us assume for instance that there are express orders in an e-commerce system. One possibility would be to generate a Boolean value with the name “fast” in the order table. This creates the following problem: domain experts have to translate the term “express order,” which they use on a daily basis, into “order with a specific Boolean value.” They might not even know what Boolean values are. This renders any discussion of the model more difficult, for terms have to be constantly explained and related to each other. The better approach is to call the table within the database scheme “express order.” In that case it is completely transparent how the domain terms are implemented in the system.

## Building Blocks

To design a domain model, DDD identifies basic patterns:

- *Entity* is an object with an individual identity. In an e-commerce application, the customer or the items could be examples for *Entities*. *Entities* are typically stored in databases. However, this is only the technical implementation of the concept *Entity*. An *Entity* belongs in essence to the domain modeling like the other DDD concepts.
- *Value Objects* do not have their own identity. An address can be an example of a *Value Object*, for it makes only sense in the context of a specific customer and therefore does not have an independent identity.
- *Aggregates* are composite domain objects. They facilitate the handling of invariants and other conditions. An order, for instance, can be an *Aggregate* of order lines. This can be used to ensure that an order from a new customer does not exceed a certain value. This is a condition that has to be fulfilled by calculating values from the order lines so that the order as *Aggregate* can control these conditions.

- *Services* contain business logic. DDD focuses on modeling business logic as *Entities*, *Value Objects*, and *Aggregates*. However, logic accessing several such objects cannot be sensibly modeled using these objects. For these cases there are *Services*. The order process could be such a *Service*, for it needs access to items and customers and requires the *Entity* order.
- *Repositories* serve to access all *Entities* of a type. Typically, there is a persistency technology like a database behind a *Repository*.
- *Factories* are mostly useful to generate complex domain objects. This is especially the case when these contain for instance many associations.

*Aggregates* are of special importance in the context of microservices: within an *Aggregate* consistency can be enforced. Because consistency is necessary, parallel changes have to be coordinated in an *Aggregate*. Otherwise two parallel changes might endanger consistency. For instance, when two order positions are included in parallel into an order, consistency can be endangered. The order has already a value of €900 and is maximally allowed to reach €1000. If two order positions of €60 each are added in parallel, both might calculate a still acceptable total value of €960 based on the initial value of €900. Therefore, changes have to be serialized so that the final result of €1020 can be controlled. Accordingly, changes to *Aggregates* have to be serialized. For this reason, an *Aggregate* cannot be distributed across two microservices. In such a scenario consistency cannot be ensured. Consequently, *Aggregates* cannot be divided between microservices.

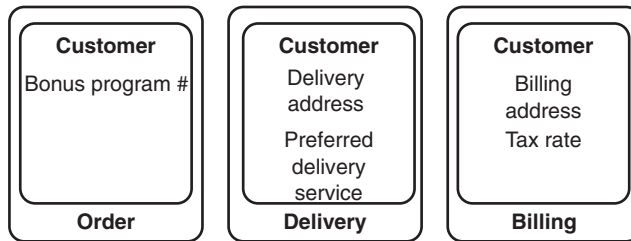
## Bounded Context

Building blocks such as *Aggregate* represent for many people the core of DDD. DDD describes, along with strategic design, how different domain models interact and how more complex systems can be built up this way. This aspect of DDD is probably even more important than the building blocks. In any case it is the concept of DDD, which influences microservices.

The central element of strategic designs is the *Bounded Context*. The underlying reasoning is that each domain model is only sensible in certain limits within a system. In e-commerce, for instance, number, size, and weight of the ordered items are of interest in regards to delivery, for they influence delivery routes and costs. For accounting on the other hand prices and tax rates are relevant. A complex system consists of several *Bounded Contexts*. In this it resembles the way complex biological organisms are built out of individual cells, which are likewise separate entities with their own inner life.

### Bounded Context: An Example

The customer from the e-commerce system shall serve as an example for a *Bounded Context* (see Figure 3.4). The different *Bounded Contexts* are Order, Delivery, and Billing. The component Order is responsible for the order process. The component Delivery implements the delivery process. The component Billing generates the bills.



**Figure 3.4** *Project by Domains*

Each of these Bounded Contexts requires certain customer data:

- Upon ordering the customer is supposed to be rewarded with points in a bonus program. In this Bounded Context the number of the customer has to be known to the bonus program.
- For Delivery the delivery address and the preferred delivery service of the customer are relevant.
- Finally, for generating the bill the billing address and the tax rate of the customer have to be known.

In this manner each *Bounded Context* has its own model of the customer. This renders it possible to independently change microservices. If for instance more information regarding the customer is necessary for generating bills, only changes to the *Bounded Context* billing are necessary.

It might be sensible to store basic information concerning the customer in a separate *Bounded Context*. Such fundamental data is probably sensible in many *Bounded Contexts*. To this purpose the *Bounded Contexts* can cooperate (see below).

(continued)

A universal model of the customer, however, is hardly sensible. It would be very complex since it would have to contain all information regarding the customer. Moreover, each change to customer information, which is necessary in a certain context, would concern the universal model. This would render such changes very complicated and would probably result in permanent changes to the model.

To illustrate the system setup in the different *Bounded Contexts* a *Context Map* can be used (see section 7.2). Each of the *Bounded Contexts* then can be implemented within one or several microservices.

### Collaboration between *Bounded Contexts*

How are the individual *Bounded Contexts* connected? There are different possibilities:

- In case of a *Shared Kernel* the domain models share some common elements; however, in other areas they differ.
- *Customer/Supplier* means that a subsystem offers a domain model for the caller. The caller in this case is the client who determines the exact setup of the model.
- This is very different in the case of *Conformist*: The caller uses the same model as the subsystem, and the other model is thereby forced upon him. This approach is relatively easy, for there is no need for translation. One example is a standard software for a certain domain. The developers of this software likely know a lot about the domain since they have seen many different use cases. The caller can use this model to profit from the knowledge from the modeling.
- The *Anticorruption Layer* translates a domain model into another one so that both are completely decoupled. This enables the integration of legacy systems without having to take over the domain models. Often data modeling is not very meaningful in legacy systems.

- *Separate Ways* means that the two systems are not integrated, but stay independent of each other.
- In the case of *Open Host Service*, the *Bounded Context* offers special services everybody can use. In this way everybody can assemble their own integration. This is especially useful when an integration with numerous other systems is necessary and when the implementation of these integrations is too laborious.
- *Published Language* achieves similar things. It offers a certain domain modeling as a common language between the *Bounded Contexts*. Since it is widely used, this language can hardly be changed anymore afterwards.

## Bounded Context and Microservices

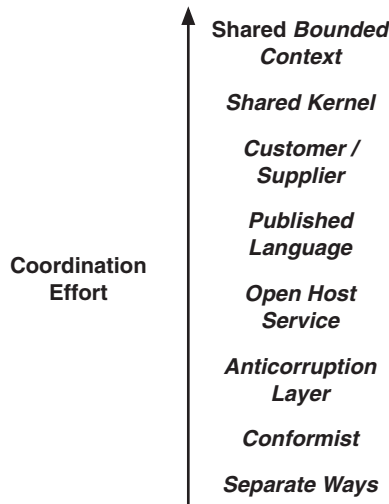
Each microservice is meant to model one domain so that new features or changes have only to be implemented within one microservice. Such a model can be designed based on *Bounded Context*.

One team can work on one or several *Bounded Contexts*, which each serve as a foundation for one or several microservices. Changes and new features are supposed to concern typically only one *Bounded Context*—and thus only one team. This ensures that teams can work largely independently of each other. A *Bounded Context* can be divided into multiple microservices if that seems sensible. There can be technical reasons for that. For example, a certain part of a *Bounded Context* might have to be scaled up to a larger extent than the others. This is simpler if this part is separated into its own microservice. However, designing microservices that contain multiple *Bounded Contexts* should be avoided, for this entails that several new features might have to be implemented in one microservice. This interferes with the goal to develop features independently.

Nevertheless, it is possible that a special requirement comprises many *Bounded Contexts*—in that case additional coordination and communication will be required.

The coordination between teams can be regulated via different collaboration possibilities. These influence the independence of the teams as well: *Separate Ways*, *Anticorruption Layer* or *Open Host Service* offer a lot of independence. *Conformist* or *Customer/Supplier* on the other hand tie the domain models very closely together. For *Customer/Supplier* the teams have to coordinate their efforts closely: the supplier needs to understand the requirements of the customer. For *Conformist*, however, the teams do not need to coordinate: one team defines the model that the other team just uses unchanged (see Figure 3.5).





**Figure 3.5** *Communication Effort of Different Collaborations*

As in the case of Conway’s Law from section 3.2, it becomes very apparent that organization and architecture are very closely linked. When the architecture enables a distribution of the domains in which the implementation of new features only requires changes to a defined part of the architecture, these parts can be distributed to different teams in such a way that these teams can work largely independently of each other. DDD and especially *Bounded Context* demonstrate what such a distribution can look like and how the parts can work together and how they have to coordinate.

## Large-Scale Structure

With large-scale structure, DDD also addresses the question how the system in its entirety can be viewed from the different *Bounded Contexts* with respect to microservices.

- A *System Metaphor* can serve to define the fundamental structure of the entire system. For example, an e-commerce system can orient itself according to the shopping process: the customer starts out looking for products, then he/she will compare items, select one item, and order it. This can give rise to three microservices: search, comparison, and order.
- A *Responsibility Layer* divides the system into layers with different responsibilities. Layers can call other layers only if those are located below them. This does not refer to a technical division into database, UI and logic. In an

e-commerce system, domain layers might be (for example) the catalog, the order process, and billing. The catalog can call on the order process, and the order process can call on the generation of the bill. However, calls into the other direction are not permitted.

- *Evolving Order* suggests it is best not to determine the overall structure too rigidly. Instead, the order should arise from the individual components in a stepwise manner.

These approaches can provide an idea how the architecture of a system, which consists of different microservices, can be organized (see also Chapter 7, “Architecture of Microservice-based Systems”).

