

Microservices from Theory to Practice

Creating Applications in IBM Bluemix Using the Microservices Approach

Shahir Daya

Nguyen Van Duy

Kameswara Eati

Carlos M Ferreira

Dejan Glozic

Vasfi Gucer

Manav Gupta

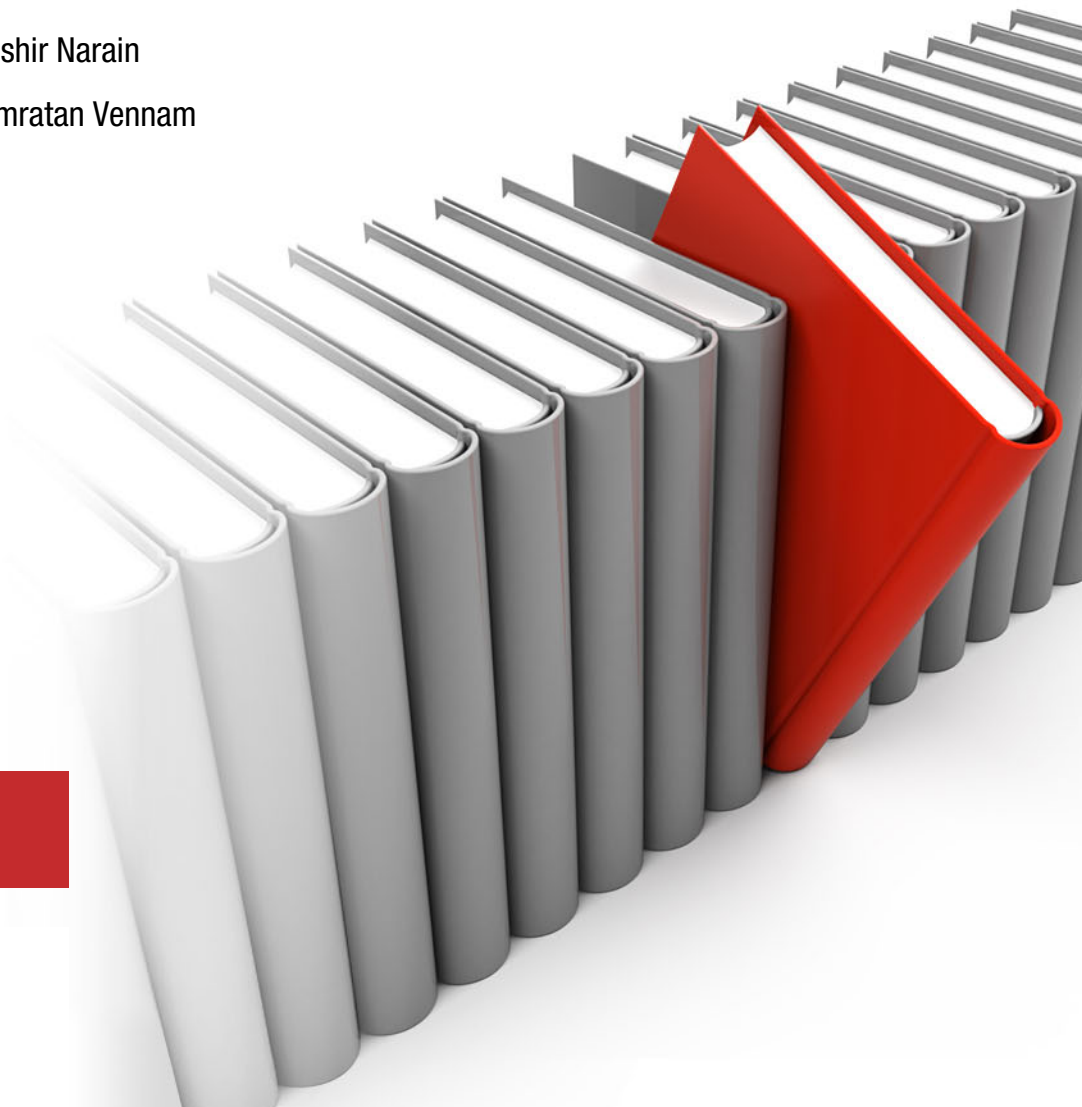
Sunil Joshi

Valerie Lampkin

Marcelo Martins

Shishir Narain

Ramratan Vennam



 **Cloud**



International Technical Support Organization

**Microservices from Theory to Practice: Creating
Applications in IBM Bluemix Using the Microservices
Approach**

August 2015

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

First Edition (August 2015)

This edition applies to IBM Bluemix Version 1.0.

© Copyright International Business Machines Corporation 2015. All rights reserved.

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

Contents

Notices	vii
Trademarks	viii
IBM Redbooks promotions	ix
Preface	xi
Authors	xi
Now you can become a published author, too	xv
Comments welcome	xv
Stay connected to IBM Redbooks	xvi
Part 1. Introducing the microservices architectural style	1
Chapter 1. Motivations for microservices	3
1.1 What are microservices	4
1.1.1 Small and focused	5
1.1.2 Loosely coupled	5
1.1.3 Language-neutral	5
1.1.4 Bounded context	5
1.1.5 Comparing microservices and monolithic architectures	6
1.2 Benefits from microservices	6
1.2.1 Enterprise solutions context	6
1.2.2 Challenges with monolithic architecture	7
1.2.3 Developer perspective	7
1.2.4 Tester perspective	8
1.2.5 Business owner perspective	8
1.2.6 Service management perspective	9
1.3 What to avoid with microservices	10
1.3.1 Don't start with microservices	11
1.3.2 Don't even think about microservices without DevOps	11
1.3.3 Don't manage your own infrastructure	11
1.3.4 Don't create too many microservices	11
1.3.5 Don't forget to keep an eye on the potential latency issue	12
1.4 How is this different than service-oriented architecture?	12
1.5 Case studies and most common architectural patterns	14
1.5.1 An e-commerce discount site	15
1.5.2 Financial services company	15
1.5.3 Large brick-and-mortar retailer	16
1.6 Example scenarios using microservices	17
1.6.1 Cloud Trader	17
1.6.2 Online Store	17
1.6.3 Acme Air	18
Chapter 2. Elements of a microservices architecture	19
2.1 Characteristics of microservices architecture	20
2.1.1 Business-oriented	20
2.1.2 Design for failure	21
2.1.3 Decentralized data management	23
2.1.4 Discoverability	24

2.1.5	Inter-service communication design	24
2.1.6	Dealing with complexity.	27
2.1.7	Evolutionary design.	28
2.2	Designing microservices	29
2.2.1	Use design thinking to scope and identify microservices	29
2.2.2	Choosing the implementation stack	31
2.2.3	Sizing the microservices	32
2.3	REST API and messaging	33
2.3.1	REST	33
2.3.2	Messaging.	34
2.3.3	REST and messaging together.	37
2.4	The future of microservices.	38
Chapter 3. Microservices and DevOps		39
3.1	Why you should use DevOps	40
3.1.1	Defining DevOps	40
3.1.2	DevOps is a prerequisite to successfully adopting microservices.	40
3.1.3	Organizing teams to support microservices	40
3.1.4	Organize a DevOps team to support other microservices teams	41
3.2	DevOps capabilities for microservices architecture.	41
3.2.1	Continuous business planning	43
3.2.2	Continuous integration and collaborative development	44
3.2.3	Continuous testing	44
3.2.4	Continuous release and deployment	45
3.2.5	Continuous monitoring	46
3.2.6	Continuous customer feedback and optimization	47
3.3	Microservices governance	47
3.3.1	Centralized versus decentralized governance	48
3.3.2	Enterprise transformation for microservices	50
3.4	DevOps capabilities: Testing strategies for microservices	51
3.4.1	Considerable testing methods	51
3.4.2	Building a sufficient testing strategy	55
Chapter 4. Developing microservices in Bluemix		57
4.1	Bluemix introduction	58
4.1.1	Bluemix deployment models	59
4.1.2	Bluemix Cloud Foundry concepts	63
4.1.3	How Bluemix works.	66
4.2	Developing microservices using Bluemix DevOps	68
4.2.1	Bluemix DevOps introduction	69
4.2.2	Delivery pipeline	74
4.3	Deployment, testing, monitoring, and scaling services in Bluemix DevOps	76
4.3.1	Deployment services in Bluemix	76
4.3.2	Testing services in Bluemix DevOps.	78
4.3.3	Monitoring and analytics services in Bluemix	79
4.3.4	Scaling in Bluemix.	79
4.4	Communication, session persistence, and logging in Bluemix	81
4.4.1	Communication	81
4.4.2	Session persistence	83
4.4.3	Logging in Bluemix	85
4.5	Considerations and opportunities	86
4.5.1	Microservice trends to consider	86
4.5.2	Controlling access and visibility of endpoints	87

4.5.3	Avoiding failures	88
4.5.4	Versioning	89
	Chapter 5. Microservices case studies in IBM	91
5.1	Microservices implementation in IBM DevOps Services	92
5.1.1	North Star Architecture	93
5.1.2	Delivery pipeline	95
5.1.3	REST and MQTT mirroring	95
5.1.4	Deployment considerations	96
5.2	Microservices case study in Bluemix console	96
5.2.1	Assessing the task	97
5.2.2	Peeling off Landing and Solutions	98
5.2.3	Composition of the common areas	99
5.2.4	“Lather, rinse, repeat”	100
5.2.5	Deployment considerations	102
5.3	Microservices case study in IBM Watson services	103
5.3.1	IBM Watson Developer Cloud	104
5.3.2	IaaS++ platform (Cloud Services Fabric)	105
5.3.3	Main components of IaaS++	106
5.3.4	IBM Watson Developer Cloud services	107
	Part 2. Example scenarios using the microservices approach	113
	Chapter 6. Scenario 1: Transforming a monolithic application to use microservices (CloudTrader)	115
6.1	Introduction to the sample	116
6.2	Refactoring the application	117
6.3	The CloudTraderAccountMSA microservice	118
6.4	The CloudTraderQuoteMSA microservice	118
6.4.1	DevOps Services	119
	Chapter 7. Scenario 2: Microservices built on Bluemix	123
7.1	Introduction to the sample	124
7.2	Breaking it down	124
7.2.1	Online Store Catalog service	125
7.2.2	Orders	126
7.2.3	Shipping	127
7.2.4	UI	128
7.3	Additional Bluemix services	129
7.3.1	DevOps Services	129
7.3.2	Monitoring	129
7.3.3	Scaling	130
	Chapter 8. Scenario 3: Modifying the Acme Air application and adding fault tolerance capabilities	131
8.1	The original Acme Air monolithic application	132
8.1.1	Deploying the Acme Air application in monolithic mode	133
8.2	Redesigning application to use microservices	139
8.2.1	Deploying the Acme Air application in microservices mode	140
8.3	Adding Hystrix to monitor services	142
8.3.1	Deploying the Hystrix Dashboard to Bluemix	143

Related publications	147
IBM Redbooks	147
Other publications	147
Online resources	147
Help from IBM	148

Notices

This information was developed for products and services offered in the U.S.A.

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

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

IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

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

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

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

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

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

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


COPYRIGHT LICENSE:

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

Trademarks

IBM, the IBM logo, and [ibm.com](http://www.ibm.com) are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at <http://www.ibm.com/legal/copytrade.shtml>

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

Bluemix™	Global Technology Services®	Rational Team Concert™
CICS®	IBM®	Redbooks®
Cloudant®	IBM MobileFirst™	Redbooks (logo)  ®
Concert™	IBM UrbanCode™	System z®
DataPower®	IBM Watson™	Tealeaf®
DB2®	Jazz™	Tivoli®
DOORS®	Netcool®	WebSphere®
Global Business Services®	Rational®	Worklight®

The following terms are trademarks of other companies:

CloudLayer, SoftLayer, and SoftLayer device are trademarks or registered trademarks of SoftLayer, Inc., an IBM Company.

ITIL is a registered trademark, and a registered community trademark of The Minister for the Cabinet Office, and is registered in the U.S. Patent and Trademark Office.

IT Infrastructure Library is a registered trademark of the Central Computer and Telecommunications Agency which is now part of the Office of Government Commerce.

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

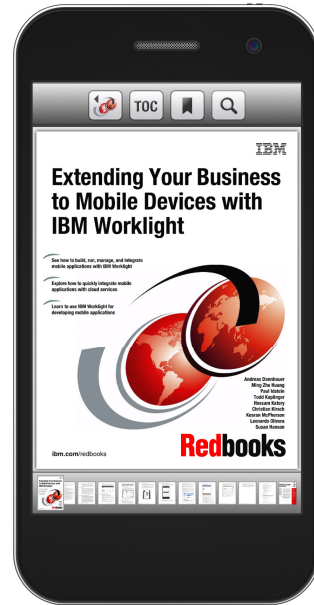
UNIX is a registered trademark of The Open Group in the United States and other countries.

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

Find and read thousands of IBM Redbooks publications

- ▶ Search, bookmark, save and organize favorites
- ▶ Get up-to-the-minute Redbooks news and announcements
- ▶ Link to the latest Redbooks blogs and videos

Get the latest version of the Redbooks Mobile App



Promote your business in an IBM Redbooks publication

Place a Sponsorship Promotion in an IBM® Redbooks® publication, featuring your business or solution with a link to your web site.

Qualified IBM Business Partners may place a full page promotion in the most popular Redbooks publications. Imagine the power of being seen by users who download millions of Redbooks publications each year!



ibm.com/Redbooks
About Redbooks → Business Partner Programs

THIS PAGE INTENTIONALLY LEFT BLANK

Preface

Microservices is an architectural style in which large, complex software applications are composed of one or more smaller services. Each of these microservices focuses on completing one task that represents a small business capability. These microservices can be developed in any programming language. They communicate with each other using language-neutral protocols, such as Representational State Transfer (REST), or messaging applications, such as IBM® MQ Light.

This IBM Redbooks® publication gives a broad understanding of this increasingly popular architectural style, and provides some real-life examples of how you can develop applications using the microservices approach with IBM Bluemix™. The source code for all of these sample scenarios can be found on GitHub (<https://github.com/>).

The book also presents some case studies from IBM products. We explain the architectural decisions made, our experiences, and lessons learned when redesigning these products using the microservices approach.

Information technology (IT) professionals interested in learning about microservices and how to develop or redesign an application in Bluemix using microservices can benefit from this book.

Authors

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



Shahir Daya is an IBM Executive Architect in the IBM Global Business Services® (GBS) Global Cloud Center of Competence. He is an IBM Senior Certified Architect and an Open Group Distinguished Chief/Lead IT Architect. Shahir has over 20 years at IBM, with the last 15 focused on application architecture assignments. He has experience with complex, high-volume transactional web applications and systems integration. Shahir has led teams of practitioners to help IBM and its clients with application architecture for several years. His industry experience includes retail, banking, financial services, public sector, and telecommunications. Shahir is currently focused on cloud application development services, and in particular platform as a service (PaaS), such as IBM Bluemix; containerization technology, such as Docker; design and development of Systems of Engagement (SOE); and cloud-based IBM DevOps, including IBM Bluemix DevOps Services.



Nguyen Van Duy is an Advisory IT Architect with IBM Global Technology Services® (GTS) in Vietnam. He is an IBM Certified IT Architect with solid experience in IBM and open technologies. On his current assignment, Duy works as the Technical Leader for the IBM Global Procurement Services Group in Vietnam to provide enterprise software development services. He is focusing on mobile solutions, including the creation of mobile solutions for IBM employees, and providing his expertise in assisting IBM clients with enterprise mobile engagements. His core experiences are in web, security, distributed computing models, and mobile technologies.



Kameswara Eati is an offering leader and executive architect in the IBM Application Modernization Practice. Kameswara (Kamesh) specifically is the offering leader for the Business Systems Evolution (BSE) area that focuses on application programming interface (API) enablement, microservices, Hybrid Integration technologies, modernization of business processes, and architectural transformations. He has 15 years of experience planning, designing, developing, and modernizing large, complex applications based on Java Platform, Enterprise Edition (Java EE) technologies. He has a solid background in middleware products (such as IBM WebSphere®, Apache Tomcat, and WebLogic) and various database management systems (DBMS), both relational (such as Oracle, IBM DB2®, and Structured Query Language (SQL) Server) and noSQL (such as IBM Cloudant® and MongoDB) technologies. He has deep knowledge in the service-oriented architecture (SOA) and API enablement field, and expertise in modern architecture trends and microservices.



Carlos M Ferreira is a Principle Technical Product Manager for IBM Bluemix. He is responsible for developer experience and roadmap planning. He holds a degree in civil engineering from the University of Rhode Island and a master's in civil engineering from Northeastern University. He is located in the United States and is a licensed professional engineer in Massachusetts. With 21 years of experience in the software development field, his technical expertise includes SOA, and web and mobile development using Java, Python, and Swift. He has written extensively on integrated application development and IT operations (DevOps) and application lifecycle management (ALM) practices.



Dejan Glozic is a Full-Stack Architect for Cloud Data Services Experience, IBM Analytics. He is responsible for the architecture and execution of the end-to-end experience of Cloud Data Services. He holds BSc, MSc, and PhD degrees from the University of Nish, Serbia. He is located in the IBM Toronto Laboratory, Canada. Over the last 21 years with IBM, Dejan has worked on a wide range of projects, including Eclipse, IBM Rational® Jazz™, IBM DevOps Services, and IBM Bluemix, mostly in the area of web development. Using Twitter and blogs, Dejan is a champion for microservices, Node.js, isomorphic apps, web sockets, and other modern technology practices and approaches.



Vasfi Gucer is an IBM Redbooks Project Leader with IBM ITSO. He has more than 18 years of experience in the areas of systems management, networking hardware, and software. He writes extensively and teaches IBM classes worldwide about IBM products. His focus has been on cloud computing for the last three years. Vasfi is also an IBM Certified Senior IT Specialist, Project Management Professional (PMP), IT Infrastructure Library (ITIL) V2 Manager, and ITIL V3 Expert.



Manav Gupta is the North America Technical Leader for IBM Cloud Software. Before this appointment, Manav worked as a Software Client Architect in Canada, bringing over 17 years of telecommunications industry experience and the value of all of the IBM Software Group (SWG) brands to clients in the telecommunications and retail industries. Before this role, Manav worked as a Service Management architect in IBM UK, having come to IBM with the acquisition of Micromuse. Before coming to IBM, Manav worked in a wide variety of software development roles, leading development of multitier unified messaging, customer relationship management (CRM), and inventory management systems. Manav has been leading strategy and architecture development in the IBM Cloud Computing Reference Architecture (CCRA), specifically the Cloud Enabled Data Center and Big Data & Analytics in Cloud workstreams. Manav co-authored several IBM Cloud certifications and enablement collateral, and is leading IBM Point of View (POV) for Big Data & Analytics in Cloud for the Cloud Standards Customer Council. In his Service Component Architecture (SCA) role, Manav led several IBM First-of-a-Kind (FOAK) projects, such as the largest private cloud deployment in Canada. He also led a big data cyber threat intelligence solution, which is now available as a solution offering. Manav has written extensively on fault and performance management using several products and technologies, such as IBM Tivoli® Netcool®, cloud computing, and big data, and has several patents filed in these areas.



Sunil Joshi is an IBM Executive Architect and is the Application Development & Innovation DevOps offering owner for IBM Global Business Services division. He is an IBM Senior Certified and Open group certified Distinguished Architect. His areas of specialization are hybrid cloud solutions, open platform as a service (PaaS), and DevOps strategy. He has created several complex hybrid solutions on cloud and DevOps for large enterprises. Sunil has contributed to several IBM Redbooks and blogs, and is also a co-author of the CCRA.



Valerie Lampkin is a Technical Resolution Specialist for IBM MQ Light and Internet of Things (IoT) Bluemix services based in Atlanta, Georgia, US. She has over 16 years of experience with IBM supporting IBM middleware. Previously, she was part of IBM support for IBM MQ, Managed File Transfer (MFT), and MessageSight. She has a bachelor's degree from Florida State University and is a regular contributor to DeveloperWorks blogs for Middleware and Bluemix. Valerie has previously co-authored three IBM Redbooks Publications on the topics of IBM MQ, IBM MessageSight, and IBM MQ Telemetry Transport (MQTT).



Marcelo Martins is an IBM and Open Group Master Certified IT Architect, a member of the IBM IT Architecture certification board, and a member of the Canadian Academy of Technology Affiliate. His focus areas are enterprise and application architecture. He has worked with major customers in Canada and worldwide in the design and delivery of complex systems. More recently, Marcelo has acted as a member of the Microservices workgroup with the OpenGroup, contributing to the definition and creation of collateral on microservices for the broader technical community.



Shishir Narain is an Open Group certified Master IT Specialist with deep skills in IBM middleware products. He works in IBM Software Services for WebSphere at the India Software Lab, Bangalore. He has 16 years of experience in developing solutions for multiple clients. He has led several end-to-end IT implementations based on SOA. He holds a Master of Technology degree from the Indian Institute of Technology, Kanpur.



Ramratan Vennam is a full stack Software Engineer with deep experience in the landscape of web application technology, with a primary focus on Java EE. At IBM, he has worked in several development roles under the WebSphere product family, and is working on polyglot application run time enablement for Bluemix, the IBM Next Generation Cloud Platform. He is passionate about enabling other developers to use the latest cloud technologies through conferences, hackathons, education sessions, and demos. Ram is an alumni of the Computer Science program at NC State.

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

Erica Wazewski
International Technical Support Organization, Poughkeepsie Center

Alan Chatt, David Currie, Ed Moffatt, Rob Nicholson
IBM UK

Dr. Ali Arsanjani, Jonathan Bond, Peter Formica, Iwao Hatanaka, Russell Kliegel, Ven Kumar, Yang Lei, Russell Levitt, Andrew Lohr, Sugandh Mehta, Dr. Gili Mendel, Rahul Narain, Richard Osowski, Aroopratan D Pandya
IBM US

Now you can become a published author, too

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

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

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

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

- ▶ Use the online **Contact us** review Redbooks form:

ibm.com/redbooks

- ▶ Send your comments in an email:

redbooks@us.ibm.com

- ▶ Mail your comments:

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

Stay connected to IBM Redbooks

- ▶ Find us on Facebook:
<http://www.facebook.com/IBMRedbooks>
- ▶ Follow us on Twitter:
<http://twitter.com/ibmredbooks>
- ▶ Look for us on LinkedIn:
<http://www.linkedin.com/groups?home=&gid=2130806>
- ▶ Explore new Redbooks publications, residencies, and workshops with the IBM Redbooks weekly newsletter:
<https://www.redbooks.ibm.com/Redbooks.nsf/subscribe?OpenForm>
- ▶ Stay current on recent Redbooks publications with RSS Feeds:
<http://www.redbooks.ibm.com/rss.html>



Part 1

Introducing the microservices architectural style

In this part, we introduce the microservices architectural style, and show how you can develop applications using this style in IBM Bluemix.

Providing a platform as a service (PaaS) environment as one of its run times, along with containers and virtual machines (VMs), Bluemix leverages the Cloud Foundry project as one of its open source technologies to accelerate new application development and integrated application development and IT operations (DevOps) methodologies. Bluemix allows separation of interface, implementation, and deployment. You can use Bluemix to create applications in the microservices style and this section shows you how.



Motivations for microservices

In this chapter, we introduce the microservices approach and motivations for using microservices in application development. We describe the following topics:

- ▶ 1.1, “What are microservices” on page 4
- ▶ 1.2, “Benefits from microservices” on page 6
- ▶ 1.3, “What to avoid with microservices” on page 10
- ▶ 1.4, “How is this different than service-oriented architecture?” on page 12
- ▶ 1.5, “Case studies and most common architectural patterns” on page 14
- ▶ 1.6, “Example scenarios using microservices” on page 17

1.1 What are microservices

This book is about microservices. So what exactly are microservices?

Microservices is an architecture style, in which large complex software applications are composed of one or more services. Microservice can be deployed independently of one another and are loosely coupled. Each of these microservices focuses on completing one task only and does that one task really well. In all cases, that one task represents a small business capability. Figure 1-1 shows a sample application using microservices.

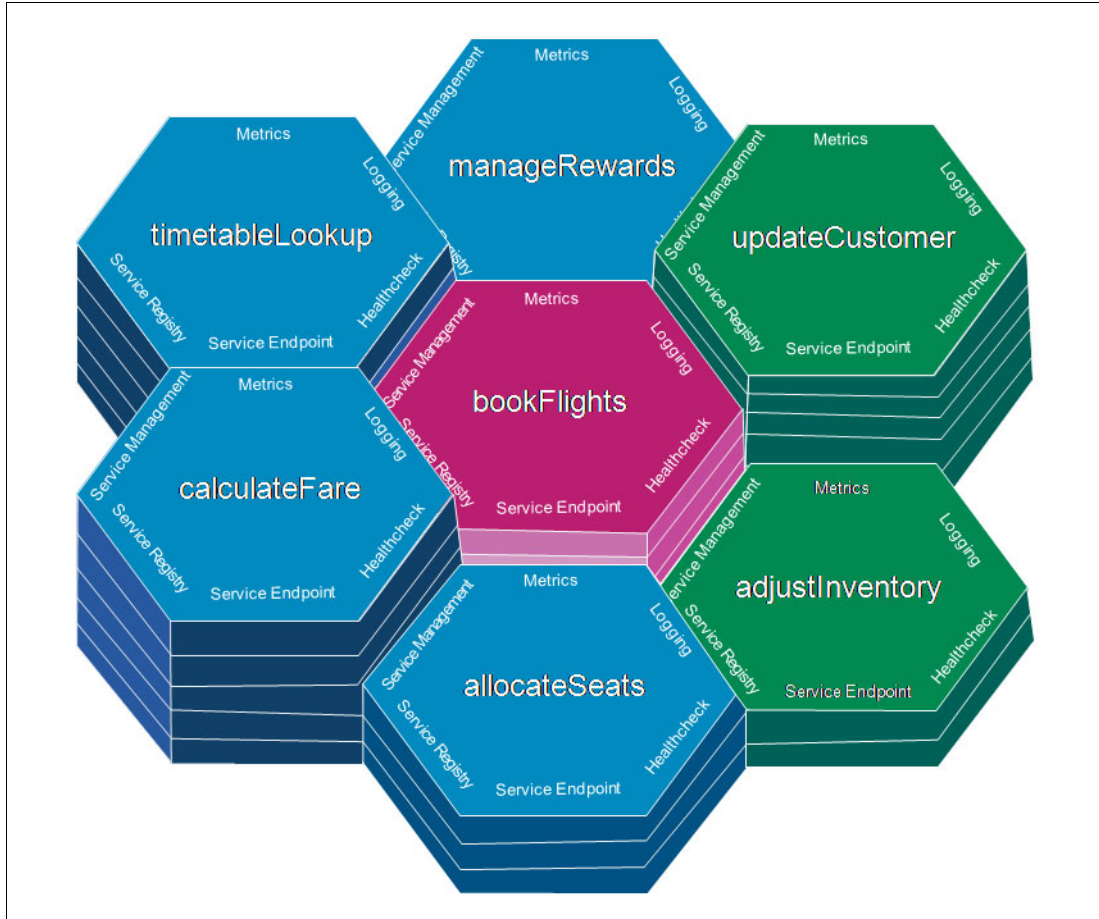


Figure 1-1 A sample application using microservices

Also, microservices can be developed in any programming language. They communicate with each other using language-neutral application programming interfaces (APIs) such as Representational State Transfer (REST). Microservices also have a bounded context. They don't need to know anything about underlying implementation or architecture of other microservices.

The following sections elaborate on the key aspects of this definition.

1.1.1 Small and focused

Microservices need to focus on a unit of work, and as such they are small. There are no rules on how small a microservice must be. A typically referenced guideline is the Two-Pizza Team rule, which states *if you cannot feed the team building a microservice with two pizzas, your microservice is too big*. You want to make the microservice small enough so that you can rewrite and maintain the entire microservice easily within a team if you need to.

Keeping the interface small: The focus should really be on keeping the interface small. Typically that means a small implementation, but that need not necessarily be the case.

A microservice also needs to be treated like an application or a product. It should have its own source code management repository, and its own delivery pipeline for builds and deployment. Although the product owner might advocate the reuse of the microservice, reuse isn't the only business motivation for microservices. There are others, such as localized optimizations to improve user interface (UI) responsiveness and to be able to respond to customer needs more rapidly.

Microservice granularity can also be determined based on business needs. Package tracking, car insurance quote service, and weather forecasting are all examples of services delivered by other third-party service providers, or delivered as a core competency chargeable service.

Latency issue: Making services too granular, or requiring too many dependencies on other microservices, can introduce latency. See 1.3.5, “Don't forget to keep an eye on the potential latency issue” on page 12 for details.

1.1.2 Loosely coupled

Loose coupling is an absolutely essential characteristic of microservices. You need to be able to deploy a single microservice on its own. There must be zero coordination necessary for the deployment with other microservices. This loose coupling enables *frequent and rapid deployments*, therefore getting much-needed features and capabilities to the consumers.

1.1.3 Language-neutral

Using the correct tool for the correct job is important. Microservices need to be built using the programming language and technology that makes the most sense for the task at hand. Microservices are composed together to form a complex application, and they do not need to be written using the same programming language. In some cases Java might be the correct language, and in others it might be Python, for example.

Communication with microservices is through language-neutral APIs, typically an Hypertext Transfer Protocol (HTTP)-based resource API, such as REST. You should *standardize on the integration* and not on the platform used by the microservice. Language-neutral makes it easier to use existing skills or the most optimal language.

1.1.4 Bounded context

What we mean by *bounded context* is that a particular microservice does not “know” anything about underlying implementation of other microservices surrounding it. If for whatever reason a microservice needs to know anything about another microservice (for example, what it does or how it needs to be called), you do not have a bounded context.

1.1.5 Comparing microservices and monolithic architectures

Table 1-1 compares microservices and monolithic architectures.

Table 1-1 Comparing monolithic and microservices architectures

Category	Monolithic architecture	Microservices architecture
Code	A single code base for the entire application.	Multiple code bases. Each microservice has its own code base.
Understandability	Often confusing and hard to maintain.	Much better readability and much easier to maintain.
Deployment	Complex deployments with maintenance windows and scheduled downtimes.	Simple deployment as each microservice can be deployed individually, with minimal if not zero downtime.
Language	Typically entirely developed in one programming language.	Each microservice can be developed in a different programming language.
Scaling	Requires you to scale the entire application even though bottlenecks are localized.	Enables you to scale bottle-necked services without scaling the entire application.

1.2 Benefits from microservices

This section describes some of the pitfalls of monolithic applications, categorizes them into specific challenges to different delivery groups, and describes how microservices provide benefits to each of them.

When not to use microservices: There are also some cases where microservices are *not* a good choice. These cases are described in 1.3, “What to avoid with microservices” on page 10. Knowing about these cases helps limit incurring costs associated with implementing microservices infrastructure, and helps you know when it is warranted.

1.2.1 Enterprise solutions context

To describe some of the benefits of a microservices architecture, we should take a look at the context of most enterprise information technology (IT) architectures and systems of architectures for application development. Over the years, most enterprise solutions have been designed and developed as large, complex, monolithic applications.

There have been some reasonable approaches to decompose applications into layered architectures, such as model view controller (MVC), as shown in Figure 1-2. However, applications are still clumsy with regards to level of business functions that each of the applications are responsible for, therefore making them large and complex to manage.

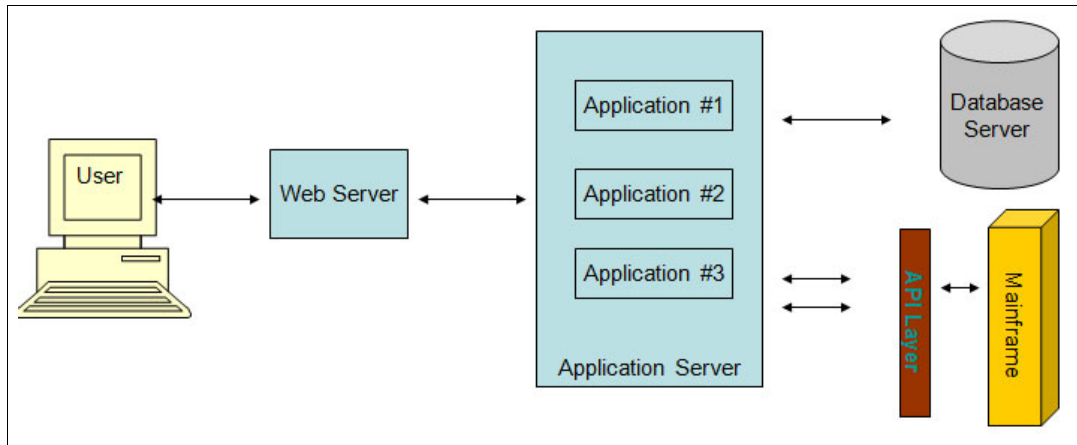


Figure 1-2 Monolithic applications with a multi-tiered architecture

What is a monolithic application? A monolithic application is an application where all of the logic runs in a single app server. Typical monolithic applications are large and built by multiple teams, requiring careful orchestration of deployment for every change. We also consider applications monolithic if, while there are multiple API services providing the business logic, the entire presentation layer is a single large web app. In both cases, microservice architecture can provide an alternative.

1.2.2 Challenges with monolithic architecture

To better understand the challenges with a monolithic architecture, consider them from the perspective of developers, testers, business owners, and service management professionals.

1.2.3 Developer perspective

Large monolithic applications typically have a large application code base, which is often intimidating to developers. When new developers join a large project team, a significant amount of time is required to become familiar with the code base. Developers are hesitant to make application enhancements because of fear of breaking something else due to some unknown dependencies.

It is not always intuitive what part of the application code needs modification for a specific feature request or a change request. This leads to a larger learning curve for onboarding developers, increases project delivery timeline, and reduces rates of enhancement and delivery of new capabilities.

Large code bases can also cause a developer's integrated development environment (IDE) to perform poorly, or in some cases just crash. In cases where services are being developed in the cloud, it also means longer deployment push times. This can increase the roundtrip of development cycle times for developers to get feedback on their code changes.

Monolithic applications can also require longer peer code reviews. The lifespan for monolithic applications tends to be longer than for a microservice, so there likely will be more developers joining and leaving, resulting in higher support costs. This situation also causes inconsistent coding practices and styles and inconsistent documentation methods. All of these elements make maintenance and code reviews more time-consuming and difficult.

Microservice applications allow developers to more easily break down their work into smaller independent teams, and to integrate that work as it is delivered and integrated.

In summary, microservices provide the following benefits for developers:

- ▶ Enables you to avoid large code base, making it easier to maintain or add to features
- ▶ Makes it easier to use existing skills, or the most optimal language
- ▶ Improves deployment times and load times for IDE
- ▶ Makes debugging easier
- ▶ Enables teams to work more independently of each other
- ▶ Simplifies tracking code dependencies
- ▶ Enables complete ownership by a self-contained single team, from definition through development, deployment, operations, and sunseting
- ▶ Makes it easier to scale bottlenecks

1.2.4 Tester perspective

Testing times tend to slow down with large applications, because they take unreasonably longer times to start, leading to lower tester productivity. Every time there is a change to the code that requires a container to reload or restart, testers become less productive.

There is also an issue of *single point of dependence* when multiple test teams are running their test cases. For any incremental change to components, the entire container must be reloaded or restarted, regardless of the size and scope of the changes. Similarly, defects have a greater chance of blocking more team members from testing compared to a microservice.

With monolithic applications, even a small change might have a large ripple effect with regards to regression testing. Typically, monolithic applications require a built up history of a large suite of regression tests. This adds to testing time cycles and adds to the number of testers required to complete testing. In some cases, there is also a risk of false positives showing up from findings unrelated to the specific change being tested.