### Try and Experiment

Look at a project you know:

- Which *Bounded Contexts* can you identify?
- Generate an overview of the *Bounded Contexts* in a *Context Map*. Compare section 7.2.
- How do the Bounded Contexts cooperate? (Anticorruption Layer Customer/Supplier etc.). Add this information to the Context Map.
- Would other mechanisms have been better at certain places? Why?
- How could the *Bounded Contexts* be sensibly distributed to teams so that features are implemented by independent teams?

These questions might be hard to answer because you need to get a new perspective on the system and how the domains are modeled in the system.

---

## 3.4 Why You Should Avoid a Canonical Data Model (Stefan Tilkov)

*by Stefan Tilkov, innoQ*

In recent times, I’ve been involved in a few architecture projects on the enterprise level again. If you’ve never been in that world, that is, if you’ve been focusing on

individual systems so far, let me give you the gist of what this kind of environment is like. There are lots of meetings, more meetings, and even more meetings; there's an abundance of slide decks, packed with text and diagrams—none of that Presentation Zen nonsense, please. There are conceptual architecture frameworks, showing different perspectives; there are guidelines and reference architectures, enterprise-wide layering approaches, a little bit of SOA and EAI and ESB and portals and (lately) API talk thrown in for good measure. Vendors and system integrators and (of course) consultants all see their chance to exert influence on strategic decisions, making their products or themselves an integral part of the company's future strategy. It can be a very frustrating but (at least sometimes) also very rewarding experience: those wheels are very big and really hard to turn, but if you manage to turn them, the effect is significant.

It's also amazing to see how many of the things that cause problems when building large systems are repeated on the enterprise level. (We don't often make mistakes ... but if we do, we make them big!) My favorite one is the idea of establishing a canonical data model (CDM) for all of your interfaces.

If you haven't heard of this idea before, a quick summary is: Whatever kind of technology you're using (an ESB, a BPM platform, or just some assembly of services of some kind), you standardize the data models of the business objects you exchange. In its extreme (and very common) form, you end up with having just one kind of Person, Customer, Order, Product, etc., with a set of IDs, attributes, and associations everyone can agree on. It isn't hard to understand why that might seem a very compelling thing to attempt. After all, even a nontechnical manager will understand that the conversion from one data model to another whenever systems need to talk to each other is a complete waste of time. It's obviously a good idea to standardize. Then, anyone who happens to have a model that differs from the canonical one will have to implement a conversion to and from it just once, new systems can just use the CDM directly, and everyone will be able to communicate without further ado!

In fact, it's a horrible, horrible idea. Don't do it.

In his book on domain-driven design, Eric Evans gave a name to a concept that is obvious to anyone who has actually successfully built a larger system: the *Bounded Context*. This is a structuring mechanism that avoids having a single huge model for all of your application, simply because that (a) becomes unmanageable and (b) makes no sense to begin with. It recognizes that a Person or a Contract are different things in different contexts on a *conceptual level*. This is not an implementation problem—it's reality.

If this is true for a large system—and trust me, it is—it's infinitely more true for an enterprise-wide architecture. Of course you can argue that with a CDM, you're

only standardizing the interface layer, but that doesn't change a thing. You're still trying to make everyone agree what a concept means, and my point is that you should recognize that not every single system has the same needs.

But isn't this all just pure theory? Who cares about this, anyway? The amazing thing is that organizations are excellent in generating a huge amount of work based on bad assumptions. The CDM (in the form I've described it here) requires coordination between all the parties that use a particular object in their interfaces (unless you trust that people will be able to just design the right thing from scratch on their own, which you should never do). You'll have meetings with some enterprise architect and a few representatives for specific systems, trying to agree what a customer is. You'll end up with something that has tons of optional attributes because all the participants insisted theirs need to be there, and with lots of things that are kind of weird because they reflect some system's internal restrictions. Despite the fact that it'll take you ages to agree on it, you'll end up with a zombie interface model will be universally hated by everyone who has to work with it.

So is a CDM a universally bad idea? Yes, unless you approach it differently. In many cases, I doubt a CDM's value in the first place and think you are better off with a different and less intrusive kind of specification. But if you want a CDM, here are a number of things you can do to address the problems you'll run into:

- Allow independent parts to be specified independently. If only one system is responsible for a particular part of your data model, leave it to the people to specify what it looks like canonically. Don't make them participate in meetings. If you're unsure whether the data model they create has a significant overlap with another group's, it probably hasn't.
- Standardize on formats and possibly fragments of data models. Don't try to come up with a consistent model of the world. Instead, create small buildings blocks. What I'm thinking of are e.g. small XML or JSON fragments, akin to microformats, that standardize small groups of attributes (I wouldn't call them business objects).
- Most importantly, don't push your model from a central team downwards or outwards to the individual teams. Instead, it should be the teams who decide to "pull" them into their own context when they believe they provide value. It's not you who's doing the really important stuff (even though that's a common delusion that's attached to the mighty Enterprise Architect title). Collect the data models the individual teams provide in a central location, if you must, and make them easy to browse and search. (Think of providing a big elastic search index as opposed to a central UML model.)

What you actually need to do as an enterprise architect is to get out of people's way. In many cases, a crucial ingredient to achieve this is to create as little centralization as possible. It shouldn't be your goal to make everyone do the same thing. It should be your goal to establish a minimal set of rules that enable people to work as independently as possible. A CDM of the kind I've described above is the exact opposite.

---

## 3.5 Microservices with a UI?

This book recommends that you equip microservices with a UI. The UI should offer the functionality of the microservice to the user. In this way, all changes in regards to one area of functionality can be implemented in one microservice—regardless of whether they concern the UI, the logic, or the database. However, microservice experts so far have different opinions in regards to the question of whether the integration of UI into microservices is really required. Ultimately, microservices should not be too large. And when logic is supposed to be used by multiple frontends, a microservice consisting of pure logic without a UI might be sensible. In addition, it is possible to implement the logic and the UI in two different microservices but to have them implemented by one team. This enables implementation of features without coordination across teams.

Focusing on microservices with a UI puts the main emphasis on the distribution of the domain logic instead of a distribution by technical aspects. Many architects are not familiar with the domain architecture, which is especially important for microservices-based architectures. Therefore, a design where the microservices contain the UI is helpful as a first approach in order to focus the architecture on the domains.

### Technical Alternatives

Technically the UI can be implemented as Web UI. When the microservices have a RESTful-HTTP interface, the Web-UI and the RESTful-HTTP interface are very similar—both use HTTP as a protocol. The RESTful-HTTP interface delivers JSON or XML, the Web UI HTML. If the UI is a Single-Page Application, the JavaScript code is likewise delivered via HTTP and communicates with the logic via RESTful HTTP. In case of mobile clients, the technical implementation is more complicated. Section 8.1 explains this in detail. Technically a deployable artifact can deliver via an HTTP interface, JSON/XML, and HTML. In this way it implements the UI and allows other microservices to access the logic.

## Self-Contained System

Instead of calling this approach “Microservice with UI” you can also call it “Self-Contained System” (SCS).<sup>6</sup> SCS define microservices as having about 100 lines of code, of which there might be more than one hundred in a complete project.

An SCS consists of many of those microservices and contains a UI. It should communicate with other SCSs asynchronously, if at all. Ideally each functionality should be implemented in just one SCS, and there should be no need for SCSs to communicate with each other. An alternative approach might be to integrate the SCSs at the UI-level.

In an entire system, there are then only five to 25 of these SCS. An SCS is something one team can easily deal with. Internally the SCS can be divided into multiple microservices.

The following definitions result from this reasoning:

- SCS is something a team works on and which represents a unit in the domain architecture. This can be an order process or a registration. It implements a sensible functionality, and the team can supplement the SCS with new features. An alternative name for a SCS is a vertical. The SCS distributes the architecture by domain. This is a vertical design in contrast to a horizontal design. A horizontal design would divide the system into layers, which are technically motivated—for instance UI, logic, or persistence.
- A microservice is a part of a SCS. It is a technical unit and can be independently deployed. This conforms with the microservice definition put forward in this book. However, the size given in the SCS world corresponds to what this book denotes as nanoservices (see Chapter 14).
- This book refers to nanoservices as units that are still individually deployable but make technical trade-offs in some areas to further reduce the size of the deployment units. For that reason, nanoservices do not share all technical characteristics of microservices.

SCS inspired the definition of microservices as put forward in this book. Still there is no reason not to separate the UI into a different artifact in case the microservice gets otherwise too large. Of course, it is more important that the microservice is small and thus maintainable than to integrate the UI. But the UI and logic should at least be implemented by the same team.

---

6. <http://scs-architecture.org>

## 3.6 Conclusion

Microservices are a modularization approach. For a deeper understanding of microservices, the different perspectives discussed in this chapter are very helpful:

- Section 3.1 focuses on the size of microservices. But a closer look reveals that the size of microservices itself is not that important, even though size is an influencing factor. However, this perspective provides a first impression of what a microservice should be. Team size, modularization, and replaceability of microservices each determine an upper size limit. The lower limit is determined by transactions, consistency, infrastructure, and distributed communication.
- Conway’s Law (section 3.2) shows that the architecture and organization of a project are closely linked—in fact, they are nearly synonymous. Microservices can further improve the independence of teams and thus ideally support architectural designs that aim at the independent development of functionalities. Each team is responsible for a microservice and therefore for a certain part of a domain, so that the teams are largely independent concerning the implementation of new functionalities. Thus, in regards to domain logic there is hardly any need for coordination across teams. The requirement for technical coordination can likewise be reduced to a minimum because of the possibility for technical independence.
- In section 3.3 domain-driven design provides a very good impression as to what the distribution of domains in a project can look like and how the individual parts can be coordinated. Each microservice can represent a *Bounded Context*. This is a self-contained piece of domain logic with an independent domain model. Between the *Bounded Contexts* there are different possibilities for collaboration.
- Finally, section 3.5 demonstrates that microservices should contain a UI to be able to implement the changes for functionality within an individual microservice. This does not necessarily have to be a deployment unit; however, the UI and microservice should be in the responsibility of one team.

Together these different perspectives provide a balanced picture of what constitutes microservices and how they can function.

## Essential Points

To put it differently: A successful project requires three components:

- an organization (This is supported by Conway's Law.)
- a technical approach (This can be microservices.)
- a domain design as offered by DDD and *Bounded Context*

The domain design is especially important for the long-term maintainability of the system.

### Try and Experiment

Look at the three approaches for defining microservices: size, Conway's Law, and domain-driven design.

- Section 1.2 showed the most important advantages of microservices. Which of the goals to be achieved by microservices are best supported by the three definitions? DDD and Conway's Law lead, for instance, to a better time-to-market.
- Which of the three aspects is, in your opinion, the most important? Why?



*This page intentionally left blank*

# Index

## Symbols

0MQ (ZeroMQ), 183

## A

acceptance tests, 59

acceptance-driven design (ATDD), 216

ACID, 31–33

ActiveMQ, 183

adjusting architecture

causes of bad architecture, 110–111

challenges with, 117–118

changes in microservices, 111

changes to overall architecture, 111–112

code reuse versus redundancy, 114–115

code transfer, 113–114

overview, 110

rewriting, 117

shared libraries, 112–113

shared services, 115–116

spawning new services, 116–117

Advanced Message Queuing Protocol (AMQP), 182

advantages of microservices. *See* microservice advantages

aggregates, 41–42

agile architecture, 75–76

AJAX (Asynchronous JavaScript and XML), 170

Amazon

AWS (Amazon Web Services), 338

Cloud

CloudWatch, 340, 349

JVM-based microservices in, 338–340

DynamoDB, 338

Elastic Beanstalk, 349

ElastiCache, 338

Kinesis, 338

Lambda, 347–349

modernization scenario example, 18–19

AMQP (Advanced Message Queuing Protocol), 182

AngularJS, 165

annotations (Hystrix), 330

@Autowired, 334

@EnableCircuitBreaker, 330

@EnableDiscoveryClient, 324–325

@EnableEurekaClient, 324–325, 332

@EnableEurekaServer, 326

@EnableHystrix, 330

@EnableHystrixDashboard, 331, 332

@EnableTurbine, 332

@EnableZuulProxy, 328

@HystrixCommand, 330

@RequestMapping, 307–308

@RestController, 307–308

@RibbonClient, 334

@SpringBootApplication, 307–308

Ansible, 255

anticorruption layer, 44, 109

Apache

Kafka, 183, 338

Mesos, 265

Qpid, 182

Thrift, 180

Zookeeper, 139, 310

API keys, 157

application.properties file, 313

applications

e-commerce legacy application

migration status, 15–16

team creation, 16

e-commerce legacy application scenario

Amazon example, 18–19

approach, 14

challenges, 14–15

feature development, 17–18

general description, 11–12

influence on organization, 18

microservice advantages, 12–13, 17

microservices-based application example

build, 311–313

communication, 327–329

deployment with Docker, 313–314

Docker Compose, 321–324

Docker Machine, 320–321

- applications (*continued*)
  - domain architecture, 304–306
  - integration of other technologies, 335–336
  - JVM-based microservices in Amazon
    - Cloud, 338–340
  - load balancing, 333–334
  - overview, 303–304
  - resilience with Hystrix, 329–333
  - service discovery, 324–326
  - Spring Framework, 306–311
  - summary, 341–342
  - testing, 336–337
  - Vagrant, 314–320
- application-test.properties file, 229, 233
- arbitrary test frameworks, 232
- architecture
  - agile architecture, 75–76
  - architecture management
    - Context Maps, 108–109
    - cycle-free software, 104–105
    - importance of, 107–108
    - tools for, 104–107
  - effect on departments, 291–292
  - of individual microservices
    - CQRS (Command Query Responsibility Segregation), 195–197
    - domain architecture, 193–194
    - event sourcing, 197–199
    - hexagonal architecture, 199–202
    - microservice implementation and, 296–297
    - resilience, 203–208
    - stability, 203–208
    - summary, 211–212
    - technical architecture, 208–211
  - macro/micro architecture
    - creating, 278–279
    - definition of, 277
    - domain architecture, 282
    - extent of, 279
    - operations and, 281
    - operations challenges, 239–240
    - responsibility for, 277–278
    - technologies, 280–281
    - testing, 282–284
  - of microservice-based systems
    - architecture management, 104–110
    - configuration, 139–141
    - documentation, 159–161
    - domain architecture, 100–104
    - EDA (event-driven architecture), 134–135
    - legacy application integration, 126–134
    - load balancing, 144–148
    - overview, 99
    - planning for growth, 118–125
    - scalability, 148–151
    - security, 151–159
    - service discovery, 141–143
    - summary, 161–162
    - technical architecture, 136–138
    - techniques to adjust architecture, 110–118
  - refactoring, 75
  - requirements and, 74–75
  - sample microservices-based application
    - build, 311–313
    - communication, 327–329
    - deployment with Docker, 313–314
    - Docker Compose, 321–324
    - Docker Machine, 320–321
    - domain architecture, 304–306
    - integration of other technologies, 335–336
    - JVM-based microservices in Amazon
      - Cloud, 338–340
    - load balancing, 333–334
    - overview, 303–304
    - resilience with Hystrix, 329–333
    - service discovery, 324–326
    - Spring Framework, 306–311
    - summary, 341–342
    - testing, 336–337
    - Vagrant, 314–320
  - SOA (service-oriented architecture)
    - characteristics of, 81–82
    - communication, 83–84
    - compared to microservices, 87–92
    - coordination, 86
    - interfaces, 85–86
    - introducing, 84
    - landscape, 82–83
    - orchestration, 86
    - services in, 84–85
    - technologies, 86–87
    - versioning, 85
  - sustainable architecture, 29
  - three-tier architecture, 71
- artifacts, 238–239
- asset servers, 166–167
- Asynchronous JavaScript and XML (AJAX), 170
- ATDD (acceptance test-driven design), 216
- ATOM Feeds, 183

- atomicity, 31
- authentication, 329
- authorization
  - OAuth2, 152–156
  - Spring Cloud Security, 329
- automation (deployment), 254–255
- AutoScaling, 339
- @Autowired, 334
- availability, 159
- avoiding
  - CDMs (canonical data models), 47–50
  - erosion of microservices, 122–123
- AWS (Amazon Web Services), 338
- Azure Service Bus, 183

## B

- back-end interfaces, 174–175
- bad architecture, causes of, 110–111
- balancing load. *See* load balancing
- batch data replication, 186
- BDD (behavior-driven design), 216
- BEAM (Bogdan/ Björn's Erlang Abstract Machine), 360
- behavior specification, 239
- behavior-driven design (BDD), 216
- benefits of microservices. *See* microservice advantages
- Berkeley Internet Name Domain Server (BIND), 142–143
- big systems, starting with, 119–120
- BigPipe, 170–171
- BIND (Berkeley Internet Name Domain Server), 142–143
- blue/green deployment, 220, 257
- blueprints (OSGi), 352
- BMC Remedy, 251
- Bogdan/ Björn's Erlang Abstract Machine (BEAM), 360
- bottlenecks during testing, 13
- Bounded Contexts
  - CDMs (canonical data models), 47–50
  - collaboration between, 44–45
  - Context Maps, 108–109
  - example, 42–44
  - large-scale structure, 46–47
  - microservices and, 45–46
  - nanoservices and, 347
  - overview, 52
  - sample microservices-based application, 305–306

- breaking up code, 126–127
- building sample microservices-based application, 311–313
- bulkhead, 204–205
- bundles, 350–353
- business advantages of microservices, 65–67

## C

- CA Opscenter, 251
- cache (REST), 176
- calling
  - Lambda functions, 348
  - Seneca functions, 363–365
- callMicroservice() function, 334
- canary releasing, 220, 257
- canonical data models (CDMs), avoiding, 47–50
- capacity tests, 59
- CatalogConsumerDrivenContractTest class, 337
- CDCs (consumer-driven contracts), 231
- CDMs (canonical data models), avoiding, 47–50
- central load balancers, 145
- certificates, 157
- challenges of microservices
  - architecture, 74–76
  - Conway's Law, 273–274
  - infrastructure, 76–77
  - operations
    - artifacts, 238–239
    - micro and macro architecture, 239–240
    - templates, 240
  - overview, 8
  - summary, 78
  - technical challenges
    - code dependencies, 71–72
    - overview, 69–71
    - technology pluralism, 73–74
    - unreliable communication, 73
- changes of perspective, DevOps and, 288–289
- changing architecture. *See* adjusting architecture
- Chef, 255
- circuit breaker design
  - Hystrix, 329
  - resilience and stability, 203–204
- classes
  - CatalogConsumerDrivenContractTest, 337
  - CustomerConsumerDrivenContractTest, 337
  - OrderTestApp, 336
  - SpringRestDataConfig, 304–305

- clients
  - client libraries, 292–293
  - client-based load balancing, 147–148
  - client-level integration, 190–191
  - Eureka, 324–325
  - mobile clients, 172–173
  - rich clients, 172–173
- cloud
  - Amazon Cloud
    - CloudWatch, 340, 349
    - JVM-based microservices in, 338–340
  - Spring Cloud, 308–310
  - Spring Cloud Netflix, 310–311
- Cloud Foundry, Spring Cloud for, 310
- CloudWatch, 340, 349
- cluster environments, 324
- CMS (content management systems), 130
- Coda Hale Metrics, 340
- code dependencies, 71–72
- code ownership, collective, 273–276
- code reuse, 114–115, 292–295
- code transfer, 113–114
- cohesion, 194
- collaboration between Bounded Contexts, 44–45
- collectd, 252–253
- collective code ownership, 273–276
- combined deployment, 258–259
- Command Query Responsibility Segregation. *See*
  - CQRS (Command Query Responsibility Segregation)
- commands. *See also* functions
  - docker kill, 319
  - docker log, 318
  - docker ps, 319
  - docker rm, 319
  - docker run, 334
  - docker-compose, 323, 334
  - docker-machine, 320–321
  - mvn package, 311
  - vagrant destroy, 319
  - vagrant halt, 319
  - vagrant provision, 319
  - vagrant ssh, 318
  - vagrant up, 319
- commit phase, 59
- Common Object Request Broker Architecture (CORBA), 70
- communication
  - data replication
    - batch, 186
    - consistency, 185–186
    - event, 186–187
    - implementation, 186
    - overview, 184–185
    - redundancy, 185–186
  - distributed communication, 28–29
    - between Docker containers, 263
    - between Erlang processes, 360–361
  - external interfaces, 187–190
  - internal interfaces, 187–190
  - levels of, 163
  - messaging
    - advantages of, 180–181
    - technologies, 182–184
    - transactions and, 181–182
  - REST (representational state transfer)
    - definition of, 175–179
    - load balancing, 144–145
    - RESTful-HTTP interfaces, 50
    - Spring Data REST, 307
  - sample microservices-based application, 305, 327–329
  - in SOA (service-oriented architecture), 83–84, 87
  - SOAP (Simple Object Access Protocol), 179–180
  - UIs (user interfaces)
    - HTML-based user interfaces, 168–175
    - SPAs (single-page apps), 164–168
  - unreliable communication, 73
- compromises, nanoservices and, 345
- confidentiality, 158
- configuration
  - Eureka, 325
  - Hystrix, 333
  - microservice-based systems, 139–141
  - operations, 256
  - Spring Cloud Config, 139, 309–310
  - Vagrant, 315–317
- conformist collaboration, 44
- consistency
  - data replication, 185–186
  - definition of, 31
  - problems with, 139–140
  - size of microservices and, 31–32
- Consul, 143
- consumer contracts, 231
- consumer-driven contract tests
  - contract types, 230–231
  - implementation, 231–232
  - overview, 230
  - sample microservices-based application, 337
  - tools, 232–233
- containers (Docker), 262–263, 318–319