1.2.5 Business owner perspective

Business owners want their teams to be able to respond to new customer and market needs. Microservices allows for more frequent delivery and faster delivery times. This enables business owners to get quicker feedback, and make adjustments to their investments.

Microservices allow you to have smaller focused teams that align with business revenue and cost centers. That enables business owners to more easily see where their resources are allocated, and to move them from low impact business areas to new or higher impact business areas.

Business owners want to delight their users. Microservices enable a better user experience by enabling you to scale individual microservice to remove slow bottlenecks. Microservices are designed to be resilient, which means improved service availability and an uninterrupted user experience.

Business owners want to be able to offload work to third-party partners without the risk of losing intellectual property. Microservices allow you to segment off work for non-core business functions without disclosing the core services.

By adopting a microservices architectural style, it becomes quickly obvious where there is duplication of services. Having a common platform for developing, building, running, and managing your microservices enables you to more easily eliminate duplicate services, reduce development expense, and lower operational management costs.

1.2.6 Service management perspective

Large monolithic applications often have a large business scope and many IT infrastructure touch points. Any change to the application results in multiple reviews and approvals, resulting in increased deployment cycle times. Minimum centralized governance means that fewer meetings are required for coordination.

By adopting a microservices architectural style with a single platform development enables service management teams to more easily support multiple product and service teams. Efficiencies can be realized by automating deployment, logging, and monitoring practices across multiple microservice project teams. Customers interviewed expect to realize a cost savings of 3 to 1 by automating much of the infrastructure and platform as a service (PaaS).

When problems do occur in production, it is easier to identify and isolate the problem. By identifying smaller non-responsive microservice processes, or reviewing log files marked with a microservice team identifier, operations can quickly identify who from development should be brought in to troubleshoot the problem, all in a self-contained team.

Applications become more resilient as you adopt a microservices architectural style, and organize into DevOps cross-functional, self-contained teams. Teams have greater awareness, responsibility, and control versus siloed functional teams. When problems do occur, the team that owns the microservice has a greater incentive to implement measures to prevent similar outages or incidents in the future.

In new age development, there are so many domain-specific languages and storage technologies available, and developers should have the flexibility to pick *best of breed* depending on the type of workload that they are trying to develop. Leveraging a PaaS provides development teams this flexibility, while enabling service management teams a consistent and more efficient way to manage security and infrastructure across run times.

Figure 1-3 shows a microservices architecture with multiple languages and data store technologies.

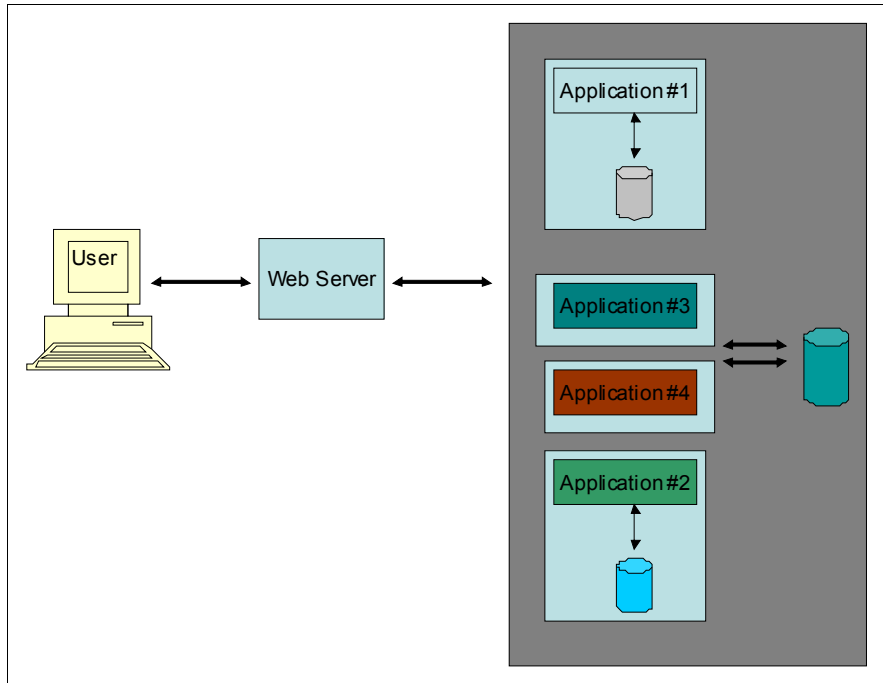


Figure 1-3 Microservices architecture with multiple languages and data store technologies

1.3 What to avoid with microservices

Architectures and approaches normally turn into trends because enough use cases exist to corroborate their genuine usefulness when solving a particular problem or class of problems. In the case of microservices before they were trendy, enough companies built monoliths beyond their manageability. They had a real problem on their hands, which was a large application that fundamentally clashed with the modern ways of scaling, managing, and evolving large systems in the cloud.

Through some trial and error, they reinvented their properties as a loose collection of microservices with independent scalability, lifecycle, and data concerns. The case studies in 1.5, “Case studies and most common architectural patterns” on page 14 are just a small sample of companies successfully running microservices in production.

It is important to remember these use cases, because the trendiness of microservices threatens to compel developers to try them out in contexts where they are not meant to be used, resulting in project failures in some cases. This is bad news for practitioners who derive genuine benefits from such an architecture.

The following section identifies where microservices are not a good choice. It helps limit incurring costs of implementing microservices infrastructure and practices to when it is warranted. It also helps avoid the microservice hype, and prevent some failures that would sour people to an otherwise sound technical approach.

1.3.1 Don't start with microservices

When beginning new applications, do not demand that microservices be included in them. Microservices attempt to solve problems of scale. When you start, your application is tiny. Even if it is not, it is just you or maybe you and a few more developers. You know it intimately, and can rewrite it over a weekend. The application is small enough that you can easily reason about it.

Rock and pebble analogy: There is a reason why we use the word *monolith*: It implies a rock big enough that it can kill you if it falls on you. When you start, your application is more like a pebble. It takes a certain amount of time and effort by a growing number of developers to even approach monolith and therefore microservice territory.

It is important to be aware of when you are approaching monolith status and react before that occurs.

1.3.2 Don't even think about microservices without DevOps

Microservices cause an explosion of moving parts. It is not a good idea to attempt to implement microservices without serious deployment and monitoring automation. You should be able to push a button and get your app deployed. In fact, you should not even do anything.

Committing code should get your app deployed through the commit hooks that trigger the delivery pipelines in at least development. You still need some manual checks and balances for deploying into production. See “Chapter 3, “Microservices and DevOps” on page 39 to learn more about why DevOps is critical to successful microservice deployments.

1.3.3 Don't manage your own infrastructure

Microservices often introduce multiple databases, message brokers, data caches, and similar services that all need to be maintained, clustered, and kept in top shape. It really helps if your first attempt at microservices is free from such concerns. A PaaS, such as IBM Bluemix or Cloud Foundry, enables you to be functional faster and with less headache than with an infrastructure as a service (IaaS), providing that your microservices are PaaS-friendly.

1.3.4 Don't create too many microservices

Each new microservice uses resources. Cumulative resource usage might outstrip the benefits of the architecture if you exceed the number of microservices that your DevOps organization, process, and tooling can handle. It is better to err on the side of larger services, and only split when they end up containing parts with conflicting demands for scaling, lifecycle, or data. Making them too small transfers complexity away from the microservices and into the service integration task. Don't share microservices between systems.

1.3.5 Don't forget to keep an eye on the potential latency issue

Making services too granular or requiring too many dependencies on other microservices can introduce latency. Care should be taken when introducing additional microservices.

When decomposing a system into smaller autonomous microservices, we essentially increase the number of calls made across network boundaries for the services to instrumentally handle a request. These calls can be either service to service calls, or service to persistence component calls. Those additional calls can potentially slow down the operating speed of the system. Therefore, running performance tests to identify the sources of any latency in any of those calls is fundamental.

Measurement is undoubtedly important so that you know where bottlenecks are. For example, you can use IBM Bluemix Monitoring and Analytics service for this purpose. Beyond that, services should be caching aggressively. If necessary, consider adding concurrency, particularly around service aggregation.

1.4 How is this different than service-oriented architecture?

This section examines a comparison of the microservices approach to service-oriented architecture (SOA).

The comparison is complex and somewhat unfair, because proponents of microservice architecture never put forward a claim that it represents a genuinely new approach to building distributed systems. In that light, comparisons between SOA and microservices are normally suggested by the SOA proponents, who want to prove that such a difference does not in fact exist.

Before answering the direct question, consider what SOA stands for:

- ▶ *A set of services and business-aligned functionality* that a business wants to provide to their customers, partners, or other areas of an organization.
- ▶ *An architectural style* that requires a service provider, a service requester with a service description and possibly mediation.

Mediation in SOA: Some more mature forms of SOA might include mediation using an enterprise service bus (ESB). For more information, see the SOA Maturity model:

<https://www.opengroup.org/soa/source-book/osimv2/model.htm>

- ▶ *A set of architectural principles, patterns, and criteria* that address characteristics, such as modularity, encapsulation, loose coupling, separation of concerns, reuse, and composability.
- ▶ *A programming model* complete with standards, tools, and technologies that supports web services, REST services, or other kinds of services.
- ▶ *A middleware solution* optimized for service assembly, orchestration, monitoring, and management.

When defined this way, it is obvious that SOA has a comprehensive and far-reaching set of goals and problems that it is attempting to solve. It also enables us to start noticing the differences between SOA and microservice architecture.

Although in both cases we are indeed talking about a set of services, the ambition of these services is different. SOA attempts to put these services forward to anybody who wants to use them. Microservices, alternatively, are created with a much more focused and limited goal in mind, which is acting as a part of a *single distributed system*.

This distributed system is often created by breaking down a large monolithic application, and the intent is that a collection of microservices continue to work together as a single application. Typically, there is no ambition to serve multiple systems at the same time.

Unlike with SOA, microservices often exist implicitly. They are not discovered at run time, and do not require mediation. They are well known to the consumers, and therefore do not require service description. This does not imply that some kind of discovery is never present.

Some more sophisticated microservice systems might apply degrees of service discovery to make them more flexible and more robust. The point is that *such an architectural pattern is not required*. A microservice system where all microservices are aware of each other using relatively simple configuration files is entirely possible, and can be an effective solution. Teams can discover what microservices are available for them to use at development time using a catalog, registry, or API management tools.

Nevertheless, SOA and microservice architecture share many principles, patterns, and a programming model. After all, microservice architecture *is* a kind of SOA, at least in a rudimentary way.

You can call it an extension or specialization of SOA, in which functional area boundaries are used to define domain models and assign them to teams who can choose their own development languages, frameworks, and deployment details. Similarly, service assembly, orchestration, monitoring, and management is a real concern, and a requirement in microservice systems in addition to more traditional SOA systems.

Unlike SOA, the business and technical motivations for microservices are different. Return on investment (ROI) is driven by accelerated realization of benefits rather than overall business transformation. Companies are less likely to invest in large and lengthy transformational initiatives. Microservices enable incremental transformation using local optimization.

Based on this analysis, are we closer to pinpointing the differences between SOA and microservices architecture? We could say that the key differences are those of *ambition* and *focus*.

When we say *ambition*, we imply the set of problems that a particular approach attempts to solve. By this criterion, microservices are intentionally “trying” to achieve less.

There is a good reason for that. Some of the real-world SOA systems have acquired a reputation of complexity, and the need for very complex tooling and layers of abstraction in order to even approach usability by humans. The most important reason for this complexity was trying to achieve too much, and not engaging in SOA best practices to achieve that recommended partitioning and complexity reduction.

Information: A good reference for SOA best practices is the following article, *Service-oriented modeling and architecture*:

<http://www.ibm.com/developerworks/library/ws-soa-design1/>

You can also go to the following website for more information on SOA:

<http://www-01.ibm.com/software/solutions/soa/>

As a direct consequence of excessive scope, standards, and tooling that SOA had in the past, microservice architecture is intentionally *focused on solving a single problem*. It is mostly focused on incrementally evolving a large, monolithic application into a distributed system of microservices that are easier to manage, evolve, and deploy using continuous integration practices. You do not want any of the microservices that take part in the microservice system to actually be used by another such system.

Although there is much emphasis on reuse, this reuse is not happening at a service level. All microservices in a microservice system are primarily driven by one master application. These microservices, if wanted, can then be reused by other existing or new applications to achieve similar benefits, as described in 1.2, “Benefits from microservices” on page 6.

It is not hard to understand how this differentiation happened. SOA was designed with an ambition to solve very complex enterprise architecture problems, with the goal of facilitating a high level of reusability.

In contrast, microservices architecture was embraced by companies attempting to scale a single web property to *web scale levels*, enable continuous evolution, make engineering teams more efficient, and avoid technology lock-in. It is no surprise that, when approaching the same general area of distributed systems composed of independent services, the end-result is something quite different.

As a direct consequence of more focused and limited scope, and the emphasis on incremental evolution from a traditional system to a microservice system, microservices are increasingly attractive to businesses with significant infrastructure investment. The approach promises less costly and more experimental, *pay as you go* transformation. In contrast, traditional SOA typically required much more serious up-front financial and architectural commitment.

To conclude, both SOA and microservices architecture are kinds of service architectures, because both deal with distributed systems of services communicating over the network. However, practical results are quite different, because the focus of SOA is on reusability and discovery, where the focus of microservices is on replacing a single monolithic application with a system that is easier to manage and incrementally evolve.

1.5 Case studies and most common architectural patterns

Several companies have adopted the microservices architectural patterns to address business challenges particular to their industries. This section describes a few examples of microservices adoption, the most commonly adopted patterns, and the observed benefits. Table 1-2 lists the case studies that we cover in this section, and the commonly adopted architectural patterns used in these case studies.

Table 1-2 Case Studies and most commonly adopted architectural patterns

Case Study	Patterns	Benefits
e-commerce	Decompose a monolithic application using Node.js	<ul style="list-style-type: none"> ▶ Agility to respond to customer needs more rapidly ▶ Faster page load times
Financial services	Incremental re-platforming	<ul style="list-style-type: none"> ▶ Faster delivery ▶ Reduce compute costs
Large brick and mortar retailer	Decompose monolithic application using Node.js	<ul style="list-style-type: none"> ▶ Enables mobile app development ▶ User perceived performance improvements

1.5.1 An e-commerce discount site

This e-commerce company offers discounted electronic coupons that can be redeemed at local or national companies.

The site was originally built as a Ruby on Rails application, but as the company expanded, the single Rails codebase grew as well, making it difficult to maintain and to incorporate new features. This original front-end layer exemplified the monolith pattern.

The company embarked on a year-long project to port its US web traffic from a monolithic Ruby on Rails application to a new Node.js stack.

At the initial phase of the transformation, the front-end layer was redesigned and split into small, independent, and more manageable pieces. Each major section of the website was built as an independent Node.js application.

They built an API layer on top of each of the back-end software platforms, and mobile clients connected to an API endpoint corresponding to the users geographic location.

In a subsequent phase, each major feature of the website was split into a separate web application, with those apps sharing cross-cutting services, such as configuration management and management of test treatments.

The following list includes some of the benefits of this architectural transformation:

- ▶ Faster page loads across the site
- ▶ Faster release of new features
- ▶ Fewer dependencies on other teams
- ▶ Reuse of features in the countries where the e-commerce site is available

1.5.2 Financial services company

This North American financial services company needs to extend one of their core systems to support new external partners, new markets globally, and multiple devices.

The current environment follows a traditional three-tiered architecture, monolithic based on the following components:

- ▶ Presentation layer: Some 3270 screens and web layer (JavaServer Pages (JSP) and servlets) connecting using proprietary messaging platform
- ▶ Business Logic: IBM CICS®, Common Business Oriented Language (COBOL)
- ▶ Data sources: IBM DB2, Virtual Storage Access Method (VSAM), Flat files

Some of the following challenges drove this company to look to modernize their systems:

- ▶ Want to move away from mainframe to distributed environment, primarily due to mainframe costs
- ▶ Difficulty to maintain, modify, and become productive quickly
- ▶ Long cycles of time-to-market to roll out new services
- ▶ Need for better ROI from existing software initiatives, to spend money for current initiatives that fits newer architecture, and a future roadmap that has longer shelf life and better ROI
- ▶ Scalability issues and performance issues
- ▶ 24 x 7 availability and security issues
- ▶ Move from batch-driven to more event-driven approach

This company decided to adopt an incremental modernization approach around microservices to minimize risk. The main strategy employed involves co-existence of existing modules and new services during the multi-year transition phase and phased rollout and decommissioning.

The target goal is to decompose the current application as a set of collaborating and separate, deployable services that can evolve over time independently of each other, and provide strategic scaling of services.

A following set of common services is required to support this architecture:

- ▶ Central logging service
- ▶ Dashboard/Monitoring service
- ▶ Service Discovery and registry service
- ▶ Event-driven component
- ▶ Workflow component

The benefits to be accomplished by employing this architecture:

- ▶ Flexibility and easier extensibility of components
- ▶ Accelerating development cycles and agility
- ▶ Reducing compute costs by moving workloads to more cost-effective run times

1.5.3 Large brick-and-mortar retailer

This traditional brick-and-mortar retailer has a very large established Java services layer running on various platforms. The existing e-commerce systems were not designed to support mobile platforms, or to allow quick turnaround of changes required to adapt to business requirements. The e-commerce platform was monolithic, designed to construct a web page on the server and serve the entire experience to a browser.

The existing e-commerce layer was reaching end of life, and the team decided to replace it with a platform more conducive to agile changes. Some of the following issues were present with the existing e-commerce platform:

- ▶ High response time, causing a diminished user experience
- ▶ Low scalability
- ▶ Low reuse and high maintenance effort and cost
- ▶ High resource consumption

The IT team chose Node.js as the main component of a new orchestration layer, as a proxy for existing APIs.

In the new architecture, Node.js was used as a proxy and orchestration layer, where only minor business logic resides. If this new layer can handle the changes required by the service consumer, the logic would be coded in Node.js. If more involved transformations or calculations were required, this proxy layer would pass the request to the existing API, until a new service in Node.js could be created.