content enricher, 128–129  
 content filters, 128  
 content management systems (CMS), 130  
 content-based routers, 127  
 Context Maps, 108–109  
 contexts, bounded. *See* Bounded Contexts  
 continuous delivery
 

- advantages of, 7–8
- challenges, 76–77
- legacy application modernization, 12
- operations
  - challenges of, 238–240
  - control, 259–260
  - deployment, 254–259
  - infrastructure, 260–266
  - logging, 241–246
  - monitoring, 246–253
  - summary, 266–267
- pipeline, 59–60, 67
- testing, 220

continuous deployment, 257  
 contract tests
 

- contract types, 230–231
- implementation, 231–232
- overview, 230

control of operations, 259–260  
 Conway, Melvin Edward, 35  
 Conway's Law
 

- challenges associated with, 273–274
- definition of, 35, 52, 273
- as enabler, 38–39
- as limitation, 36–38
- microservices and, 39–40
- reasons for, 36

coordination
 

- of deployments, 85–86
- microservice-based systems, 139–141
- pull requests for, 276
- SOA (service-oriented architecture), 86

CORBA (Common Object Request Broker Architecture), 70  
 CoreOS, 265, 324  
 correlating data with events, 249  
 correlation IDs, 245  
 CORS (cross-origin resource sharing), 166–167  
 CQRS (Command Query Responsibility Segregation)
 

- advantages of, 196–197
- challenges, 197
- microservices and, 196
- overview, 195–196

CRM (customer relationship management), 82–83, 88–90  
 cross-functional teams, 285–290  
 cross-origin resource sharing (CORS), 166–167  
 customer interface, 290–292  
 customer relationship management (CRM), 82–83, 88–90  
 CustomerConsumerDrivenContractTest class, 337  
 customer/supplier collaboration, 44  
 cycle-free software, 104–105  
 cyclic dependencies, 103–104

## D

dashboard (Hystrix), 331  
 data replication
 

- batch, 186
- consistency, 185–186
- event, 186–187
- implementation, 186
- overview, 129–130, 184–185, 191–192
- redundancy, 185–186

data security, 158–159  
 data storage for sample microservices-based application, 304–305  
 databases
 

- Amazon DynamoDB, 338
- HSQL database, 307, 319–320
- legacy application integration, 133–134

Datensparsamkeit, 157  
 DDD (domain-driven design)
 

- Bounded Contexts
  - CDMs (canonical data models), 47–50
  - collaboration between, 44–45
  - Context Maps, 108–109
  - example, 42–44
  - large-scale structure, 46–47
  - microservices and, 45–46
  - nanoservices and, 347
  - overview, 52
  - sample microservices-based application, 305–306
- building blocks, 41–42
- CDMs (canonical data models), avoiding, 47–50
- large-scale structure, 46–47
- overview, 40–41, 194
- strategic design, 100–101
- summary, 52
- ubiquitous language, 41

declarative services (OSGi), 352

- delegation to teams, 238–239
- delivery. *See* continuous delivery
- departments
  - architecture leading to, 291–292
  - microservice implementation and, 296
- dependencies
  - code dependencies, 71–72
  - dependency management
    - cyclic dependencies, 103–104
    - overview, 101–102
    - unintended domain-based dependencies, 102–103
  - hidden dependencies, 133–134
- deployment
  - automation, 254–255
  - combined versus separate, 258–259
  - coordination of, 85–86
  - deployment monoliths
    - migrating to microservices, 370–371
    - overview, 4
    - problems with, 121
    - technical coordination, 271
  - deployment strategies, 256–258
  - installation and configuration, 256
  - risk mitigation, 220–222, 256
  - sample microservices-based application, 313–314
  - separate deployment, 270–271
- deployment monoliths
  - migrating to microservices, 370–371
  - overview, 4
  - problems with, 121
  - technical coordination, 271
- desktop applications, nanoservices and, 346
- Developer Anarchy, 284
- development
  - signaling systems
    - challenges, 21–22
    - general description, 19–20
    - microservice advantages, 20–21, 22
    - suitability of microservices for, 22–23
  - sustainable development, 6, 57
  - test-driven development, 214–215
- DevOps, 285–290
- directories
  - docker-vagrant, 318
  - log-analysis, 341
- distributed communication, 28–29
- distributed systems
  - distributed communication, 28–29
  - distributed tracing, 245–246
  - distribution by domain. *See* DDD (domain-driven design) (domain-driven design)
    - overview, 8, 20
- DNS (Domain Name System), 142
- Docker
  - cluster environments, 324
  - commands
    - docker kill, 319
    - docker log, 318
    - docker ps, 319
    - docker rm, 319
    - docker run, 334
  - containers, 262–263, 318–319
  - Docker Compose, 321–324
  - Docker Machine, 320–321
  - file system, 261–263
  - overview, 255
  - registry, 264
  - sample microservices-based application
    - deployment, 313–314
    - servers, 264–265
- Docker Compose, 265, 321–324
- docker kill command, 319
- docker log command, 318
- Docker Machine, 265, 320–321
- docker ps command, 319
- docker rm command, 319
- docker run command, 334
- Docker Swarm, 265
- docker-compose command, 323, 334
- Dockerfile, 313–314
- docker-machine command, 320–321
- docker-vagrant directory, 318
- documentation
  - microservice-based system architecture, 159–161
  - Spring Guides, 311
  - testing as, 214–215
- domain architecture
  - importance of, 8
  - of individual microservices, 193–194
  - macro/micro architecture, 282
  - of microservice-based systems
    - dependency management, 101–104
    - Otto GmbH example, 101
    - overview, 100
    - strategic design, 100–101
  - sample microservices-based application, 304–306
- Domain Name System (DNS), 142
- domain-driven design. *See* DDD (domain-driven design)

dpkg, 255  
 durability, 32  
 dynamic environments, monitoring in, 250  
 dynamic scaling, 148–150  
 DynamoDB, 338

## E

EC2 (Elastic Computing Cloud), 349  
 Echo Verticles (Vert.x), 358–359  
 e-commerce legacy application scenario
 

- Amazon example, 18–19
- approach, 14
- challenges, 14–15
- feature development, 17–18
- general description, 11–12
- influence on organization, 18
- microservice advantages, 12–13, 17
- migration status, 15–16
- team creation, 16

 EDA (event-driven architecture), 134–135  
 Edge Side Includes (ESI), 171–172  
 EJB (Enterprise JavaBeans), 70  
 Elastic Beanstalk, 349  
 Elastic Computing Cloud (EC2), 349  
 Elastic Load Balancing (ELB), 146, 339  
 ElastiCache, 338  
 elasticity, 210  
 Elasticsearch, 242–244  
 ELB (Elastic Load Balancing), 146, 339  
 Elixir, 363  
 ELK (Elasticsearch, Logstash, Kibana) stack, 242–244  
 Ember.js, 165  
 @EnableCircuitBreaker, 330  
 @EnableDiscoveryClient, 324–325  
 @EnableEurekaClient, 324–325, 332  
 @EnableEurekaServer, 326  
 @EnableHystrix, 330  
 @EnableHystrixDashboard, 331, 332  
 enabler, Conway’s Law as, 38–39  
 @EnableTurbine, 332  
 @EnableZuulProxy, 328  
 enabling monitoring, 252  
 encapsulation, 194  
 engines, process, 208–209  
 Enterprise Integration Patterns, 127–129  
 Enterprise JavaBeans (EJB), 70  
 entities, 41  
 E-Post Development GmbH, 122  
 Erlang, 360–363  
 erosion of microservices, avoiding, 122–123

ESI (Edge Side Includes), 171–172  
 etcd, 139  
 Eureka
 

- client, 324–325
- configuration, 325
- overview, 143, 311, 324
- server, 326

 evaluation of nanoservices, 366–367  
 Evans, Eric, 40  
 event-driven architecture (EDA), 134–135  
 events
 

- correlating data with, 249
- data replication, 186–187
- EDA (event-driven architecture), 134–135
- event sourcing, 197–199

 evolving order, 47  
 example application. *See* sample microservices-based application  
 expenditures, minimizing with testing, 214  
 explorative tests, 59  
 Ext JS, 165  
 Extensible Markup Language (XML), 177–178  
 external interfaces, 85, 187–190

## F

Facebook BigPipe, 170–171  
 factories, 42  
 fail fast, 205  
 Fat JARs, 359  
 Feign, 311  
 file system (Docker), 261–263  
 files
 