This approach allowed for the following benefits:

- ▶ In-place migration of individual services
- ▶ More control over the user experience due to better ability to detect client disconnects
- ▶ Improved performance due to caching productivity
- ▶ Faster deployments compared to traditional Java builds

1.6 Example scenarios using microservices

This chapter describes different types of applications and advantages of employing a microservices architecture. Next, look at three different applications and the benefits that can be gleaned by implementing microservices techniques:

- ▶ Cloud Trader: An online stock trading system
- ▶ Online Store: An online store application built by using cloud services
- ▶ Acme Air: Fictitious airline application that handles booking flights, including customer data, authentication, and baggage services

Part 2, “Example scenarios using the microservices approach” on page 113, delves deeper into these three scenarios, and explores the techniques to create a microservices architecture. Links to the source code for each of the applications is provided so that you can deploy and run them yourself on Bluemix.

1.6.1 Cloud Trader

This scenario demonstrates how you can functionally decompose an existing monolithic Java application to use individual microservices. The quote service will be separated from the monolithic application and implemented in Node.js. It also demonstrates how to use DevOps capabilities.

The Cloud Trader enables users to log in, view their portfolio, look up stock quotes, and buy or sell stock shares. It is a monolithic application that is being redesigned to benefit from being transformed into smaller architectural components. In its original design, it was big and difficult to add new capabilities to. It also did not provide flexibility to use different trade service providers.

See Chapter 6, “Scenario 1: Transforming a monolithic application to use microservices (CloudTrader)” on page 115 for details about how we restructured this monolithic application to benefit from the microservices approach.

1.6.2 Online Store

This scenario demonstrates how you can design a new *born on the cloud*, microservices designed e-commerce application. It includes three separate microservices that are implemented in three different programming languages.

The Online Store application uses a Bluemix cloud offering. The developers have the freedom to choose the programming language and database best suited for their needs. They are able to break the application into smaller architectural components. This enables each microservice to be built and delivered quickly, and provides for continuous delivery. The individual services are subsequently stronger, both independently and as a whole.

Each of the individual components of the Online Store, *Catalog*, *Order*, and *UI*, are interacting components that are loosely coupled. The overall design has decreased the risk of the entire application being down due to a single point of failure. This does not mean a failure will not occur, but rather that the application has been built to tolerate failure.

When the application is scaled in the cloud with multiple instances for each of the individual components, if a single service instance fails, another service instance can take over when consumers make requests.

Application developers can work independently on their individual services, and are able to quickly deploy fixes and enhancements, enabling a continuous delivery environment. The elasticity provided by the Bluemix cloud services makes it possible for autoscaling to occur.

This scenario outlines the flexibility that exists when the application is designed using a microservice architecture:

- ▶ If the Online Store is getting a large volume of traffic, perhaps only the catalog application needs to be scaled further.
- ▶ If additional security is required to process customer orders, that portion of the application can be updated to enforce more security checks without restricting the other components of the application.
- ▶ If a mobile front end needs to be developed, the RESTful communication between each component makes it easy to add the new interface.

See Chapter 7, “Scenario 2: Microservices built on Bluemix” on page 123 for more details about this application.

1.6.3 Acme Air

Acme Air is a fictitious airline company that required an application to be built with a few key business requirements:

- ▶ The ability to scale to billions of web API calls per day
- ▶ The need to develop and deploy the application in public clouds (as opposed to dedicated, pre-allocated infrastructure)
- ▶ The need to support multiple channels for user interactions (with mobile enablement first and browser/Web 2.0 second)

This scenario demonstrates how you can functionally decompose a monolithic application to use individual microservices. It also shows how to do service monitoring with Hystrix Dashboard (Netflix OSS). Hystrix is a latency and fault tolerance library designed to isolate points of access to remote systems, services, and third-party libraries; stop cascading failure; and enable resilience in complex distributed systems.

The Acme Air application encompasses the following activities that pertain to an airline business:

- ▶ Flights
- ▶ Customer account information
- ▶ Authentication
- ▶ Baggage services

When the monolithic version is converted to a microservices version of the Acme Air application, a web front-end will run with an authentication microservice on a separate run time. Communication between the microservices occurs using RESTful calls. Using the Hystrix Dashboard, the separate services can be monitored.

Details of this application can be found in Chapter 8, “Scenario 3: Modifying the Acme Air application and adding fault tolerance capabilities” on page 131.



Elements of a microservices architecture

In this chapter we introduce the key characteristics of a microservices architecture that make it so appealing for modern application (app) and product development. We also go into a detailed description of how to design microservices and microservices integration. Finally, we give some thought to what the future of microservices architecture will be.

This chapter has the following sections:

- ▶ 2.1, “Characteristics of microservices architecture” on page 20
- ▶ 2.2, “Designing microservices” on page 29
- ▶ 2.3, “REST API and messaging” on page 33
- ▶ 2.4, “The future of microservices” on page 38

2.1 Characteristics of microservices architecture

In this section, we describe the characteristics of microservices architecture.

2.1.1 Business-oriented

As a target system gets broken down into constituent services, there is a real risk that the decomposition might be done along existing boundaries within the organization. This means that there is potential for the system to be more fragile, because there are more independent parts to manage, and parts that do not integrate well even though the resulting services can communicate with each other.

Further, if the services that make up the system are not designed toward the correct goal, they cannot be reused. Development and maintenance costs can also increase if the services are designed along organizational or technology boundary lines.

It is critical to design microservices with the ultimate business objective in mind. This might require teams across organizational boundaries to come together and design the services jointly, rather than using microservices to route calls between teams.

Consider the scenario of “old” style service design shown in Figure 2-1.

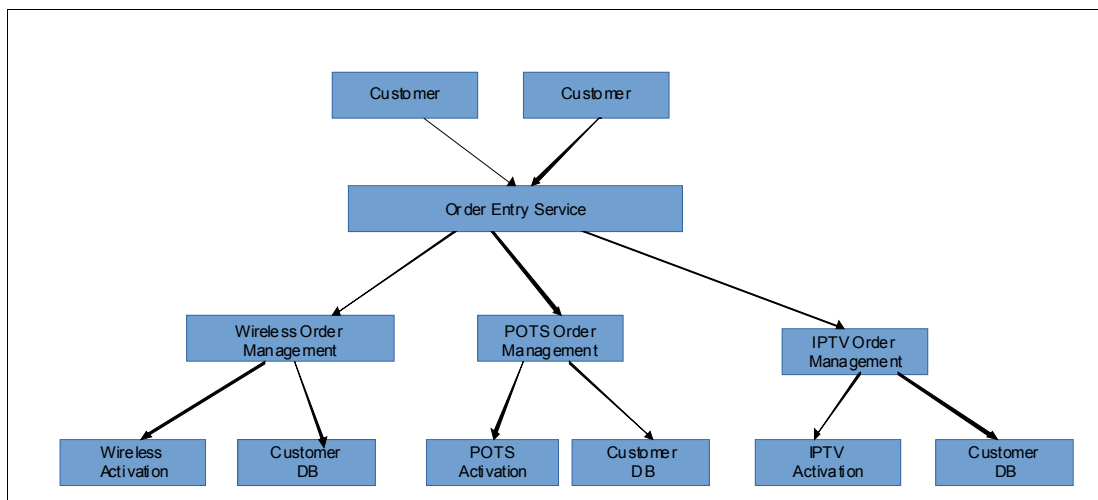


Figure 2-1 “Old” style of service design

In the previous scenario, the organization’s communication structure has been mimicked within the system design, and there are obvious flaws that have been introduced in perfectly isolated services. The ordering system cannot scale, because each organization has its own activation and order management system, each organization has its own view of the customer, and no two parts of the organization have the same view of the customer.

The services designed in the previous scenario are also going to be affected by changes that typically affect an organization, such as churn, mergers & acquisitions, and changes to the network devices used to deliver the services. It wouldn’t be unreasonable to expect that such services are being measured by each department, with its own metrics of bugs, failures, and performance. However, the order entry system as a whole is unable to serve the organization’s purpose.

The microservice approach works well if designers and implementers of each part of the organization communicate with each other to design the services jointly to service a common business purpose. Figure 2-2 shows the application redesigned with the microservices approach. Note that a similar structure would result from using an SOA approach as well.

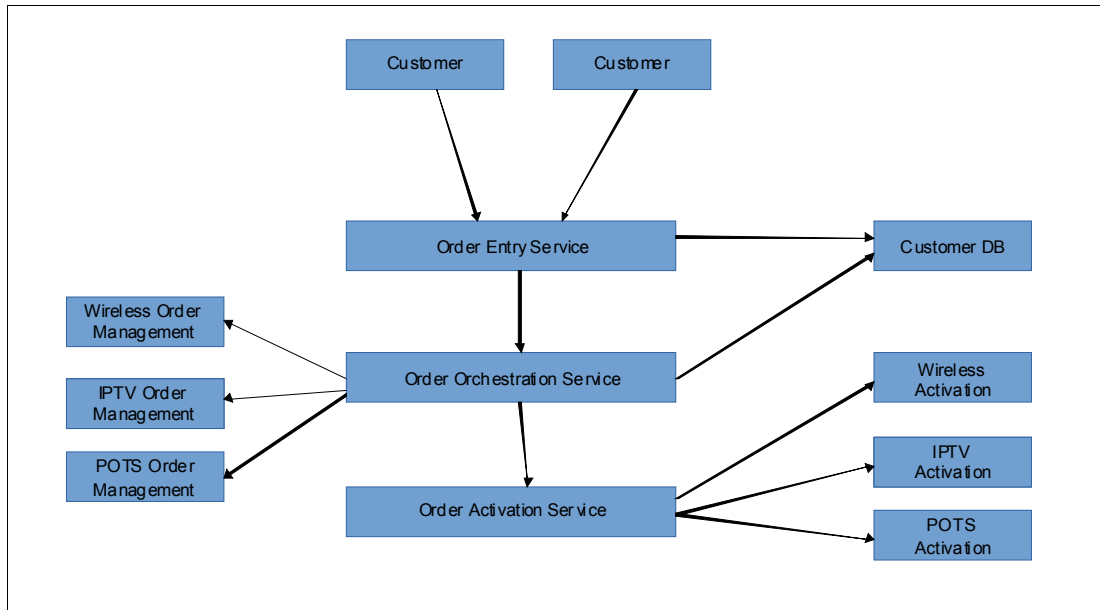


Figure 2-2 Application redesigned with the microservices approach

In the previous scenario, cross-functional teams have come together to design a common order entry service. When an order is received, it is stored in the database, providing a single unified view of the customer. Upon order receipt, the order is sent to the orchestration service, which calls each individual ordering system, as required. These calls then activate each part of the order.

More important than the actual system architecture is the fact that services design does not mimic organization, technological, or communication boundaries. The services can then withstand changes to the organization (such as new products or services being added, or new acquisitions), and changes to staffing. In addition, services are isolated in support of a business function. This is an example of an embodiment of Conway's Law.

2.1.2 Design for failure

Software engineering can borrow many insights from civil engineering, where engineers wrestle deals with the design, construction, and maintenance of the physical environment, such as roads, bridges, canals, dams, and buildings. Civil engineers design systems expecting failure of individual components, and build several layers of redundancies to ensure safe and stable buildings.

The same mind-set can be applied to software engineering. The mind-set shift dictates acceptance of the fact that isolated failures are inevitable, but the design goal is to keep the system functioning for as long as possible. Engineering practices, such as fault modeling and fault injection, should be included as part of a continual release process to design more reliable systems. There are several design patterns for designing for failure and stability. Some of these patterns are described in the following sections.

Circuit breaker

The circuit breaker pattern is commonly used to ensure that when there is failure that the failed service does not adversely affect the entire system. This would happen if the volume of calls to the failed service was high, and for each call we'd have to wait for a timeout to occur before moving on. Making the call to the failed service and waiting would use resources that would eventually render the overall system unstable.

The circuit breaker pattern behaves just like a circuit breaker in your home electrical system. It trips to protect you. Calls to a microservice are wrapped in a circuit breaker object. When a service fails, the circuit breaker object allows subsequent calls to the service until a particular threshold of failed attempts is reached. At that point, the circuit breaker for that service trips, and any further calls will be short-circuited and will not result in calls to the failed service. This setup saves valuable resources and maintains the overall stability of the system.

Figure 2-3 shows the sequence diagram for the circuit breaker design pattern.

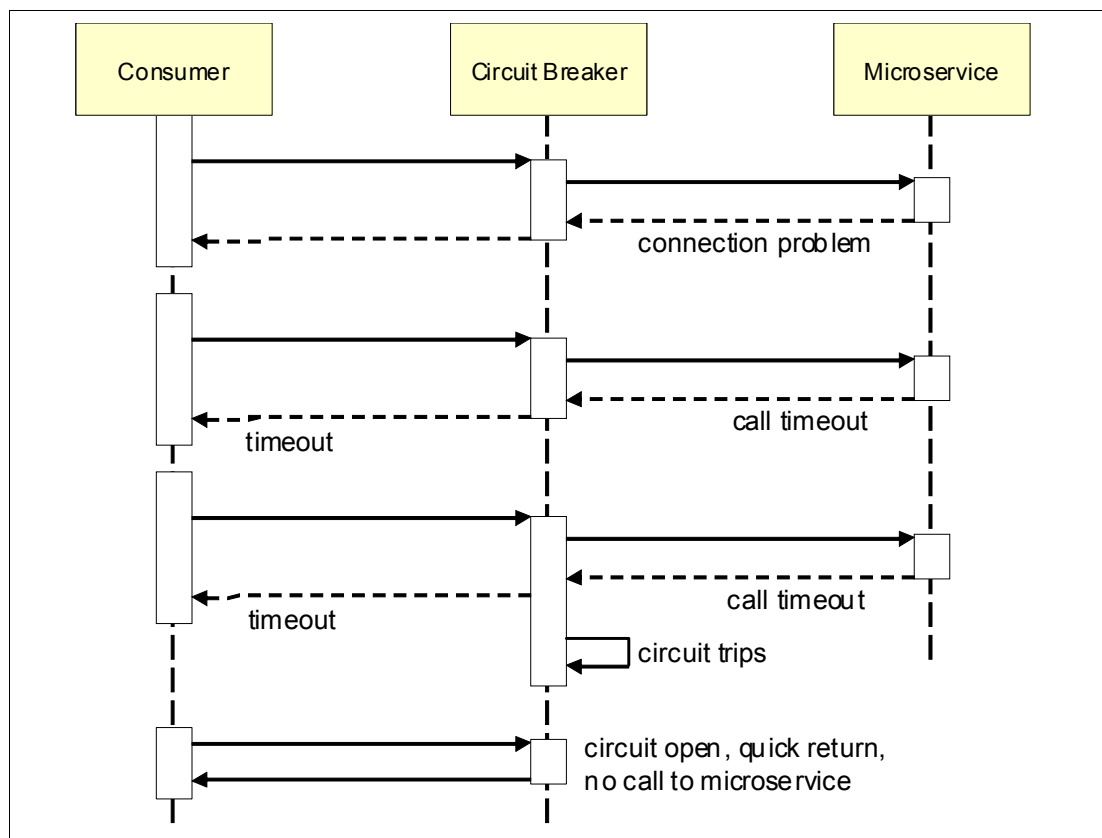


Figure 2-3 Circuit breaker sequence diagram

When the circuit breaker trips and the circuit is open, a fallback logic can be started instead. The fallback logic typically does little or no processing, and returns a value. Fallback logic must have little chance of failing, because it is running as a result of a failure to begin with.

Bulkheads

A ship's hull is composed of several individual watertight bulkheads. The reason for this is that if one of the bulkheads gets damaged, that failure is limited to that bulkhead alone, as opposed to taking down the entire ship.

This kind of a partitioning approach can be used in software as well, to isolate failure to small portions of the system. The service boundary (that is, the microservice itself) serves as a bulkhead to isolate any failures. Breaking out functionality (as we would also do in an SOA architecture) into separate microservices serves to isolate the effect of failure in one microservice.

The bulkhead pattern can also be applied within microservices themselves. As an example, consider a thread pool being used to reach two existing systems. If one of the existing systems started to experience a slow down and caused the thread pool to get exhausted, access to the other existing system would also be affected. Having separate thread pools would ensure that a slowdown in one existing system would exhaust only its own thread pool, and not affect access to the other existing system.

See Figure 2-4 for an illustration of this separate thread pool usage.

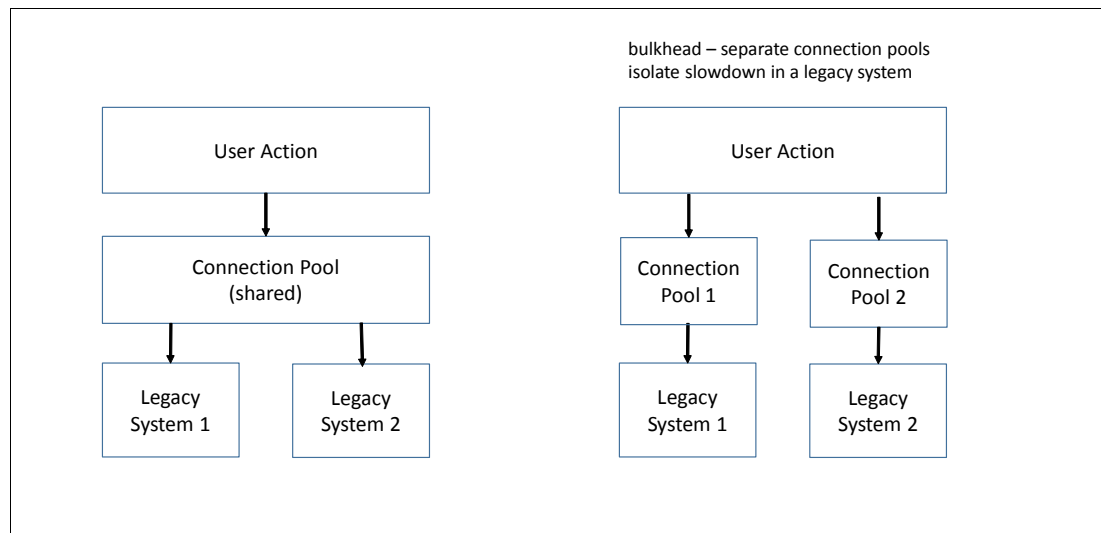


Figure 2-4 Example using the bulkhead pattern: Using separate thread pools to isolate failure

2.1.3 Decentralized data management

In a monolithic application, it is easy to deal with transactions because all of the components are part of the monolith. When you move to a microservices architecture that is distributed, you now must potentially deal with transactions that are spread across multiple services. In a microservices architecture, the preference is for *BASE* (*Basically Available, Soft state, Eventual consistency*) over *ACID* (*Atomicity, Consistency, Isolation, Durability*). We avoid distributed transactions whenever possible.

Ideally, you want every microservice to manage its own database. This enables polyglot persistence, using different databases (for example, Cloudant versus MongoDB, both of which are NoSQL) and different types of data stores (such as Structured Query Language (SQL), NoSQL, graph). However, we might need to have more than one microservice use the same database for one of many reasons, for example, to preserve the ACID nature of a transaction that would otherwise be distributed across microservices and databases.

Whatever the reason, careful thought must be given to sharing the database across microservices. The pros and cons of doing so must be considered. Sharing a database does violate some of the principles of a microservices-based architecture. For example, the context is no longer bounded, where the two services sharing a database need to know about each other, and changes in the shared database need to be coordinated between the two.

In general, the level of sharing between microservices should be limited as much as possible to make the microservices as loosely coupled as possible.

2.1.4 Discoverability

As we described earlier, microservices architecture requires making services reliable and fault tolerant. This in turn implies construction of microservices in such a way that, as the underlying infrastructure is created and destroyed, the services can be reconfigured with the location of the other services that they need to connect to.

With the use of cloud computing and containers for deployment of microservices, these services need to be reconfigured dynamically. When the new service instance is created, the rest of the network can quickly find it and start to communicate with it.

Most service discovery models rely on a registry service that all services explicitly register with, and make calls to it later to communicate to the other services that they rely upon. There are several open source registry services currently available that provide service discovery capabilities, such as Zookeeper, Consul, and Eureka.

Remember that not all service discovery models rely on a registry service. For example, Cloud Foundry, the open source platform as a service (PaaS) that Bluemix is built on, does not rely on a registry service. Services in Cloud Foundry are advertised to the Cloud Controller to make it aware of the existence of the service and its availability for provisioning.

Good discovery services are a rich source of metadata, which the registering services can provide, and which can then be used by requesting services to make informed decisions about how to use it. For example, requesting (or *client*) services can ask about whether a service has any specific dependencies, their implications, and whether it is able to fulfil any specific requirements. Some discovery services can also have load balancing capabilities.

2.1.5 Inter-service communication design

When microservices are spread across the deployment container of choice (servers, virtual machines (VMs), or containers), how do they communicate with each other? For one-way communications, message queues might be enough, but that wouldn't work for synchronous 2-way communications.

In monolithic applications, clients of the application, such as browsers and native applications, make Hypertext Transfer Protocol (HTTP) requests to a load balancer, which spreads the requests to several identical instances of the application. But in microservices architecture, the monolith has been replaced by a collection of services.

One scenario could be that the clients of the system can make RESTful application programming interface (API) calls, as depicted in Figure 2-5.

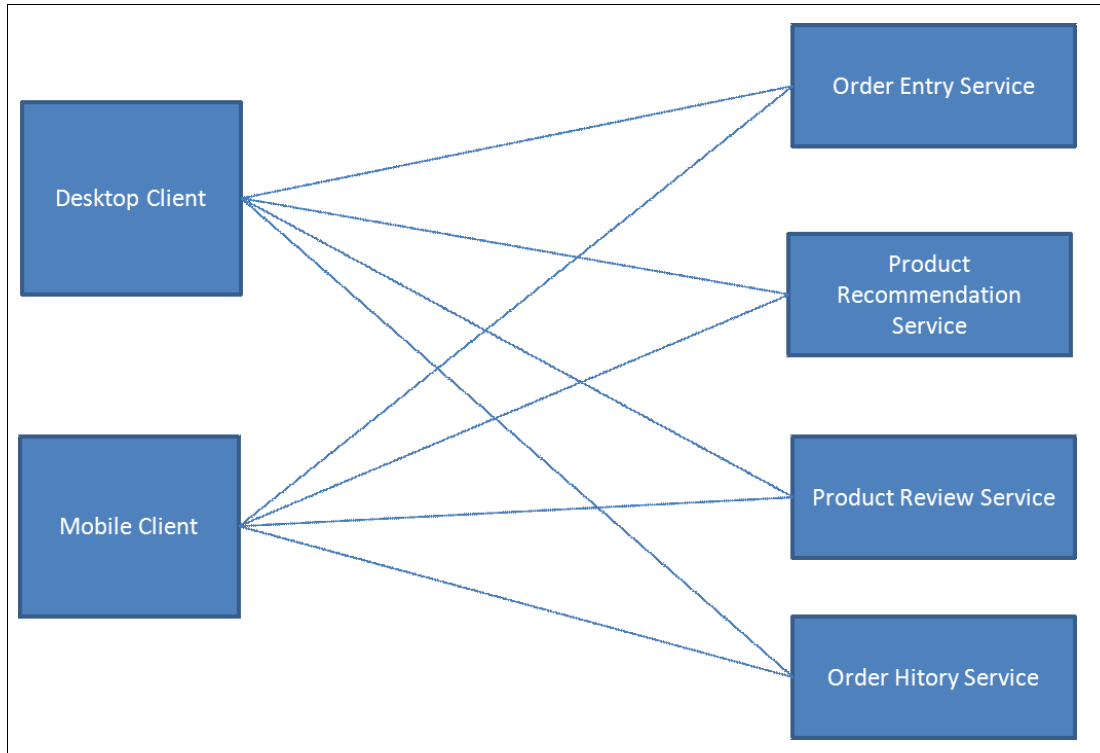


Figure 2-5 RESTful API calls between client and services

On the surface, this might seem desirable. However, there is likely a mismatch service granularity. Some clients can be “chatty”; others can require many calls to get the necessary data. Because the services would probably scale differently, there are other challenges to be addressed such as handling of partial failures.

A better approach is for the clients to make a few requests, perhaps only one, using a front-end such as an API Gateway, as shown in Figure 2-6.

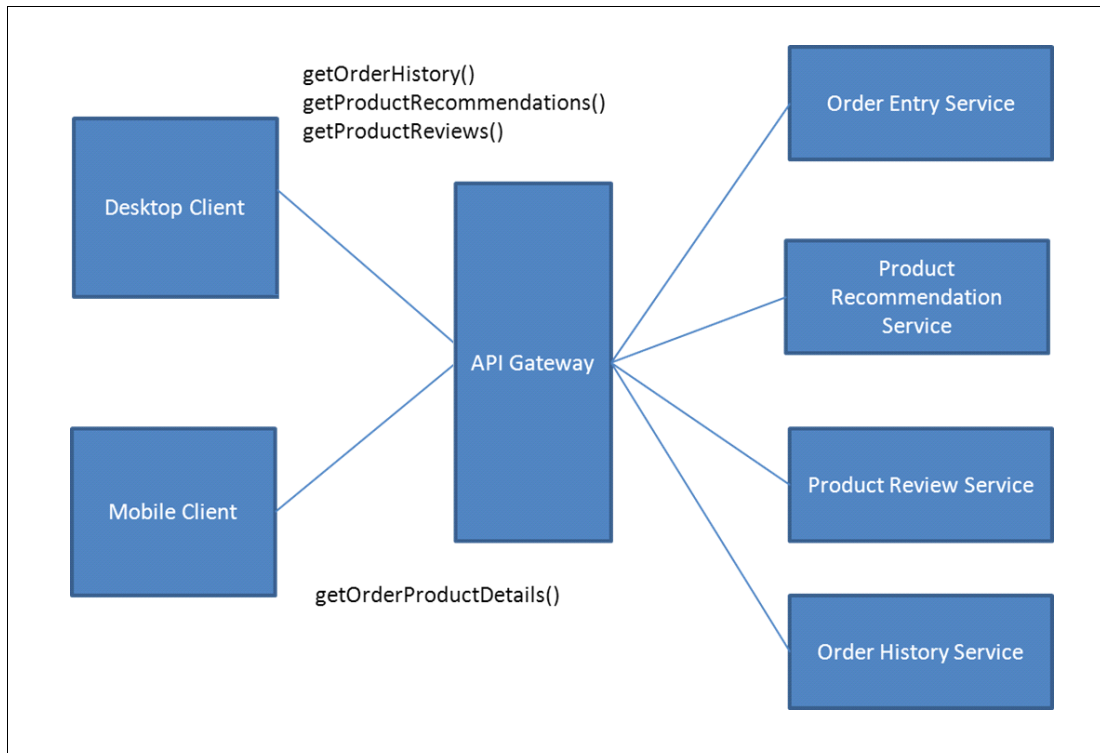


Figure 2-6 Fronting your microservices with an API Gateway

The API Gateway sits between the clients and microservices, and provides APIs that are tailored for the clients. The API Gateway is principally about hiding technological complexity (for example, connectivity to a mainframe) versus interface complexity. Informally, an API Gateway is a facade. However, a facade provides a uniform view of complex internals to external clients, where an API Gateway provides a uniform view of external resources to the internals of an application.

Much like the facade design pattern, the API Gateway provides a simplified interface to the clients, making the services easier to use, understand, and test. This is because it can provide different levels of granularity to desktop and browser clients. The API Gateway might provide coarse-grained APIs to mobile clients, and fine-grained APIs to desktop clients that might use a high performance network.

In this scenario, the API Gateway reduces chattiness by enabling the clients to collapse multiple requests into a single request optimized for a given client (such as the mobile client). The benefit is that the device then experiences the consequences of network latency once, and leverages the low latency connectivity and more powerful hardware server-side.

The following sections describe other considerations for implementing communications between services.

Synchronous HTTP versus asynchronous messaging

There are two main approaches to interprocess communication:

- ▶ Synchronous HTTP-based mechanisms, such as Representational State Transfer (REST), SOAP, or WebSockets, which enable keeping a communication channel between browser and server open for event-driven request/response
- ▶ Asynchronous messaging using a message broker

Synchronous messaging is a simple and familiar mechanism, which is firewall-friendly. The disadvantage is that it does not support other patterns, such as a publish/subscribe model. The publish/subscribe messaging style is more flexible than point-to-point queuing, because it enables multiple receivers to subscribe to the flow of messages. This enables more microservices to be easily added on to the application.

It also does not allow queuing of requests, which can act as a kind of shock absorber. Asynchronous messaging decouples the services from each other in time, enabling the sending thread to get on with work while the messaging system takes responsibility for the message, keeping it safely until it can be delivered to the destination.

With synchronous applications, both the client and server must be simultaneously available, and the clients always need to know the host and port of the server, which is not always straightforward when services are auto-scaling in a cloud deployment. In such scenarios, one of the service discovery mechanisms described previously is required.

Alternatively, an asynchronous mechanism uses a message broker, which decouples the message consumers and producers. The message broker buffers messages until the consumer is able to process them.

A further benefit of messaging is that the back-end applications behind the queue can distribute workload among several clients. This type of *worker offload* scenario enables developers to build scalable, responsive applications. Implementing one or more instances of a back-end worker application enables the putting application to continue its work without the need to wait for the back-end application to complete.

There are pros and cons to both approaches. Most applications use a combination of the two approaches to suit their needs.

2.1.6 Dealing with complexity

A microservices architecture creates additional resource use for many operations. There is clearly a huge increase in moving parts, and as such, complexity. Rationalizing the behavior of each individual component might be simpler, but understanding the behavior of the whole system is much more difficult.

Tuning a microservices-based system, which can consist of dozens of processes, load balancers, and messaging layers, requires a high-quality monitoring and operations infrastructure. Promoting the plethora of microservices through the development pipeline into production requires a high degree of release and deployment automation.

The following list describes some considerations for dealing with complexity introduced by the use of microservices:

Testing Testing a complex system requires suites of unit and integration tests, and a testing framework that simulates failed or failing components, service degradation, stress testing, and the overall system's behavior over the course of a typical day. Such test frameworks ensure system correctness and its overall resilience, and in turn can be used as a monitoring tool.

Because of the focus on ensuring that the system behaves well when a component fails, it is not uncommon to test failure in the production environment in a microservices architecture. Automated tests that force a component to fail enable testing the ability of the system to recover, and testing the monitoring capabilities.

Monitoring In addition to conventional monitoring systems, well-designed services can provide additional health check information using publish/subscribe over a messaging queue. In this way, information can be collected by the monitoring service, providing rich health and configuration indicators.

These indicators in turn can be used to ensure that the services have enough capacity to serve current request volume, and compare current performance against expectations. Custom health checks can be registered to gather additional data. All collection information can then be aggregated, and displayed in the monitoring dashboard.

DevOps The operations challenge of keeping microservices up and running means that you need high-quality DevOps skills embedded within your development team. The team needs to be operationally focused and production aware.

Polyglot programming Idiomatic use of microservices also means that the resulting system would be polyglot, which presents a unique challenge of maintaining a hyper-polyglot system. As a result, the database administrator (DBA) now needs to be replaced with programmers who can deploy, run, and optimize noSQL products.

Maintaining a pipeline of skilled developers, tinkerers, and researchers is as critical as maintaining the DevOps pipeline to ensure that there are enough skilled developers to maintain the platform, and to improve system design as new advancements are made. Note that most organizations have a short list of programming languages that they use. There are benefits to that from a code and skills reuse standpoint.

2.1.7 Evolutionary design

Microservices architecture enables an evolutionary design approach. The core idea is that you can introduce new features and capabilities in your application by changing a single microservice. You only need to deploy and release the microservices that you changed. The change only affects the consumers of that microservice and, if the interface has not changed, all consumers continue to function.

Because microservices are loosely coupled, small and focused, and have a bounded context, changes to them can be made rapidly and frequently with minimal risk. This ease of upgradability of a microservice enables evolution.

Because a microservice is small and usually designed, developed, and maintained by a small “two-pizza” team, it is typical to find microservices that get entirely rewritten, as opposed to being upgraded or maintained. A common trade-off is in how small you make your microservice. The more microservices you have in the decomposition of a monolithic application, the smaller each is. And the smaller each is, the lower the risk in deploying/releasing changes to it.

However, the smaller each microservice is, the more likely it is that a typical application update will require more microservices to be deployed, therefore increasing the risk. The principles of microservices, such as independent deployability and loose coupling, need to be considered when determining what capability becomes a microservice. There is a more detailed description of this in the next section.

2.2 Designing microservices

One of the great advantages of microservice architecture is the freedom to make decisions about technology stack and microservice size on a per-service basis. As usual in life, with great freedom comes great responsibility. Although it is true that many options are available to the developers, they are not equal and must be considered carefully. It starts by having a clear understanding of the purpose of the microservice in fulfilling the user’s experience.

2.2.1 Use design thinking to scope and identify microservices

Design thinking is a process for envisioning the whole user experience. Rather than focusing on a feature, you instead focus on the user experience (what the person is thinking, doing, and feeling as they have the experience). Design thinking is helpful for scoping work to usable and releasable units of function. Designs make it easier to functionally decompose and identify microservice.

Design thinking includes the following concepts:

- ▶ Hills
- ▶ Hills Playback
- ▶ Scenario
- ▶ User story
- ▶ Epics
- ▶ Sponsor users
- ▶ Identifying microservices opportunities

Hills

Hills provide business goal for your release timeline. A Hill defines the who, what, and how of what you want to accomplish. A team typically identifies three Hills per project, and a technical foundation. Hills provide the “commander’s intent” to allow teams to use their own discretion on how to interpret and implement a solution that provides for a good user experience. Hills definition should be targeted at a user and measurable. Avoid using vague or non-quantifiable adjectives. Example 2-1 shows a sample Hill.

Example 2-1 Sample Hill

Allow a developer to learn about iOS Bluemix Solution and deploy an application within 10 minutes.

Hills Playback

Hills playback provides a summary of what the team will target. Hills Playback sets the scope for a specific release time period. Playback 0 is when the team has completed sizings, and commits to the business sponsors regarding the outcomes that it wants to achieve. Playbacks are performed weekly with participation from the cross-functional teams, and the sponsor users who try to perform the Hills. Figure 2-7 shows a Hills Playback timeline.

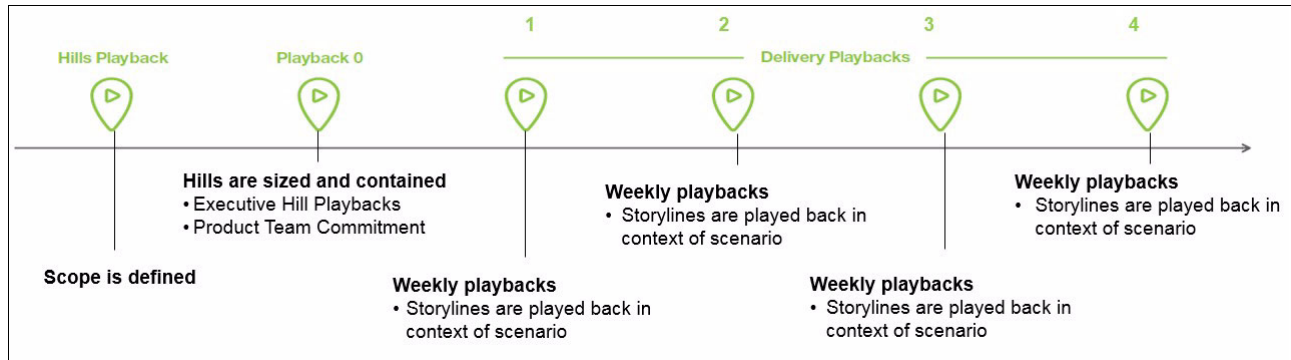


Figure 2-7 Hills Playback timeline

Scenario

A scenario is a single workflow through an experience, and it sets the story and context used in Hills Playbacks. Large scenarios can be further decomposed into *scenes* and *product user stories*. Scenarios capture the “as is” scenario and the “to be” improved scenario.