- application.properties, 313
- application-test.properties, 229, 233
- Dockerfile, 313–314
- JAR (Java archive) files, 307, 354
- log analysis, 341
- MANIFEST.MF, 350–351
- pom.xml, 311–312
- Vagrantfile, 315–317
- WAR (web application archive), 307, 354

 filters
 

- content filters, 128
- message filters, 127
- Zuul filters, 329

 findAll() function, 330  
 firewalls, 157  
 First Rule of Distributed Object Design, 29  
 flexibility (SOA), 87–88  
 flexible transport (SOAP), 179–180  
 front-end servers, 171–172



functions. *See also* commands  
 callMicroservice(), 334  
 findAll(), 330  
 Lambda functions, 348  
 Seneca, 363–365

## G

Gentsch, Lars, 122–123  
 George, Fred, 284  
 Gephi, 107  
 global architecture, 122  
 Grafana, 250  
 grants, authorization, 153–154  
 Graphite, 250  
 Graylog, 244  
 growth, planning for  
   big systems, 119–120  
   deployment monoliths, 121  
   erosion of microservices, avoiding, 122–123  
   global architecture, 122  
   new functionality, incorporating, 123–125  
   overview, 118–122  
   replaceability, 121  
   small systems, 120–121

## H

HAL (Hypertext Application Language), 177  
 handshaking, 205  
 HAProxy, 146  
 Hashicorp Vault, 157–158  
 HATEOAS (Hypermedia as the Engine of Application State), 177  
 Heusingfeld, Alexander, 287  
 hexagonal architecture  
   example, 201–202  
   versus layered architecture, 200  
   microservices and, 201  
   overview, 199–200  
 hidden dependencies, 132–133  
 high cohesion, 102  
 high technological complexity, 21–22  
 HornetQ, 183  
 hosts, open, 45  
 HP Operations Manager, 251  
 HSQL database, 307, 319–320  
 HTML (Hypertext Markup Language)  
   HTML-based user interfaces  
     back end, 174–175  
     front-end servers, 171–172

JavaScript, 170–171  
 mobile clients, 172–173  
 organizational levels, 173  
 ROCA (resource-oriented client architecture), 168–169  
 routing, 169–170  
 overview, 178  
 HTTP (Hypertext Transfer Protocol)  
   httpd, 145  
   load balancing, 144–145  
   RESTful-HTTP interfaces, 50, 175–179  
 httpd, 145  
 hype versus reality, 371–372  
 Hypermedia as the Engine of Application State (HATEOAS), 177  
 Hypertext Application Language (HAL), 177  
 Hypertext Markup Language. *See* HTML (Hypertext Markup Language)  
 Hypertext Transfer Protocol. *See* HTTP (Hypertext Transfer Protocol)  
 Hypoport AG, 258  
 Hystrix  
   annotations, 330  
     @Autowired, 334  
     @EnableCircuitBreaker, 330  
     @EnableDiscoveryClient, 324–325  
     @EnableEurekaClient, 324–325, 332  
     @EnableEurekaServer, 326  
     @EnableHystrix, 330  
     @EnableHystrixDashboard, 331, 332  
     @EnableTurbine, 332  
     @EnableZuulProxy, 328  
     @HystrixCommand, 330  
     @RequestMapping, 307–308  
     @RestController, 307–308  
     @RibbonClient, 334  
     @SpringBootApplication, 307–308  
   circuit breaker, 329  
   configuration, 333  
   dashboard, 331  
   overview, 311, 329  
   resilience and stability, 207–208  
   Turbine, 331–332  
 @HystrixCommand, 330

## I

IBM Tivoli, 251  
 Icinga, 250  
 IDs, correlation IDs, 245  
 immutable servers, 140, 255

- implementation of microservices. *See also*
  - benefits of microservices; organizational effects of microservices
  - approaches, 370–371
  - consumer-driven contract tests, 231–232
  - data replication, 186
  - external interfaces, 188–189
  - hype versus reality, 371–372
  - microservice implementation without organizational changes, 295–297
  - sample microservices-based application
    - build, 311–313
    - communication, 327–329
    - deployment with Docker, 313–314
    - Docker Compose, 321–324
    - Docker Machine, 320–321
    - domain architecture, 304–306
    - integration of other technologies, 335–336
    - JVM-based microservices in Amazon Cloud, 338–340
    - load balancing, 333–334
    - overview, 303–304
    - resilience with Hystrix, 329–333
    - service discovery, 324–326
    - Spring Framework, 306–311
    - summary, 341–342
    - testing, 336–337
    - Vagrant, 314–320
  - summary, 372
  - without organizational changes, 295–297
- independence
  - independent scaling, 7, 61
  - technical independence, 63, 270–273
- infrastructure
  - challenges, 76–77
  - influence on microservice size, 30–31
  - legacy applications and, 131
  - overview, 260–261
  - PaaS (platform as a service), 266
  - virtualization with Docker, 261–265
- innoQ, 47, 287
- installation
  - deployments, 256
  - Docker Machine, 321
  - scripts, 254
  - tools, 140–141
  - Vagrant, 318
- installation scripts, 254
- installation tools, 140–141
- integration
  - data replication
    - batch, 186
    - consistency, 185–186
    - event, 186–187
    - implementation, 186
    - overview, 184–185
    - redundancy, 185–186
  - external interfaces, 187–190
  - integration platforms, 83
  - integration tests, 216, 226–227
  - internal interfaces, 187–190
  - legacy applications
    - breaking up code, 126–127
    - CMS (content management systems), 130
    - databases, 133–134
    - Enterprise Integration Patterns, 127–129
    - hidden dependencies, 132–133
    - infrastructure and, 131
    - limited integration, 129
    - overview, 57–58, 67, 126–127
    - UIs (user interfaces), 129–130
  - levels of, 163, 190–192
  - messaging
    - advantages of, 180–181
    - technologies, 182–184
    - transactions and, 181–182
  - of other systems, 21
  - REST (representational state transfer), 175–179
  - shared integration tests, 223–224
  - SOAP (Simple Object Access Protocol), 179–180
  - UIs (user interfaces)
    - HTML-based user interfaces, 168–175
    - SPAs (single-page apps), 164–168
  - integration tests, 216, 226–227
- integrity, 158
- interfaces
  - control via, 260
  - customer interface, 290–292
  - external interfaces, 187–190
  - internal interfaces, 187–190
  - Postel’s Law, 189–190
  - separating, 188
  - SOA (service-oriented architecture), 85–86
  - versioning, 192
- internal interfaces, 187–190
- introducing
  - microservices
    - approaches, 370–371
    - hype versus reality, 371–372
    - summary, 372
  - SOA (service-oriented architecture), 84

intrusion detection, 157  
isolation, 32

## J

JAR (Java archive) files, 307, 354

Java

bundles, 350–353

JAR (Java archive) files, 307, 354

Java EE

example, 356–357

microservices with, 355–356

nanoservices with, 355

overview, 354

JMS (Java Messaging Service), 183

JPA (Java Persistence API), 354

JRE (Java Runtime Environment), 307

JSF (Java ServerFaces), 354

JTA (Java Transaction API), 354

JVM-based microservices in Amazon  
Cloud, 73–74

OSGi, 350–353

Spring Framework

HSQL database, 307, 319–320

Spring Boot, 307–308

Spring Cloud, 308–310

Spring Cloud Netflix, 310–311

Spring Data REST, 307

Java EE

example, 356–357

microservices with, 355–356

nanoservices with, 355

overview, 354

Java Messaging Service (JMS), 183

Java Persistence API (JPA), 354

Java Runtime Environment (JRE), 307

Java ServerFaces (JSF), 354

Java Transaction API (JTA), 354

Java Virtual Machine (JVM), 73–74

JavaScript

HTML-based user interfaces, 170–171

JSON (JavaScript Object Notation)

JWE (JSON Web Encryption), 154–155

JWT (JSON Web Token), 154–155

overview, 178

Seneca, 363–365

JavaScript Object Notation. *See* JSON  
(JavaScript Object Notation)

JavaScript Object Notation (JSON), 178

Jetty, 338–339

JMS (Java Messaging Service), 183

JPA (Java Persistence API), 354

jQAssistant, 107

JRE (Java Runtime Environment), 307

JSF (Java ServerFaces), 354

JSON (JavaScript Object Notation)

JWE (JSON Web Encryption), 154–155

JWT (JSON Web Token), 154–155

overview, 178

JTA (Java Transaction API), 354

JUnit tests, 232

JVM (Java Virtual Machine), 73–74

JVM-based microservices in Amazon Cloud,  
338–340

JWE (JSON Web Encryption), 154–155

JWT (JSON Web Token), 154–155

## K

Kafka (Apache), 183, 338

Kerberos, 155

kernels, shared, 108

Kibana, 242–244

Kinesis, 338

Kubernetes, 265, 324

## L

Lambda (Amazon), 347–349

landscape (SOA), 82–83

languages

Elixir, 363

Erlang, 360–363

HAL (Hypertext Application Language), 177

HTML (Hypertext Markup Language), 178

Java

bundles, 350–353

Java EE, 354–357

OSGi, 350–353

published language, 45

ubiquitous language, 41

XML (Extensible Markup Language),  
177–178

large-scale structure, DDD (domain-driven  
design) and, 46–47

latency, 69–70

layered architecture, 200

leadership, technical, 284

leaseRenewalIntervalInSeconds property  
(Eureka), 325

legacy applications

development, 6

integration

breaking up code, 126–127

CMS (content management systems), 130

databases, 133–134

- Enterprise Integration Patterns, 127–129
  - hidden dependencies, 132–133
  - infrastructure and, 131
  - limited integration, 129
  - overview, 57–58, 67, 126–127
  - UIs (user interfaces), 129–130
  - modernization scenario
    - Amazon example, 18–19
    - approach, 14
    - challenges, 14–15
    - feature development, 17–18
    - general description, 11–12
    - influence on organization, 18
    - microservice advantages, 12–13, 17
    - migration status, 15–16
    - team creation, 16
  - levelized structure maps (LSMs), 104
  - levels
    - of independence, 271–273
    - of integration, 163, 190–192
  - libraries
    - client libraries, 292–293
    - Coda Hale Metrics, 340
    - shared libraries, 112–113
  - limitations, Conway’s Law as, 36–38
  - limited integration, 129
  - lines of code (LOC), 28, 343
  - Linux containers, 262–263
  - Linux servers, 264
  - load balancing
    - architecture and, 148
    - central load balancers, 145
    - client-based load balancing, 147–148
    - ELB (Elastic Load Balancing), 146, 339
    - load balancers (HTTP), 176
    - number of load balancers, 145
    - REST/HTTP, 144–145
    - sample microservices-based application, 333–334
    - service discovery, 146–147
    - technologies, 145–146
  - load tests, 217
  - LOC (lines of code), 28, 343
  - log analysis, 341
  - logging
    - correlation IDs, 245
    - ELK (Elasticsearch, Logstash, Kibana) stack, 242–244
    - Graylog, 244
    - log analysis, 341
    - for microservices, 241–242
    - Splunk, 244
    - stakeholders, 245
    - Zipkin, 245–246
  - logic layer integration, 191
  - Logstash, 242–244
  - loose coupling, 102
  - LSMs (levelized structure maps), 104
- ## M
- macro architecture
    - creating, 278–279
    - definition of, 277
    - domain architecture, 282
    - extent of, 279
    - operations and, 281
    - operations challenges, 239–240
    - responsibility for, 277–278
    - technologies, 280–281
    - testing, 282–284
  - MANIFEST.MF file, 350–351
  - manual tests, 217
  - maps
    - Context Maps, 108–109
    - LSMs (levelized structure maps), 104
  - Maven, 311–313
  - Mesos, 265, 324
  - Mesosphere, 324
  - message filters, 127
  - message routers, 127
  - message translators, 127
  - message-driven components, 210
  - messaging
    - advantages of, 180–181
    - message filters, 127
    - message routers, 127
    - message translators, 127
    - technologies, 182–184
    - transactions and, 181–182
  - metadata, 159–160
  - metrics, monitoring. *See* monitoring
  - Metrics framework, 252
  - micro architecture
    - creating, 278–279
    - definition of, 277
    - domain architecture, 282
    - operations challenges, 239–240, 281
    - responsibility for, 277–278
    - technologies, 280–281
    - testing, 282–284
  - microservice advantages
    - business advantages, 65–67
    - choice of technologies, 7

- microservice advantages (*continued*)
  - continuous delivery, 7–8
  - independent scaling, 7
  - legacy application development, 6
  - modularization, 5
  - organizational benefits, 63–65
  - overview, 4–5, 270
  - replaceability, 5–6
  - sustainable development, 6
  - technical benefits
    - choice of technologies, 7, 62–63
    - continuous delivery, 7–8, 59–60
    - independence, 63, 270–273
    - independent scaling, 7
    - legacy integration, 6, 57–58
    - microservice replacement, 56–57
    - modularization, 5
    - overview, 55–56
    - replaceability, 5–6
    - robustness, 61–62
    - scaling, 7, 61
    - sustainable development, 6, 57
    - time to market, 6–7
  - time to market, 6–7
- microservice challenges
  - architecture, 74–76
  - of Conway’s Law, 273–274
  - infrastructure, 76–77
  - operations
    - artifacts, 238–239
    - micro and macro architecture, 239–240
    - templates, 240
  - overview, 8
  - summary, 78
  - technical challenges
    - code dependencies, 71–72
    - overview, 69–71
    - technology pluralism, 73–74
    - unreliable communication, 73
- microservice definition, 3–4
- microservice-based systems
  - architecture management
    - Context Maps, 108–109
    - cycle-free software, 104–105
    - importance of, 107–108
    - tools for, 104–107
  - configuration, 139–141
  - documentation, 159–161
  - domain architecture
    - dependency management, 101–104
    - Otto GmbH example, 101
    - overview, 100
    - strategic design, 100–101
- EDA (event-driven architecture), 134–135
- legacy application integration
  - breaking up code, 126–127
  - CMS (content management systems), 130
  - databases, 133–134
  - Enterprise Integration Patterns, 127–129
  - hidden dependencies, 132–133
  - infrastructure and, 131
  - limited integration, 129
  - overview, 126–127
  - UIs (user interfaces), 129–130
- load balancing
  - architecture and, 148
  - central load balancers, 145
  - client-based load balancing, 147–148
  - number of load balancers, 145
  - REST/HTTP, 144–145
  - service discovery, 146–147
  - technologies, 145–146
- overview, 99
- planning for growth
  - big systems, 119–120
  - deployment monoliths, 121
  - erosion of microservices, avoiding, 122–123
  - global architecture, 122
  - new functionality, incorporating, 123–125
  - overview, 118–119
  - replaceability, 121
  - small systems, 120–121
- scalability
  - advantages of microservices, 150
  - dynamic scaling, 148–150
  - sharding, 150–151
  - throughput and response times, 151
- security
  - availability, 159
  - confidentiality, 158
  - data security, 158–159
  - Datensparsamkeit, 157
  - firewalls, 157
  - Hashicorp Vault, 157–158
  - integrity, 158
  - intrusion detection, 157
  - JWT (JSON Web Token), 154–155
  - Kerberos, 155
  - OAuth2, 152–156
  - overview, 151–152

- service discovery, 141–143
  - summary, 161–162
  - technical architecture, 136–138
  - techniques to adjust architecture
    - causes of bad architecture, 110–111
    - challenges with, 117–118
    - changes in microservices, 111
    - changes to overall architecture, 111–112
    - code reuse versus redundancy, 114–115
    - code transfer, 113–114
    - overview, 110
    - rewriting, 117
    - shared libraries, 112–113
    - shared services, 115–116
    - spawning new services, 116–117
  - middleware, uncoupling via, 206
  - migration status, 15–16
  - minimizing expenditures with testing, 214
  - minimum size of microservices, 344–345
  - mitigating risk, 220–222, 256
  - mobile clients, 172–173
  - mocks, 215–216
  - Moco, 229
  - mod\_include, 171
  - mod\_proxy\_balancer, 145
  - modernizing legacy applications
    - approach, 14
    - overview, 57–58
    - sample scenarios
      - challenges, 14–15
      - feature development, 17–18
      - general description, 11–12
      - influence on organization, 18
      - microservice advantages, 12–13, 17
      - migration status, 15–16
      - team creation, 16
  - modularization
    - influence on microservice size, 28
    - overview, 5
    - sample microservices-based application, 306
    - SPAs (single-page apps), 167–168
    - strong modularization, 67
  - Möllering, Sascha, 338–340
  - monitoring
    - additional metrics, 248
    - with Amazon CloudWatch, 340
    - basic information, 247
    - correlation with events, 249
    - dynamic environments, 250
    - enabling, 252
    - Erlang, 361
      - with Hystrix dashboard, 331
      - overview, 77, 246–247
      - stakeholders, 248–249
      - technologies for, 250–253
      - versus testing, 249
  - monoliths. *See* deployment monoliths
  - mountebank, 229
  - Müller, Jörg, 258–259
  - multiple services, splitting into, 116–117
  - multiple single-page apps. *See* SPAs (single-page apps)
  - mvn package command, 311
- ## N
- Nagios, 250
  - nanoservices
    - advantages of, 344–346
    - Amazon Lambda, 347–349
    - definition of, 346–347
    - Erlang, 360–363
    - Java EE
      - microservices with, 355–356
      - nanoservices with, 355
      - overview, 354, 355
    - OSGi, 350–353
    - overview, 343
    - Seneca, 363–365
    - technical evaluation, 366–367
    - Vert.x, 357–359
  - Netflix, 310–311, 371
  - networking, Vagrant, 317
  - NewRelic, 251
  - nginx, 145, 171
  - ngx\_http\_ssi\_module, 171
  - Node.js, 363–364
  - NUMMI car factory, 279
- ## O
- OAM (operations and maintenance support), 362
  - OAuth2, 152–156
  - open host services, 45
  - open source projects, 293–295
  - Open Telecom Platform (OTP) framework, 360
  - operations
    - challenges
      - artifacts, 238–239
      - micro and macro architecture, 239–240
      - overview, 76–77
      - templates, 240
    - control, 259–260

operations (*continued*)

- deployment
  - automation, 254–255
  - combined versus separate, 258–259
  - deployment strategies, 256–258
  - installation and configuration, 256
  - risks associated with, 256
- DevOps, 285–290
- infrastructure
  - overview, 260–261
  - PaaS (platform as a service), 266
  - virtualization with Docker, 261–265
- logging
  - correlation IDs, 245
  - ELK (Elasticsearch, Logstash, Kibana) stack, 242–244
  - Graylog, 244
  - for microservices, 241–242
  - Splunk, 244
  - stakeholders, 245
  - Zipkin, 245–246
- macro/micro architecture, 281
- microservice implementation and, 296–297
- monitoring
  - additional metrics, 248
  - basic information, 247
  - correlation with events, 249
  - dynamic environments, 250
  - enabling, 252
  - overview, 246–247
  - stakeholders, 248–249
  - technologies for, 250–253
  - versus testing, 249
- summary, 266–267
- operations and maintenance support (OAM), 362
- Operations Manager, 251
- Opscenter, 251
- orchestration (SOA), 83, 86–87
- order systems, 83
- OrderTestApp class, 336
- organizational effects of microservices
  - benefits of microservices, 63–65, 270–273
  - collective code ownership, 273–276
  - customer interface, 290–292
  - DevOps, 285–290
  - implementation without organizational changes, 295–297
  - macro/micro architecture
    - creating, 278–279
    - definition of, 277
    - domain architecture, 282
    - extent of, 279

- operations and, 281
- responsibility for, 277–278
- technologies, 280–281
- testing, 282–284
- overview, 18, 269
- reusable code, 292–295
- summary, 297–299
- technical leadership, 284
- organizational-level interfaces, 173
- OSGi, 350–353
- OTP (Open Telecom Platform) framework, 360
- Otto GmbH, 101
- outdated documentation, 160
- ownership, collective code ownership, 273–276