User story

A user story is a self-contained, codeable requirement that can be developed in one or two days, expressed in terms of user experience, for example: “As a developer, I want to find samples and quickly deploy them on Bluemix so I can try them.”

Epics

Epics group stories into a form that can be reused multiple times across the scenario, so that stories aren’t repeated.

Sponsor users

Sponsor users are users that are engaged throughout the project to represent target personas for a project. They are expected to lead or participate in Playbacks. Sponsor users can be clients who use the applications that include microservices. They can also be internal developers who use microservice onboarding tools that you are developing in support of your DevOps release process.

Identifying microservices opportunities

For each design, identify opportunities for potential reuse of a service across other designs. The Hill definition described earlier was targeted at iOS solutions with Bluemix. We discovered that we needed the ability to deploy samples on the other Solutions pages for Bluemix and other websites. We targeted *Deploy to Bluemix* as a separate microservice.

The Deploy to Bluemix microservice team owns implementing the service and performing functional and system integration testing of the stories that the microservice is used in. The service includes logging data to collect usage data. This enables us to better quantify the business effect that the service is providing. It provides visibility into the most popular applications being deployed, and user acquisition after users deploy a sample.

2.2.2 Choosing the implementation stack

Because microservice systems consist of individual services running as separate processes, it stands to reason to expect any competent technology capable of supporting communication protocols, such as HTTP REST, or messaging protocols, such as MQ Telemetry Transport (MQTT) or Advanced Message Queuing Protocol (AMQP), to work. Nevertheless, several considerations need to be considering when choosing the implementation stack:

- ▶ Synchronous versus asynchronous. Classic stacks, such as Java Platform, Enterprise Edition (Java EE), work by synchronous blocking on network requests. As a result, they must run in separate threads to be able to handle multiple concurrent requests. Asynchronous stacks handle requests using an event loop that is often single-threaded, yet can process many more requests when handling them requires downstream input/output (I/O) operations.

Tip: Node.js and alternative Java stacks, such as Netty, are better candidates for microservices for this reason.

- ▶ I/O versus processor (CPU) bound. Solutions such as Node.js are great for microservices that predominantly deal with I/O operations. Node.js provides for waiting for I/O requests to complete without holding up entire threads. However, because the execution of the request is performed in the event loop, complex computations adversely affect the ability of the dispatcher to handle other requests. If a microservice is performing long-running operations, it is better to do one of the following two things:
 - Offload long-running operations to a set of workers written in a stack best suited for CPU-intensive work (Java, Go, or even C).
 - Implement the entire service in a stack capable of multi-threading.
- ▶ Memory and CPU requirements. microservices are always expressed in plural, because we run several of them, not one. Each microservice is further scaled by running multiple instances of it. There are many processes to handle, and memory and CPU requirements are an important consideration when assessing the cost of operation of the entire system. Traditional Java EE stacks are less suitable for microservices from this point of view, because they are optimized for running a single application container, not a multitude of containers. Again, stacks such as Node.js and Go are seen as a go-to technology because they are more lightweight and require less memory and CPU power per instance.

In theory, it is possible to create a microservice system in which each service uses a different stack. In most situations, this would be foolish. Economy of scale, code reuse, and developer skills all limit this number at a level that is around 2 - 3.

In most microservice systems, developers use Node.js microservices for serving web pages, due to the great affinity of Node.js with client-side JavaScript running in the browser. They use a good CPU-friendly platform (Java, Go) to run back-end services, and reuse existing system libraries and toolkits. More than two stacks might not be practical for most teams.

Nevertheless, there is always a possibility of trying out a new stack in a new microservice, without dragging the rest of the system through the costly rework. What we are trying to say here is that this situation is a reflection of a new stack pilot phase, not the norm. If the pilot is successful, the rest of the system is ported to again arrive at the 1 - 2 stack norm.

2.2.3 Sizing the microservices

One of the most frustrating and imprecise tasks when designing a microservice system is deciding on the number and size of individual microservices. There is no strict rule regarding the optimal size, but there are some practices that have been proven in real-world systems. Most sizing exercises revolve around partitioning the problem space (in a green-field system) or an existing monolithic application into individual microservices. Several techniques can be used alone or in combination:

- ▶ Number of files. You can gauge the size of a microservice in a system by the number of files it consists of. This is imprecise, but at some point you will want to break up a microservice that is physically too large. Large services are hard to work with, hard to deploy, and take longer to start and stop.

However, care should be taken to not go overboard in the other direction. When microservices are too small (frequently called *nanoservices* as an anti-pattern), the resource cost of deploying and operating such a service overshadows its utility. Although microservices are often compared to the UNIX design ethos (*do one thing and do it correctly*), it is better to start with larger services. You can always split one service into two later.

- ▶ Too many responsibilities. A service that is simultaneously “floor wax and desert topping” might need to be broken up, because it can be hard to reason about, test, maintain, and deploy. Even if all of these responsibilities are of the same type (for example, REST endpoints), you might have too many of them for a single service to handle.
- ▶ Service type. A good rule is that a microservice should do one thing, for example, one of the following tasks:
 - Handle authentication
 - Serve several REST endpoints
 - Serve several web pages

Normally, you don’t want to mix these heterogeneous responsibilities. Although this might seem the same as the *too many responsibilities* rule, it is not. It deals with the quality, not the quantity, of the responsibilities. An anti-pattern would be a service that serves web pages and also provides REST end-points, or serves as a worker.

- ▶ Bounded context separation. This rule is important when an existing system is being partitioned into microservices. The name comes from a design pattern proposed by Martin Fowler (<http://martinfowler.com/bliki/BoundedContext.html>).

It represents parts of the system that are relatively self-sufficient, so that there are few links to sever to turn them into microservices. If a microservice needs to talk to 10 other microservices to complete its task, that might be an indication that the cut was made in an incorrect place in the monolith.

- ▶ Team organization. It is not a secret that many microservice systems are organized around teams responsible for writing the code. It stands to reason that microservice partition goes along team lines to maximize team independence.

After all, one of the key reasons microservices are popular as an architectural and organizational pattern is that they allow teams to plan, develop, and deploy features of a system in the cloud without tight coordination. It is therefore to be expected that microservice number and size are dictated by organizational *and* technical principles.

In the end, a well-designed microservice system uses a combination of these rules. They require a degree of good judgment that is acquired with time and experience with the system. Until that experience is acquired, we suggest starting small, with microservices that might be on the larger size (more like *mini-services*) until more “fault lines” have been observed for subsequent subdivisions.

Note that, in a well-designed system fronted with a reverse proxy, this reorganization can be run without any disruption. Where a single microservice was serving two URL paths, two new microservices can serve one path each. The moral is that microservice system design is an ongoing story. It is not something that must be done all at once, immediately.

2.3 REST API and messaging

In this section, we describe using REST API and messaging in applications developed using the microservices approach.

2.3.1 REST

REST is an architectural style for networked applications, primarily used to build web services that are lightweight, maintainable, and scalable. A service based on REST is called a RESTful service. REST is not dependent on any protocol, but almost every RESTful service uses HTTP as its underlying protocol.

REST is often used for web applications as a way to allow resources to communicate by exchanging information. If you consider the web as an application platform, REST enables you to have applications that are loosely coupled, can be scaled and provide functionality across services.

Over the past decade, REST has emerged as a predominant web service design model, and almost every major development language includes frameworks for building RESTful web services. The REST APIs use HTTP verbs to act on a resource. The API establishes a mapping between **CREATE**, **READ**, **UPDATE**, and **DELETE** operations, and the corresponding **POST**, **GET**, **PUT**, and **DELETE** HTTP actions.

The RESTful interface focuses on the components' roles and resources, and ignores their internal implementation details. Requests are sent to a server that is component-aware, which masks the intricate details from the users. The interactions must be stateless, because the requests might travel between layered intermediaries between the original client agent and the server.

These intermediaries might be proxies or gateways, and sometimes cache the information. A constraint exists with REST that requires stateless communication. Every request should contain all of the information that is required to interpret that request. This increases the visibility, reliability, and scalability, but also decreases performance because of the larger messages required for stateless communication.

Figure 2-8 is a simple overview of REST architecture showing how intermediary proxies and gateways, and the REST server, can cache information.

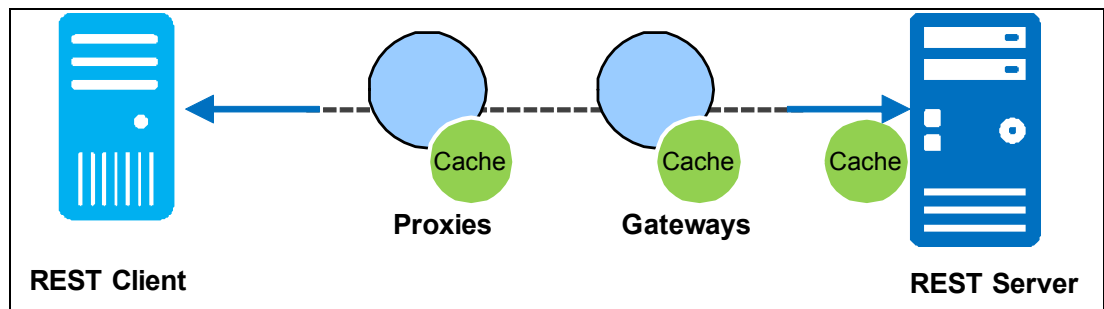


Figure 2-8 REST architecture with intermediary caches

As shown in Figure 2-8 on page 33, one or more proxies or gateways can exist between the client agent that is requesting the information and the server. The responses must have a mechanism to define themselves as cacheable or non-cacheable. Well-managed caching improves the scalability and performance of a RESTful service.

Because REST is designed around a request/response model, if the wanted response is not available, the application would need to do the call again later. In some cases, this could lead to frequent polling by the application.

2.3.2 Messaging

In today's cloud environment it is desirable to implement a microservices architecture for many scenarios. Moving away from monolithic applications to a coalition of smaller collaborating applications requires a way for these small discrete microservices to communicate with each other. This is where messaging can help. By decomposing your application, actually breaking it down to components, it keeps the services modular and scalable so that they are easier to debug.

Messaging is a critical component to the application development world, enabling developers to break down an application into its components, and loosely couple them together. Messaging supports asynchronous coupling between the components, with messages sent using message queues or topics.

A messaging publish/subscribe pattern reduces on unnecessary linkages and network traffic between applications. Rather than having to use the frequent REST polling calls, the application can sit idly until the message is published to all of the interested (subscribed) parties. At that point, the subscribed application receives the message. In essence, rather than using a REST "are we there yet" call, the subscriber gets a "shoulder tap" to indicate that the message they want is available.

In a publish/subscribe topology, the publishing application sends a message to a destination, often a topic string. The ability of the publisher to put the message is not dependent on subscriber applications being accessible. The message broker acts as an intermediary, and is responsible for delivering the message to every subscriber.

At any point in time, a subscriber could be unavailable. Message brokers can retain the message until a subscriber is able to consume it. The level of message delivery assurance is handled by the quality of service (QoS), such as *at most once* or *at least once*.

An important advantage of using message queues in your project is feature growth. What begins as a simple application can easily grow into a massive app with a barrage of new features. A fairly straightforward app gets more complex as new requirements arrive, and you need a simple way to add those features with the least disruption to the current application.

Because messaging helps decouple the components of an application, a new application feature can be written to use messaging as a means of communicating with the existing application, and not require changes or downtime for the main application when the new feature is deployed.

Alternatively, you might have a large monolithic application that is part of your *systems of record*, and you want to extend your enterprise and focus on integrating *systems of engagement*. Messaging can be instrumental in integrating the existing systems with new front-end features, by enabling communication between the two without the need for both applications to be running on the same platform or written in the same language.

Many existing systems might already employ some type of enterprise messaging system. Therefore, an optimal design for a new microservice would be one that employs messaging that can be tied into the existing system and communicate with its existing enterprise messaging system.

Some common implementation scenarios for using messaging would be *event notification* and *worker offload*, as shown in Figure 2-9.

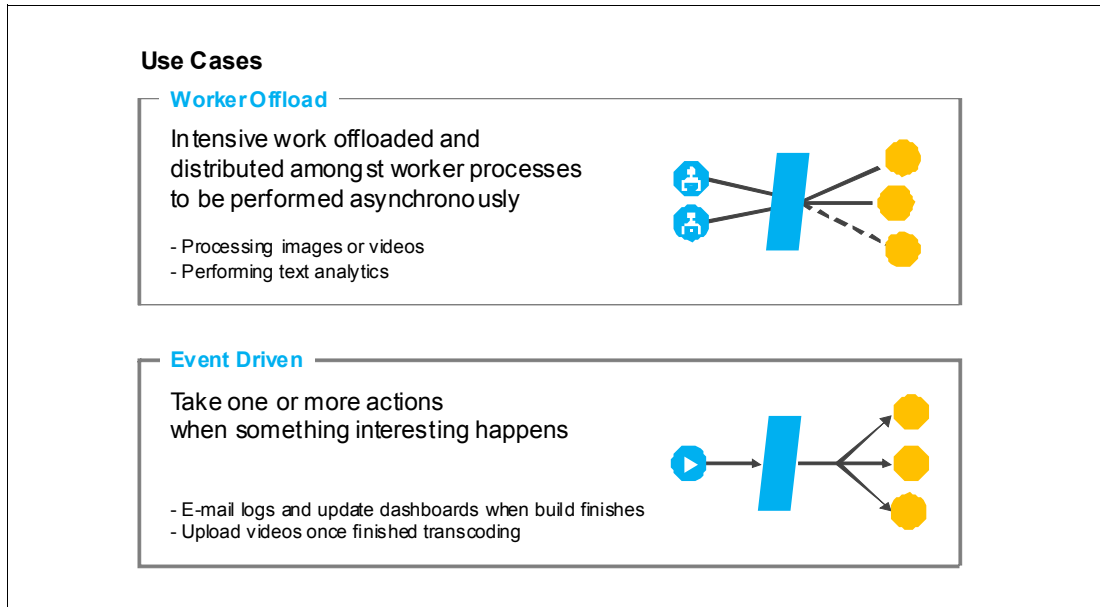


Figure 2-9 Common use cases for messaging

For example, a stock broker application, as shown in Figure 2-10, publishes price updates to a set of trading applications in an *event notification* scenario. The messaging broker manages the storage and distribution of the messages to each interested application. The subscriber does not have to send individual requests, it just subscribes for the topics that it wants, and when those messages are published, they are delivered to the subscriber.

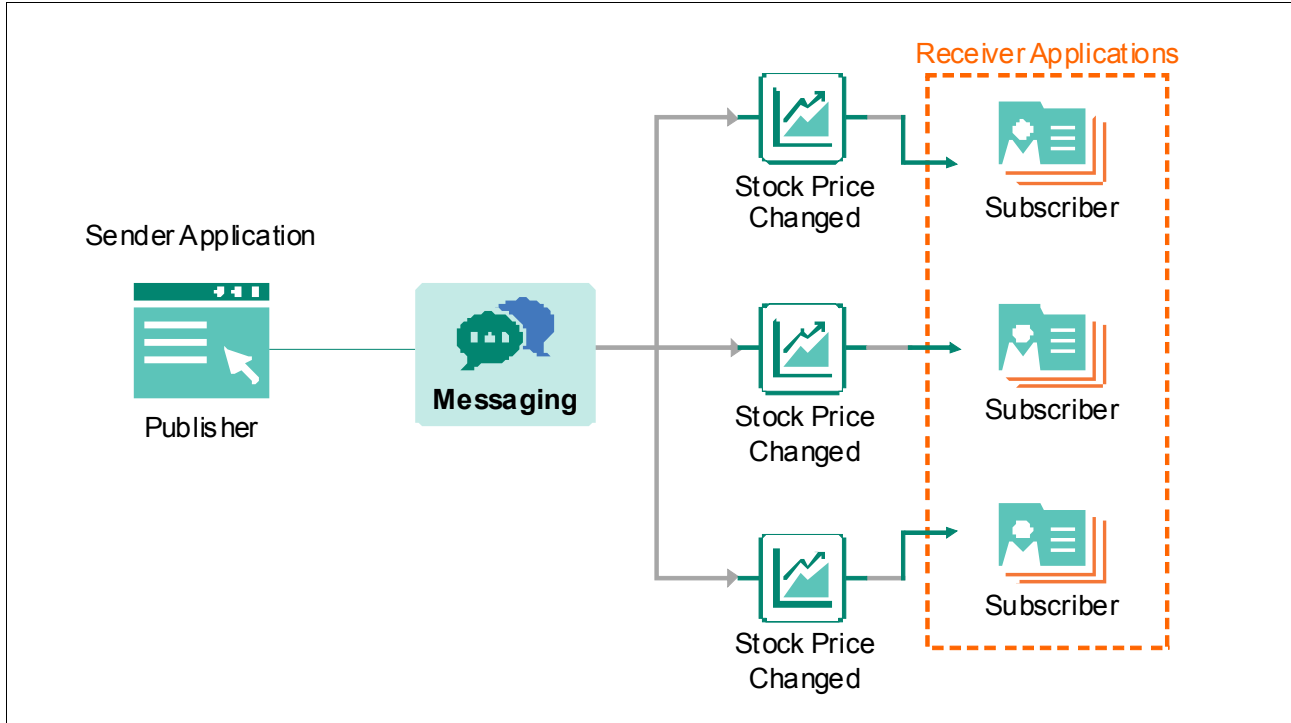


Figure 2-10 Event notification for stock price

Another example of messaging would be a web application that needs to process work while being responsive to the front-end users. When an application has intensive work to be done, it can be offloaded and distributed among worker processes to be performed asynchronously.

Several instances of a back-end worker application could exist, enabling the web application to send a work request for processing by one of several worker application instances. Messaging helps implement a *worker offload* scenario in this case, distributing the work among the worker applications and, if the load increases, additional worker applications can be created without the need to alter the web application.

Messaging helps make applications more flexible by enabling the separate microservices to process asynchronously. Incorporated a messaging service in your microservices design can help ensure that applications remain responsive even when another system is not available or responding fast enough.

The IBM MQ Light service in Bluemix is a fully managed cloud service, so all of the operations activities, such as maintenance, availability, and upgrades, are part of the Bluemix service. IBM MQ Light offers different *qualities of service* to ensure either *at most once message delivery* or *at least once message delivery*.

The acknowledgment of message delivery, and high-availability (HA) deployment of IBM MQ Light in Bluemix, are critical in resilient systems design. See Chapter 4, “Developing microservices in Bluemix” on page 57, and Chapter 7, “Scenario 2: Microservices built on Bluemix” on page 123, for more information about IBM MQ Light, which is available as a Bluemix service.

2.3.3 REST and messaging together

REST and Messaging do not have to be an *either/or* proposition. The two approaches can complement each other. In some cases, it can be optimal to combine applications performing REST calls with applications that use message transport, to achieve a marriage of the two techniques in the context of microservices.

REST can pose a problem when services depend on being up-to-date with data that they don't own or manage. Having up-to-date information requires polling, which quickly can tax a system, especially with many interconnected services. For real-time data, it is often desirable to have the data sent as it changes, rather than polling to ask if it has changed. A polling interval might even mean that you miss a change. In these types of situations, a messaging protocol (such as MQTT or AMQP) can be better than REST to allow real-time event updates.

When an application uses the request/response pattern associated with RESTful services, a broken link means that no collaboration is happening. Similarly, what if your message broker fails? Then messages are not delivered. To safeguard against a disruption in message delivery, an HA configuration can be implemented. You can also scale your applications so that you have multiple instances available to handle messages if one instance becomes unavailable.

In many cases, microservices need to know when changes occur without polling. REST and messaging protocols can act in conjunction with each other. Augmenting REST with messaging can result in a powerful combination, as there might be failures at some point for either method. This helps make the system more dynamic, and provides for looser coupling and more sustainable future growth.

When data flow is sent to subscribers, the onus of storing the data is put on the event recipients. Storing the data in event subscribers enables them to be self-sufficient and resilient. They can operate even if the link to the event publisher is temporarily severed.

However, with every second of the link breakage, they operate on potentially increasingly stale data. In a design where services are simultaneously connected to a message broker, API services fire messages notifying about data changes, and clients who have subscribed for the information can react when they receive the messages.

When messaging is used to augment rather than replace REST, client services are not required to store data. They still need to track the baseline data they obtained through the REST call, and to be able to correlate messages to this baseline.

Messaging that uses publishers and subscribers for topics allows for the possibility to use the topic structure delimiters forward slash (`/`) as a way to sync up REST URLs and topics, enabling mirroring of REST and messaging. This technique can be applied when microservices need to know if the state that they are interested in changes. To do this without constant polling, using the end-point URL as a messaging topic enables another microservice to subscribe for the topic and receive information when the state is changed.

If you use a REST API and add messages for REST endpoints that result in a state change (**POST**, **PUT**, **PATCH**, or **DELETE**), an HTTP **GET** request can be done to fetch a resource. If another service issues an update and publishes a message to notify subscribers that the information has changed, a downstream microservice consuming a REST API endpoint can also subscribe for the topic that matches the endpoint syntax. With this technique, you can reduce REST frequent polling and still have the application up to date.

More information about REST and messaging used together (mirroring), as described previously, is given in 5.1.3, "REST and MQTT mirroring" on page 95.

2.4 The future of microservices

There is much hype around microservices today. The modern systems architecture idea, which can provision small services with full lifecycle control, is here to stay. Developers deploy only the services that are needed as part of a rollout, update them in place, spinning up additional instances as required.

However, as with gaining adoption of any new technology, there are a few broad classes of problems that microservices cannot solve, or that require special attention:

- ▶ **Incorrect requirements.** If we start with requirements that are wrong, it won't matter how fast we can develop, deploy, or scale a system.
- ▶ **Significant operations resource use.** As we break a monolithic system into tens of separate services that need to be built, tested, and communicate with each other, the resources required to manage these services can be daunting. There is not much that exists today in terms of frameworks to support this from an operational perspective.
- ▶ **Substantial DevOps skills required.** The operations challenges of keeping microservices up and available mean that you definitely need high-quality DevOps and release automation skills embedded in your development team.
- ▶ **Implicit interfaces.** As soon as you break an existing monolithic system into collaborating components, you are introducing interfaces between them. Each interface requires maintenance across releases, despite some capabilities available using leading practices, such as compatibility with earlier versions. Leading practices suggest that each interface is treated like a strict contract, and in a distributed world it is not easy to keep contracts fulfilled across releases that are governed by sometimes conflicting business needs.

In the future, we expect to see standards for describing microservices. Currently, you can use any technology to create a microservice, if it exposes JavaScript Object Notation (JSON) or Extensible Markup Language (XML) over the HTTP to provide a REST API. These standards would provide guidelines about how to describe, maintain, and retire microservices.

There is also much maturity in the logical application of microservices architecture to your systems. Over time, rather than decoupling layers in our architectures, we can instead focus on creating conceptual service boundaries that span multiple software systems and multiple process boundaries. We can concentrate on achieving high cohesion vertically within our layers, and low coupling horizontally between data that is conceptually unrelated.

Organizations will become smarter in addressing Conway's law, and will establish cross-functional teams working on each microservice, delivering the same business capability across the enterprise, leveraging their inherent domain expertise in that business capability rather than "reinventing the wheel" within each system boundary.

Finally, developers will stop building systems, but rather compose them from reusable building blocks as requirements dictate.



Microservices and DevOps

This chapter describes how a DevOps method to develop and deliver application code is a key element to the success of a microservices architecture implementation.

In this chapter, we introduce some of the *side effects* that can result with microservices architecture, and provide some guidance about how DevOps can mitigate some of these challenges. We also introduce some of the key capabilities of Bluemix DevOps Services that can be leveraged while implementing a microservices architecture for applications.

This chapter has the following sections:

- ▶ 3.1, “Why you should use DevOps” on page 40
- ▶ 3.2, “DevOps capabilities for microservices architecture” on page 41
- ▶ 3.3, “Microservices governance” on page 47
- ▶ 3.4, “DevOps capabilities: Testing strategies for microservices” on page 51

3.1 Why you should use DevOps

Chapter 1, “Motivations for microservices” on page 3 explains that one of the most common business drivers for adopting microservices is business agility. Microservices allow you to respond quickly and incrementally to business opportunities. Incremental and more frequent delivery of new capabilities drives the need for organizations to adopt DevOps practices.

3.1.1 Defining DevOps

DevOps is the set of concepts, practices, tools, and team organizational structures that enable organizations to more quickly release new capabilities to their clients. Organizations that adopt DevOps are more easily able to release and monitor their microservices. They are able to respond quickly to new requirements, or to problems that occur in production. DevOps commonly includes the following processes:

- ▶ Agile practices (<http://agilemanifesto.org/principles.html>)
- ▶ Continuous integration
- ▶ Release automation
- ▶ Functional unit testing
- ▶ System integration testing
- ▶ Service and infrastructure monitoring

3.1.2 DevOps is a prerequisite to successfully adopting microservices

As monolithic applications are incrementally functionally decomposed into foundational platform services and vertical services, you no longer just have a single release team to build, deploy, and test your application. Microservices architecture results in more frequent and greater numbers of smaller applications (microservices) being deployed. DevOps is what enables you to do more frequent deployments, and to scale to handle the growing number of new teams releasing microservices.

DevOps is a prerequisite to being able to successfully adopt microservices at scale in your organization. Teams that have not yet adopted DevOps must invest significantly in defining release processes and corresponding automation and tools. This is what enables you to onboard new service teams, and achieve efficient release and testing of microservices.

Without it, each microservice team must create their own DevOps infrastructure and services, which results in higher development costs. It also means inconsistent levels of quality, security, and availability of microservices across teams.

3.1.3 Organizing teams to support microservices

There are typically two levels for implementing microservices. For larger organizations or more complex applications, there can be more. Teams align around business function, and own all lifecycle phases of the service in the application:

- ▶ Requirements
- ▶ Testing
- ▶ Deployment
- ▶ Operational monitoring of the service

3.1.4 Organize a DevOps team to support other microservices teams

As you begin to reorganize teams to align with business components and services, also consider creating microservices DevOps teams who provide those cross-functional teams with tool support, dependency tracking, governance, and visibility into all microservices. This provides business and technical stakeholders greater visibility into microservices investment and delivery as microservices move their lifecycle.

Smaller, focused, autonomous teams are what allow you to focus on a core service capability that can be independently deployed more frequently. The DevOps services team provides the needed visibility into what services are being deployed, used by other teams, and ultimately used by clients. This loosely coupled approach provides greater business agility.

3.2 DevOps capabilities for microservices architecture

DevOps achieves the goal of continuous delivery, which is required for a microservices architecture, through its three pillars approach:

- ▶ Automation
- ▶ Standardization
- ▶ Frequent code releases

Automation on steroids is the theme of DevOps. As depicted in Figure 3-1, every traditional *hand off* point is automated in a DevOps approach. Automation begins from provisioning of the infrastructure and configuration of the platform for the workloads. This is followed by the automation of application deployment, database changes, and other application-related configurations.

Finally, maximum benefits in a DevOps approach are gained through test automation in the form of unit tests, system integration tests, and user acceptance tests. Because microservices represent small, composable business functions, the level of automation possible through a DevOps approach aligns perfectly for its development lifecycle.

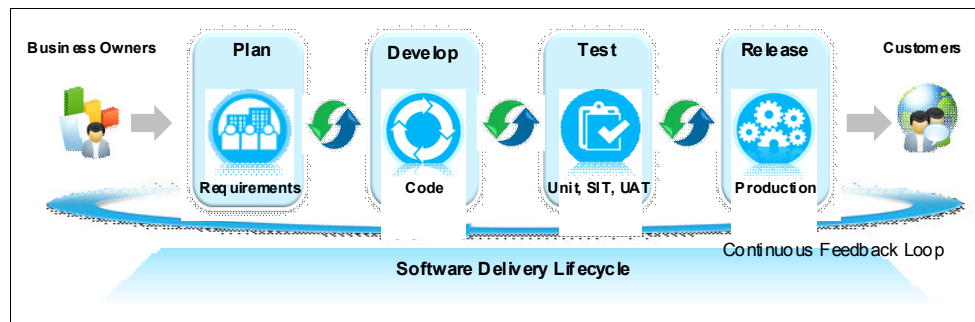


Figure 3-1 DevOps approach to software development: Continuous delivery

The aspect of standardization in DevOps can easily be confused with the flexibility that a microservices architecture provides with regards to technology and data store stack. In a DevOps world, standardization to reduce risk refers to the following factors:

- ▶ Using standard platforms
- ▶ Reliable and repeatable process
- ▶ High level of confidence by the time you are ready to release

Frequent releases keep applications relevant to business needs and priorities. Smaller releases means lesser code changes, and that helps reduce risk significantly. With smaller release cycles, it is also easier to detect bugs much earlier in the development lifecycle. Quick feedback from the user base also supports code adaptability. All of these characteristics of a DevOps approach augur well for microservices development.

Several common pain points that affect a traditional software delivery, as depicted in Figure 3-2, are mitigated to a large extent with the use of a DevOps approach for microservices development.

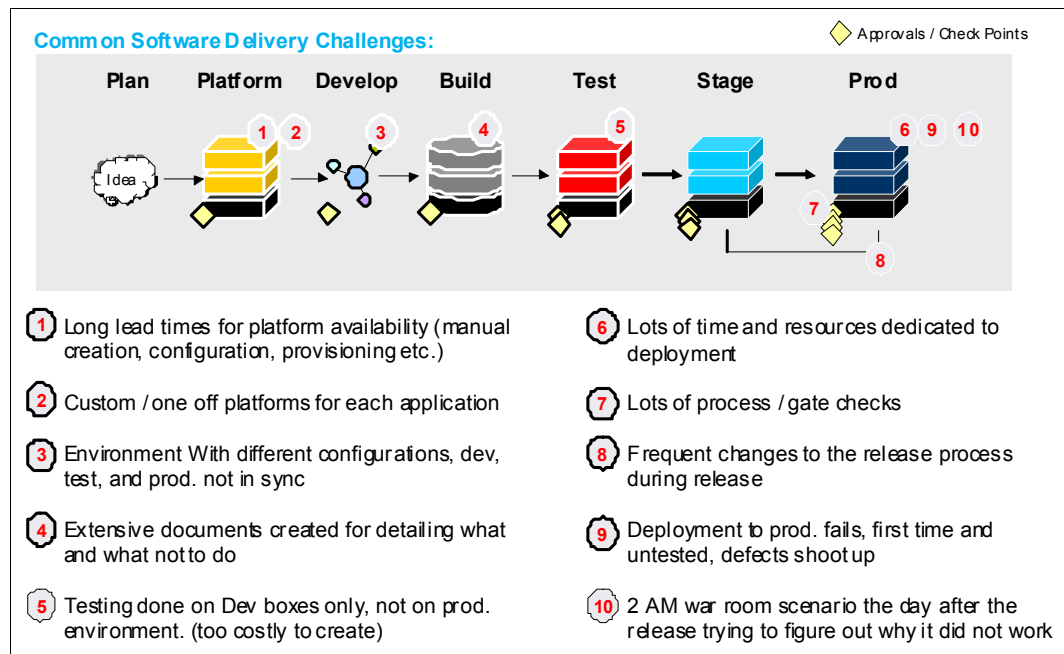


Figure 3-2 Common challenges with traditional software delivery

IBM DevOps framework addresses continuous delivery through a series of *continuous* steps, as shown in Figure 3-3.

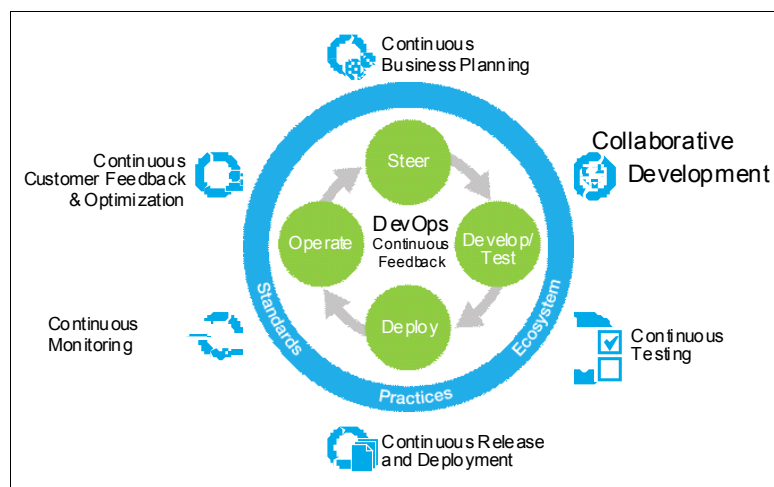


Figure 3-3 IBM DevOps framework to continuous software delivery

IBM Bluemix DevOps Services is a software as a service (SaaS) on the cloud that supports continuous delivery. With IBM Bluemix DevOps Services, you can develop, track, plan, and deploy software in one place. From your projects, you can access everything that you need to build all types of applications (apps). To simplify team work, use the collaboration tools. After you build an app, you can deploy it to the IBM Bluemix cloud platform. You can go from source code to a running app in minutes.

IBM Bluemix DevOps Services provide the following capabilities:

- ▶ Agile planning, through the Track & Plan service
- ▶ A web integrated development environment (IDE) for editing and managing source control
- ▶ Source control management (SCM) through Git, Jazz SCM, or GitHub
- ▶ Automated builds and deployments, through the Delivery Pipeline service

The following sections describe these capabilities, and explain how they are useful in context of microservices.

3.2.1 Continuous business planning

Continuous business planning, also known as *continuous steering*, helps to continuously plan, measure, and bring business strategy and customer feedback into the development lifecycle. It provides the tools and practices to help organizations *do the correct things* and focus on activities where they can gain most value:

- ▶ Attacking the high-value and high-risk items first
- ▶ Predicting and quantifying the outcomes and resources
- ▶ Measuring honestly with distributions of outcomes
- ▶ Learning what customers really want
- ▶ Steering with agility.

Continuous business planning employs lean principles to start small by identifying the outcomes and resources needed to test the business vision and value, to adapt and adjust continually, measure actual progress, learn what customers really want, shift direction with agility, and update the plan.

Continuous business planning has the following benefits:

- ▶ Helps you continuously plan business needs and prioritize them
- ▶ Integrate customer feedback with business strategy
- ▶ Align customer feedback into the development lifecycle as business needs
- ▶ Focus on *doing the correct things*
- ▶ Prioritize business needs that adds the most value

Some of the most commonly used industry tools and offerings for continuous business planning are described in the following list:

- ▶ IBM Rational Team Concert™
- ▶ IBM Rational DOORS® Next Generation
- ▶ Collaborative Lifecycle Management as a service (CLMaaS)
- ▶ JIRA
- ▶ Kanboard.net

3.2.2 Continuous integration and collaborative development

Continuous integration refers to the leading practice of integrating the code of the entire team regularly, to verify that it works well together. Continuous integration is a software development practice that requires team members to integrate their work frequently. Integrations are verified by an automated build that runs regression tests to detect integration errors as quickly as possible. Teams find that this approach leads to significantly fewer integration problems, and enables development of cohesive software more rapidly.

Collaborative development enables collaboration between business, development, and quality assurance (QA) organizations (including contractors and vendors in outsourced projects spread across time zones) to deliver innovative, quality software continuously. This includes support for polyglot programming, multi-platform development, elaboration of ideas, and creation of user stories, complete with cross-team change and lifecycle management.