## P

- PaaS (platform as a service), 266
- package managers, 255
- Packetbeat, 251
- Pact, 233
- pact-jvm, 233
- Pacto, 233
- parallel work, 12, 65–67
- personification of servers, avoiding, 287–288
- perspectives, DevOps and, 288–289
- “pets vs. cattle” slogan (DevOps), 287–288
- pipeline, continuous delivery, 59–60, 67
- planning for growth
  - big systems, 119–120
  - deployment monoliths, 121
  - erosion of microservices, avoiding, 122–123
  - global architecture, 122
  - new functionality, incorporating, 123–125
  - overview, 118–119
  - replaceability, 121
  - small systems, 120–121
- platform as a service (PaaS), 266
- Play, 210
- pluralism, 73–74
- pom.xml files, 311–312
- portals (SOA), 83
- Postel’s Law, 189–190
- Prana, 336
- preferIpAddress property (Eureka), 325
- process engines, 208–209
- programs. *See* tools
- projects
  - open source projects, 293–295
  - project size, 65
- Protocol Buffer, 178
- provider contracts, 230

proxy-based load balancing, 144–145  
 published language, 45  
 pull requests for coordination, 276  
 Puppet, 255

## Q-R

Qpid (Apache), 182  
 RabbitMQ, 182  
 Reactive Manifesto, 207, 209–210  
 reactive systems, 209–210  
 reasons for microservices. *See* benefits of microservices  
 redundancy, 114–115, 185–186  
 refactoring, 8, 29–30, 75  
 reference environments, 228  
 registry (Docker), 264  
*Release It!*, 62, 203  
 Remedy, 251  
 remote procedure call (RPC), 179  
 replaceability
 

- advantages of, 56–57
- influence on microservice size, 31
- overview, 5–6, 67
- system planning, 121

 replicating data
 

- batch, 186
- consistency, 185–186
- event, 186–187
- implementation, 186
- overview, 129–130, 184–185, 191–192
- redundancy, 185–186

 repositories, 42  
 representational state transfer. *See* REST (representational state transfer)  
 @RequestMapping, 307–308  
 requests, pull, 276  
 resilience
 

- bulkhead, 204–205
- circuit breaker design, 203–204
- fail fast, 205
- handshaking, 205
- Hystrix, 207–208
- overview, 203, 209
- Reactive Manifesto, 207
- sample microservices-based application
  - annotations, 330
  - circuit breaker, 329
  - configuration, 333
  - dashboard, 331
  - overview, 329
  - Turbine, 331–332

- steady state, 205
- test harnesses, 206
- timeouts, 203
- uncoupling via middleware, 206

 resource-oriented client architecture (ROCA), 168–169  
 response times, 151  
 responsibility layers, 46–47  
 responsiveness, 209  
 REST (representational state transfer)
 

- definition of, 175–179
- load balancing, 144–145
- RESTful-HTTP interfaces, 50
- Spring Data REST, 307

 @RestController, 307–308  
 reuse of code, 114–115, 292–295  
 rewriting services, 117  
 Ribbon, 310, 333–334  
 @RibbonClient, 334  
 rich clients, 172–173  
 Riemann, 250  
 risk mitigation, 220–222, 256  
 robustness, 61–62  
 Robustness Principle, 189–190  
 ROCA (resource-oriented client architecture), 168–169  
 roll forward, 257  
 rollbacks, 256  
 routers, 127–129  
 routing
 

- Enterprise Integration Patterns, 127–129
- HTML-based user interfaces, 169–170
- Zuul, 327–328

 RPC (remote procedure call), 179  
 rpm, 255  
 RSpec, 232–233  
 RxJava, 210  
 RxJS, 210

## S

Salt, 255  
 SAML, 155  
 sample microservices-based application
 

- build, 311–313
- communication, 327–329
- deployment with Docker, 313–314
- Docker Compose, 321–324
- Docker Machine, 320–321
- domain architecture, 304–306
- integration of other technologies, 335–336



- sample microservices-based application
  - (*continued*)
  - JVM-based microservices in Amazon Cloud, 338–340
  - load balancing, 333–334
  - overview, 303–304
  - resilience with Hystrix
    - annotations, 330
    - circuit breaker, 329
    - configuration, 333
    - dashboard, 331
    - overview, 329
    - Turbine, 331–332
  - service discovery, 324–326
  - Spring Framework
    - HSQL database, 307
    - Spring Boot, 307–308
    - Spring Cloud, 308–310
    - Spring Cloud Netflix, 310–311
    - Spring Data REST, 307, 319–320
  - summary, 341–342
  - testing
    - consumer-driven contract tests, 337
    - overview, 336
    - stubs, 336–337
  - Vagrant
    - configuration, 315–317
    - how it works, 314
    - installation, 318
    - networking, 317
- Scala, 64, 210
- scalability
  - advantages of microservices, 150
  - AutoScaling, 339
  - dynamic scaling, 148–150
  - ELK (Elasticsearch, Logstash, Kibana) stack, 243–244
  - independent scaling, 7, 61, 67
  - sharding, 150–151
  - throughput and response times, 151
- scenarios
  - legacy application modernization
    - approach, 14
    - challenges, 14–15
    - feature development, 17–18
    - general description, 11–12
    - influence on organization, 18
    - microservice advantages, 12–13, 17
    - migration status, 15–16
    - team creation, 16, 18–19
  - signaling systems
    - challenges, 21–22
    - general description, 19–20
    - microservice advantages, 20–21, 22
    - suitability of microservices for, 22–23
- scripts, installation, 254
- SCS (self-contained system), 51
- security
  - API keys, 157
  - availability, 159
  - certificates, 157
  - confidentiality, 158
  - data security, 158–159
  - Datensparsamkeit, 157
  - firewalls, 155
  - Hashicorp Vault, 157–158
  - integrity, 158
  - intrusion detection, 157
  - JWT (JSON Web Token), 154–155
  - Kerberos, 155
  - OAuth2, 152–156
  - overview, 151–152
  - SAML, 155
  - Spring Cloud Security, 309
  - SSL/TLS, 155
- self-contained system (SCS), 51
- Semantic Versioning, 189
- Seneca, 363–365
  - seneca.act() function, 364
  - seneca.add() function, 364
  - seneca.listen() function, 364
- separate deployment, 258–259
- separate ways, 45
- separating interfaces, 188
- Server Side Includes (SSI), 171–172
- servers
  - asset servers, 166–167
  - Docker and, 264–265
  - Eureka, 326
  - front-end servers, 171–172
  - immutable servers, 140, 255
  - “pets vs. cattle” slogan (DevOps), 287–288
  - Zuul, 327–329
- Serverspec, 235
- service discovery
  - BIND (Berkeley Internet Name Domain Server), 142–143
  - Consul, 143
  - DNS (Domain Name System), 142
  - Eureka
    - client, 324–325
    - configuration, 325
    - overview, 143, 324
    - server, 326
  - load balancing, 146–147
  - overview, 141–142

- service-oriented architecture. *See* SOA (service-oriented architecture)
- Seyren, 250
- sharding, 150–151
- shared integration tests, 223–224
- shared kernels, 44, 108
- shared libraries, 112–113, 292–293
- shared services, 115–116
- signaling system development
  - challenges, 21–22
  - general description, 19–20
  - microservice advantages, 20–21, 22
  - suitability of microservices for, 22–23
- Simple Notification Service (SNS), 348
- Simple Object Access Protocol (SOAP), 179–180
- Simple Queue Service (SQS), 348
- single-page apps. *See* SPAs (single-page apps)
- size of microservices
  - factors influencing
    - consistency, 31–32
    - distributed communication, 28–29
    - infrastructure, 30–31
    - LOC (lines of code), 28
    - modularization, 28
    - overview, 27–28, 33–35
    - refactoring, 29–30
    - replaceability, 31
    - sustainable architecture, 29
    - team size, 30
    - transactions, 31–33
  - overview, 344–345
  - project size, 65
  - summary, 52
- small systems, starting with, 120–121
- SNS (Simple Notification Service), 348
- SOA (service-oriented architecture)
  - characteristics of, 81–82
  - communication, 83–84
  - compared to microservices
    - architecture, 88–90
    - communication, 87
    - flexibility, 87–88
    - orchestration, 87
    - synergies, 91
    - table of differences, 91
  - coordination, 86
  - interfaces, 85–86
  - introducing, 84
  - landscape, 82–83
  - orchestration, 86
  - services in, 84–85
  - technologies, 86–87
  - versioning, 85
- SOAP (Simple Object Access Protocol), 179–180
- software development
  - sustainable software development, 57, 67
  - test-driven development, 214–215
- SPAs (single-page apps)
  - asset servers, 166–167
  - SPA modules, 167–168
  - SPA per microservice, 165–166
  - technologies, 164–165
- spigo, 110
- Splunk, 244
- Spring Boot, 307–308
- Spring Cloud, 308–310
- Spring Cloud Bus, 310
- Spring Cloud Cluster, 310
- Spring Cloud Config, 139, 309–310
- Spring Cloud Connectors, 310
- Spring Cloud Consul, 310
- Spring Cloud Data Flow, 310
- Spring Cloud Data Tasks, 310
- Spring Cloud for Cloud Foundry, 310
- Spring Cloud Netflix, 310–311
- Spring Cloud Security, 309
- Spring Cloud Sleuth, 310
- Spring Cloud Stream, 310
- Spring Cloud Zookeeper, 310
- Spring Data REST, 307
- Spring Framework
  - HSQL database, 307, 319–320
  - Spring Boot, 307–308
  - Spring Cloud, 308–310
  - Spring Cloud Netflix, 310–311
  - Spring Data REST, 307
- Spring Guides, 311
- @SpringBootApplication, 307–308
- SpringRestDataConfig class, 304–305
- SQS (Simple Queue Service), 348
- Squid, 171
- SSI (Server Side Includes), 171–172
- SSL/TLS, 155
- stability
  - bulkhead, 204–205
  - circuit breaker design, 203–204
  - fail fast, 205
  - handshaking, 205
  - Hystrix, 207–208
  - microservices and, 206–207
  - overview, 203
  - Reactive Manifesto, 207
  - steady state, 205
  - test harnesses, 206
  - timeouts, 203
  - uncoupling via middleware, 206

- stakeholders
    - for logs, 245
    - for monitoring, 248–249
  - statelessness, 209
  - StatsD, 252
  - status of migration, 15–16
  - steady state, 205
  - strategic design, 100–101
  - strategies (deployment), 256–258
  - strong modularization, 67
  - Structure 101, 107
  - stubby4j, 229
  - stubs, 216, 228–229, 336–337
  - sustainable architecture, 29
  - sustainable development, 6, 57, 67
  - synchronous RESTful HTTP, 179
  - synergies between SOA and microservices, 91
  - system architecture
    - architecture management
      - Context Maps, 108–109
      - cycle-free software, 104–105
      - importance of, 107–108
      - tools for, 104–107
    - configuration, 139–141
    - documentation, 159–161
    - domain architecture
      - dependency management, 101–104
      - Otto GmbH example, 101
      - overview, 100
      - strategic design, 100–101
    - EDA (event-driven architecture), 134–135
    - legacy application integration
      - breaking up code, 126–127
      - CMS (content management systems), 130
      - databases, 133–134
      - Enterprise Integration Patterns, 127–129
      - hidden dependencies, 132–133
      - infrastructure and, 131
      - limited integration, 129
      - overview, 126–127
      - UIs (user interfaces), 129–130
    - load balancing
      - architecture and, 148
      - central load balancers, 145
      - client-based load balancing, 147–148
      - number of load balancers, 145
      - REST/HTTP, 144–145
      - service discovery, 146–147
      - technologies, 145–146
    - overview, 99
    - planning for growth
      - big systems, 119–120
      - deployment monoliths, 121
      - erosion of microservices, avoiding, 122–123
      - global architecture, 122
      - new functionality, incorporating, 123–125
      - overview, 118–119
      - replaceability, 121
      - small systems, 120–121
    - scalability
      - advantages of microservices, 150
      - dynamic scaling, 148–150
      - sharding, 150–151
      - throughput and response times, 151
    - security
      - API keys, 157
      - availability, 159
      - certificates, 157
      - confidentiality, 158
      - data security, 158–159
      - Datensparsamkeit, 157
      - firewalls, 157
      - Hashicorp Vault, 157–158
      - integrity, 158
      - intrusion detection, 157
      - JWT (JSON Web Token), 154–155
      - Kerberos, 155
      - OAuth2, 152–156
      - overview, 151–152
      - SAML, 155
      - SSL/TLS, 156
    - service discovery, 141–143
    - summary, 161–162
    - technical architecture, 136–138
    - techniques to adjust architecture
      - causes of bad architecture, 110–111
      - challenges with, 117–118
      - changes in microservices, 111
      - changes to overall architecture, 111–112
      - code reuse versus redundancy, 114–115
      - code transfer, 113–114
      - overview, 110
      - rewriting, 117
      - shared libraries, 112–113
      - shared services, 115–116
      - spawning new services, 116–117
  - system metaphor, 46
  - system testing, 222–224
- ## T
- teams
    - creating, 16
    - delegating to, 238–239
    - DevOps, 285–290

- size of, 30
- us versus them attitude, 288
- technical architecture, 136–138
- technical benefits
  - choice of technologies, 62–63
  - continuous delivery, 59–60
  - independence, 63, 270–273
  - legacy integration, 57–58
  - microservice replacement, 56–57
  - overview, 55–56
  - robustness, 61–62
  - scaling, 61
  - sustainable software development, 57
- technical challenges
  - code dependencies, 71–72
  - overview, 69–71
  - technology pluralism, 73–74
  - unreliable communication, 73
- technical evaluation of nanoservices, 366–367
- technical leadership, 284
- technical standards, testing, 233–235
- technologies
  - choice of, 7, 62–63
  - high technological complexity, 21–22
  - load balancing, 145–146
  - messaging, 182–184
  - monitoring tools, 250–253
  - sample microservices-based application
    - HSQL database, 307
    - Spring Boot, 307–308
    - Spring Cloud, 308–310
    - Spring Cloud Netflix, 310–311
    - Spring Data REST, 307, 319–320
  - for service discovery, 142–143
  - SOA (service-oriented architecture), 86–87
  - technology stack per team, 21
- technology pluralism, 73–74
- templates, 240
- test harnesses, 206
- test pyramid
  - for individual services, 217–219
  - for systems, 222
- test-driven development, 214–215
- testing
  - arbitrary test frameworks, 232
  - benefits of, 213–215
  - bottlenecks during, 13
  - continuous delivery pipeline, 220
  - individual services, 227–229
  - legacy applications, 225–227
  - macro/micro architecture, 282–284
  - versus monitoring, 249
  - risk mitigation, 220–222
- sample microservices-based application
  - consumer-driven contract tests, 337
  - overview, 336
  - stubs, 336–337
- summary, 235–236
- system testing, 222–224
- technical standards, 233–235
- test pyramid
  - for individual services, 217–219
  - for systems, 222
- test types
  - acceptance tests, 59
  - capacity tests, 59
  - consumer-driven contract tests, 230–233, 337
  - exploratory tests, 59
  - integration tests, 216, 226–227
  - JUnit tests, 232
  - load tests, 217
  - manual tests, 217
  - shared integration tests, 223–224
  - UI tests, 216–217
  - unit tests, 215–216
- three-tier architecture, 71
- Thrift, 180
- throughput, 151
- Tilkov, Stefan, 47–50
- time to market, 6–7
- timeouts, 203
- Tivoli, 251
- Tolerant Reader concept, 190
- tools
  - 0MQ, 183
  - ActiveMQ, 183
  - Ansible, 255
  - Apache Kafka, 183
  - Apache Mesos, 265
  - Apache Qpid, 182
  - Apache Thrift, 180
  - Apache Zookeeper, 139
  - architecture management, 104–107
  - ATOM Feeds, 183
  - Azure Service Bus, 183
  - BIND (Berkeley Internet Name Domain Server), 142–143
  - Chef, 255
  - collectd, 252–253
  - Consul, 143
  - CoreOS, 265, 324
  - DNS (Domain Name System), 142
  - Docker
    - cluster environments, 324
    - containers, 262–263, 318–319

- tools (*continued*)
    - file system, 261–263
    - overview, 255
    - registry, 264
    - sample microservices-based application
      - deployment, 313–314
      - servers, 264–265
    - Docker Compose, 265, 321–324
    - Docker Machine, 265, 320–321
    - Docker Swarm, 265
    - etcd, 139
    - Eureka
      - client, 324–325
      - configuration, 325
      - overview, 143, 311, 324
      - server, 326
    - Feign, 311
    - Gephi, 107
    - Grafana, 250
    - Graphite, 250
    - HAProxy, 146
    - Hashicorp Vault, 157–158
    - HornetQ, 183
    - httpd, 145
    - Hystrix
      - annotations, 330
      - circuit breaker, 329
      - configuration, 333
      - dashboard, 331
      - overview, 311, 329
      - resilience and stability, 207–208
      - Turbine, 331–332
    - Icinga, 250
    - jQAssistant, 107
    - Kerberos, 155
    - Kubernetes, 265, 324
    - Maven, 311–313
    - Mesos, 324
    - Mesosphere, 324
    - Metrics framework, 252
    - Moco, 229
    - mountebank, 229
    - Nagios, 250
    - nginx, 145
    - OAuth2, 152–156
    - Operations Manager, 251
    - Opscenter, 251
    - package managers, 255
    - Packetbeat, 251
    - Pact, 233
    - Pacto, 233
    - Puppet, 255
    - RabbitMQ, 182
    - Remedy, 251
    - Ribbon, 310, 333–334
    - Riemann, 250
    - RSpec, 232–233
    - Salt, 255
    - SAML, 155
    - Serverspec, 235
    - Seyren, 250
    - Spring Framework
      - HSQL database, 307, 319–320
      - Spring Boot, 307–308
      - Spring Cloud, 308–310
      - Spring Cloud Config, 139
      - Spring Cloud Netflix, 310–311
      - Spring Data REST, 307
    - Squid, 171
    - SSL/TLS, 155
    - StatsD, 252
    - Structure 101, 107
    - stubby4j, 229
    - Tivoli, 251
    - Turbine, 311
    - uniform tools, 239
    - Vagrant
      - configuration, 315–317
      - how it works, 314
      - installation, 318
      - networking, 317
    - Varnish, 171
    - VirtualBox, 318
    - WireMock, 229
    - Zuul, 310, 327–329
  - tracing, 245–246, 310
  - transactions
    - influence on microservice size, 31–33
    - messages and, 181–182
  - transferring code, 113–114
  - TSL, 155
  - Turbine, 331–332
- ## U
- ubiquitous language, 41
  - UIs (user interfaces)
    - integration
      - HTML-based user interfaces, 168–175
      - overview, 129–130
      - SPAs (single-page apps), 164–168
      - overview, 47–50, 129–130

- SCS (self-contained system), 51
- technical alternatives, 50
- UI tests, 216–217
- uncoupling via middleware, 206
- uniformity
  - uniform tools, 239
  - via asset servers, 166–167, 311
- unintended domain-based dependencies, 102–103
- unit tests, 215–216
- UNIX, 3
- unreliable communication, 73
- updating Docker containers, 319
- “us versus them” attitude, 288
- user interfaces. *See* UIs (user interfaces)

## V

- Vagrant
  - configuration, 315–317
  - how it works, 314
  - installation, 318
  - networking, 317
- vagrant destroy command, 319
- vagrant halt command, 319
- vagrant provision command, 319
- vagrant ssh command, 318
- vagrant up command, 319
- Vagrantfile, 315–317
- value objects, 41
- Varnish, 171

- v.cpus setting (Vagrantfile), 318
- versioning
  - interfaces, 192
  - Semantic Versioning, 189
  - SOA (service-oriented architecture), 85
  - version control, 77
- Verticles (Vert.x), 358–359
- Vert.x, 210, 357–359
- viewing Docker containers, 318–319
- virtual machines, 260
- VirtualBox, 318
- virtualization (Docker), 261–265
- v.memory setting (Vagrantfile), 318

## W

- WAR (web application archive), 258, 307, 354
- web integration. *See* UIs (user interfaces)
- Wehrens, Oliver, 133–134
- WireMock, 229

## X-Y-Z

- XML (Extensible Markup Language), 177–178
- zanox AG, 338
- ZeroMQ, 183
- Zipkin, 245–246
- ZMQ, 183
- Zookeeper (Apache), 139, 310
- Zuul, 310, 327–329
- zypper, 255