Collaborative development includes the practice of continuous integration, which promotes frequent team integrations and automatic builds. By integrating the system more frequently, integration issues are identified earlier, when they are easier to fix. The overall integration effort is reduced using continuous feedback, and the project shows constant and demonstrable progress. Continuous integration has the following benefits:

- ▶ Each developer integrates daily, leading to multiple integrations per day.
- ▶ Integrations are verified by automated builds that run regression tests to detect integration errors as quickly as possible.
- ▶ Small, incremental, frequent builds help with early error detection, and require less rework.
- ▶ Continuous integration is a leading practice to make a global collaborative development work successfully.

Collaborative development has the following benefits:

- ▶ Bringing together customer and IBM team stakeholders toward a partnered goal
- ▶ Focused on delivering a tangible and functional business outcome
- ▶ Working within a time-boxed scope
- ▶ Using common tools and process platform (customer, traditional, or IBM Cloud)

The following list describes some of the most commonly used industry tools and offerings for continuous integration and collaborative development:

- ▶ Rational Collaboration Lifecycle Management
- ▶ Rational Lifecycle Integration Adapter
- ▶ Rational Developer for System z®
- ▶ IBM Worklight® Studio
- ▶ IBM UrbanCode™ Build
- ▶ Git
- ▶ Jenkins
- ▶ Gerrit
- ▶ Bugzilla

3.2.3 Continuous testing

Continuous testing reduces the cost of testing while helping development teams balance quality and speed. It eliminates testing bottlenecks through virtualized dependent services, and simplifies the creation of virtualized test environments that can be easily deployed, shared, and updated as systems change. These capabilities reduce the cost of provisioning and maintaining test environments, and shorten test cycle times by enabling integration testing earlier in the lifecycle.

The following list describes some of the benefits of continuous testing:

- ▶ Avoidance of unexpected disruption to business
- ▶ More reliable applications
- ▶ Proactive problem prevention
- ▶ More effective and quicker testing
- ▶ Less rework
- ▶ More efficient and happier workers
- ▶ Lower operational costs and improved topline

The following list describes some of the most commonly used industry tools and offerings for continuous testing:

- ▶ Highly Available Test Chassis (HATC)
- ▶ Delphix (Database Virtualization)
- ▶ Selenium
- ▶ JUnit
- ▶ Cucumber

3.2.4 Continuous release and deployment

Continuous delivery is a series of practices designed to ensure that code can be rapidly and safely deployed to production by delivering every change to a production-like environment, and ensuring that business applications and services function as expected through rigorous automated testing. Because every change is delivered to a staging environment using complete automation, you can have confidence that the application can be deployed to production with the “push of a button” when the business is ready.

Continuous release and deployment provides a continuous delivery pipeline that automates deployments to test and production environments. It reduces the amount of manual labor, resource wait-time, and rework by using push-button deployments that enable a higher frequency of releases, reduced errors, and end-to-end transparency for compliance.

Continuous delivery has the following benefits:

- ▶ The leading practice of deploying code rapidly and safely into production-like environments is followed.
- ▶ Deployments start automated tests to ensure that components perform business functions as expected.
- ▶ Every change is automatically deployed to staging.
- ▶ Deployments are on-demand and self-service.
- ▶ Applications can be deployed into production with the push of a button, when business is ready.

The following list includes some of the most common industry tools and offerings for continuous release and deployment used in concert with each other:

- ▶ IBM UrbanCode Deploy with Patterns
- ▶ IBM UrbanCode Release
- ▶ Chef
- ▶ Puppet
- ▶ Juju
- ▶ Docker
- ▶ Ansible
- ▶ Salt

3.2.5 Continuous monitoring

Continuous monitoring offers enterprise-class, easy-to-use reporting that helps developers and testers understand the performance and availability of their application, even before it is deployed into production. The early feedback provided by continuous monitoring is vital to lowering the cost of errors and change, and for steering projects toward successful completion.

In production, the operations team manages and ensures that the application is performing as wanted, and that the environment is stable using continuous monitoring. Although the Ops teams have their own tools to monitor their environments and systems, DevOps principles suggest that they also monitor the applications. They need to ensure that the applications are performing at optimal levels, down to levels lower than system monitoring tools would allow.

This requires that Ops teams use tools that can monitor application performance and issues. It might also require that they work with Dev to build self-monitoring or analytics gathering capabilities directly into the applications being built. This would allow for true end-to-end monitoring, continuously.

Continuous monitoring has the following benefits:

- ▶ The leading practice of managing the infrastructure, platform, and applications to ensure that they are functioning optimally is followed.
- ▶ Thresholds can be set for what is considered optimal.
- ▶ Any untoward incident triggers an automatic alert or remediation.
- ▶ Monitoring logs are used for operational analytics.
- ▶ Developers can also self-monitor applications and look at real-time reports.

Browsing through the Bluemix catalog, you will come across the Monitoring and Analytics service. This service can be used with any application using the run times supported by Bluemix, such as IBM Liberty, Node.js, and Ruby.

Note: At the time of writing this book, IBM Liberty provides the most metrics, followed by Node.js and Ruby. More support for additional CF build packs, and Containers and virtual machines (VMs) are in process.

This service offers various additional features besides log collection, but log collection comes in the no initial charge plan. With this service, you can only view logs from the past 24 hours. Therefore, if you need your logs to be persisted longer than 24 hours, consider one of the third-party services. The nice thing about this service is that it offers great searching functionality, so you can easily identify patterns in your logs.

The following list includes some of the most commonly used industry tools and offerings for continuous monitoring:

- ▶ SmartCloud Application Performance Management (APM)
- ▶ IBM Service Engage
- ▶ Nagios
- ▶ Nmon
- ▶ Cacti
- ▶ Logstash
- ▶ Fluentd
- ▶ New Relic

3.2.6 Continuous customer feedback and optimization

Efficient DevOps enables faster feedback. Continuous customer feedback and optimization provides visual evidence and full context to analyze customer behavior, pinpoint customer struggles and understand customer experiences using web or mobile applications.

Experimentation, learning through first-hand client experiences, and continuous feedback are critical to being successful in this new world. Whether it is entering a new market space or evolving a current set of capabilities, like we did, delivering a minimally viable product, learning and pivoting in a responsive way challenges the status quo. This is DevOps in action, showing how instrumented feedback, included with sponsor users connected into cross functional feature teams able to act quickly, can be an incredible combination.

The trick is to evolve in a disruptive yet healthy way that enables the team to innovate while remaining connected with the system of record team, where the data and transactions live. Finding this balance and a pattern that works is essential to capturing the value of hybrid cloud and mobile.

Continuous customer feedback and optimization provides the following benefits:

- ▶ Provide clients the ability to gain insight with intuitive reporting and optimized web, mobile, and social channels
- ▶ Empower stakeholders, including end-users
- ▶ Provide and assess important feedback, from idea through live-use
- ▶ Respond with insight and rapid updates
- ▶ Maintain and grow competitive advantage

The following list includes some of the most commonly used industry tools and offerings for continuous customer feedback and optimization:

- ▶ IBM Tealeaf® Customer Behavior Analysis Suite
- ▶ IBM Digital Analytics
- ▶ Mobile First Quality Assurance
- ▶ Open Web Analytics (OWA)
- ▶ Webalizer
- ▶ W3Perl

3.3 Microservices governance

In the world of developers, governance many times is a rather unpopular term. Just for the fact that developers seek unhindered freedom to experiment with new languages, patterns, frameworks, data stores, and other innovative aspects of IT development. Operations people, alternatively, can be very uncomfortable when confronted with that level of experimentation from development. There are some concerns about microservices architecture being devoid of any governance, or having a *lightweight governance* with regards to service-oriented architecture (SOA).

It is worth clarifying that microservices architecture is still an architectural practice, and can only be successful with the necessary discipline, use of tools, and leading practices. You can make an argument that much more design and project management discipline can be necessary with a microservices architecture than is needed for a monolithic approach to make it successful. Microservices architecture brings about some unique aspects to governance that can actually draw a balance between unstructured versus controlled chaos.

3.3.1 Centralized versus decentralized governance

Any enterprise-grade application and its lifecycle without governance would sound scary to the CIO office and the like. What a microservices architecture brings about is the concept of decentralized governance rather than centralized governance. Microservices promote a more polyglot model regarding technology stack for supported languages, tools, and data stores.

Alternatively, in a traditional monolithic model of centralized governance, there is a push toward enterprise-level standardization of technology, platform, and service providers. One of the perceived drawbacks of monolithic architecture is *excessive governance* that can inhibit innovation, but a balance has to be drawn between excessive and none at all.

Reusability of assets and tools, rather than centralized forceful standards, seems to be the microservices mantra. In addition to asset reuse, some of the other salient features of microservices are functional isolation, localized scalability, and ease of maintenance. All of features come as part of being a self-contained, fully consumable, fine-grained application with decentralized governance that satisfies a specific micro business function.

Silo-driven development is often frowned upon in the architecture community. However, large enterprises can lack the agility and executive support to spin off innovative projects to develop proof points of a certain technology that might drive business value. Therein lies the advantages of what some call *skunk work*. Although microservices architecture is not a throwaway type of work, it does have the advantage of realizing business value quickly, or failing quickly, so other aspects can be considered as part of a solution.

For these reasons, a *siloed approach* can have a reason to exist. There are arguments in the industry that often describe silos as logical pieces of a larger linear process that ultimately have a common goal of solving a business problem. But the larger argument for such a method is budgetary constraints. Enterprises seldom allocate funding and time to fully understand a specific business problem well enough to deliver a solution as a service.

Decentralization of governance can help mitigate such situations. The enterprise has time constraints on full-scope product deployment, and also has limited budgets on incremental development and delivery. The key value with decentralized governance is that a proof point can be quickly established, with some knowledge of return on investment (ROI) and a time frame for that, without incurring extensive risks regarding time and money.

One of the ramifications of decentralized governance is the concept of *build it and run it*. This is a disruptive model to a traditional organization built on functional teams not based on business domains. A microservice team is responsible for all aspects of the service delivery, from build to run to fix. From an architectural thinking perspective, it is not enough to design for functional delivery.

In addition, non-functional and support models become increasingly important. The same team that designed a microservice can end up being the one answering support calls over the weekend. This is clearly an organizational transformation issue, as illustrated in 3.3.2, “Enterprise transformation for microservices” on page 50.

Decentralization of data stores

A typical microservices architecture also decentralizes data store decisions. Whether the reason is *best of breed* for the workload, or licensing, or just a skills decision, the choice of a data store technology is left up to the microservices team. It is not federally dictated, as in a monolithic model.

Decentralizing data stores (Figure 3-4) does have its implications concerning governance. Managing updates can be quite challenging with disparate data stores. Multiple data stores means several network calls, which increases the risk of a failed update or poorly performing updates. The importance of appropriate design thinking, especially about eventual data consistency for microservices, cannot be stressed enough, if it is to be a useful implementation.

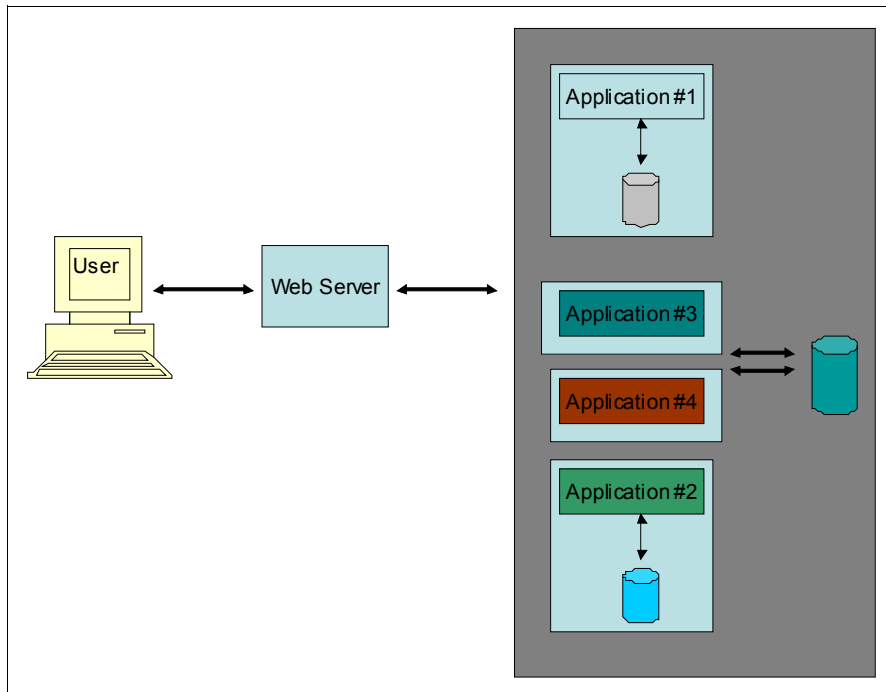


Figure 3-4 Decentralized data store for microservices

Other aspects to be considered about centralized versus decentralized governance are monitoring, logging, single sign-on, backups, operational metrics, and reporting.

One of the major challenges that an enterprise embarking on a microservices journey can encounter is to identify common services across domains, and the scope and size of each microservice. Typically, functions necessary for logging have centralized governance. All domains should use a central logging service.

That way, there is consistency regarding the style and content of logging, and the logging is centrally available for any analytics and reporting. Sign-on is another function that has centralized governance, and typically it is integrated to some form of an enterprise-wide directory service.

Other system management capabilities, such as monitoring and backup, are also part of centralized governance. Information about monitored units, the frequency of monitors and alerts, and the reporting about any issues and their workloads, are all determined and managed centrally. Backup frequency, testing the cadence of restore scripts, and data retention procedures are also all part of centralized governance.

3.3.2 Enterprise transformation for microservices

Embarking on a microservices transformation journey is not a trivial endeavor. As depicted in Figure 3-5, microservices adoption is really an evolution into an overall enterprise transformation. You cannot simply “think through” microservices design aspects, and then expect the implementation to be successful or to provide return on investments. You must also perform other necessary steps, such as a process transformation through DevOps, and an organizational transformation.

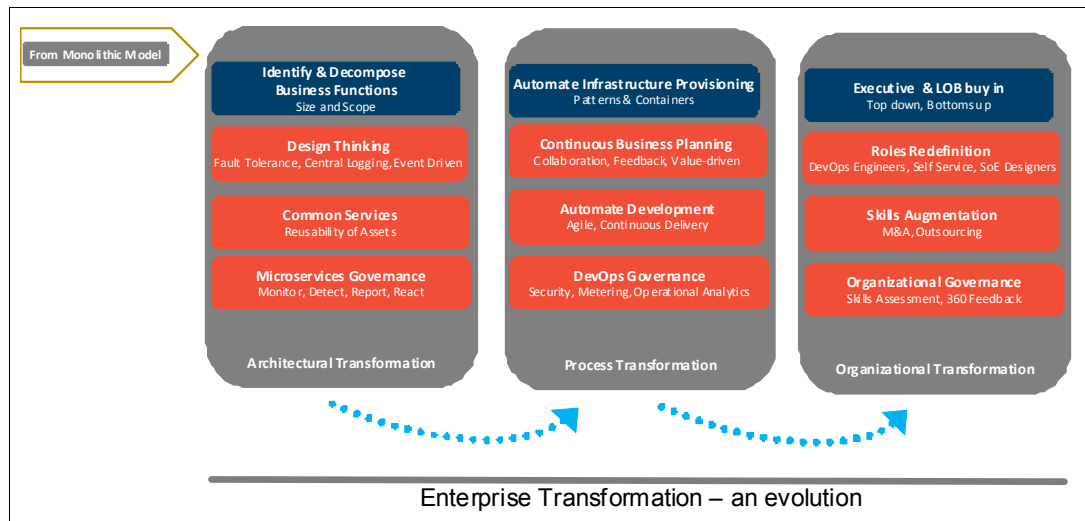


Figure 3-5 Microservices transformation

Architectural transformation begins with identifying and decomposing business functions into independently manageable units or workloads. It is certainly a disruption in traditional architectural thinking methods, and it requires a few iterations of design planning to get it correct.

As a growing number of self-contained composable services are identified, there is also an element of service management complexity due to rapidly increasing volumes of workloads to manage. Source code management, deployment, and release managements and testing strategies, all become slightly more complicated than with a monolith.

This is where the process transformation of DevOps becomes essential for a microservices architecture to have any degree of success. From automation of environment provisioning to automation of deployments and releases, all play a key part in reducing manual resource demands, number of errors, and a truly agile way to deliver software. With the level of automation involved, there is also an infusion of several new tools and scripting languages that come into play.

Just as DevOps transformation is a key driver for microservices architecture to succeed, an organizational transformation is essential for the process transformation to succeed. With DevOps, new job roles are created, and existing ones are redefined. Skills training and a new way of thinking become crucial for success. With large enterprises, this means that there might be new partnerships with vendors and service providers to supplement skills.

Because microservices teams are designed to align with its architecture, teams will also be self-contained. The concept of *build it, run it* is often talked about with microservices. There is always resistance to change with large enterprises, so executive commitment and coaching becomes critical during transformation.

3.4 DevOps capabilities: Testing strategies for microservices

Microservice architecture is considered an evolutionary way to build an IT system, one which comes with several advantages. However, it also introduces typical challenges, which can turn into problems if not appropriately addressed. Testing is one of the biggest challenges on the list.

This section describes the general testing methods, and the approach to build a sufficient testing strategy that contains the following highlights:

- ▶ Considerable testing methods
- ▶ Creating sufficient testing strategy

3.4.1 Considerable testing methods

This section describes some common testing methods to be considered as part of a holistic testing strategy for a microservice-powered system. The following list includes these common testing methods:

- ▶ Unit testing
- ▶ Integration testing
- ▶ Component testing
- ▶ Contract testing
- ▶ End-to-end testing
- ▶ Performance testing

The following sections describe each of these methods in detail.

Unit testing

A *unit test* in microservices is the same as in other systems, and can be understood as running the smallest piece of testable software to determine whether it behaves as expected. Although there is no exact definition of how big the testable piece against which we run the unit testing should be, a typical unit to be tested should have some of the following common elements:

- ▶ The unit is a low-level piece of the system, and focuses on a specific small scope of the system. It might be a class of code, a single function, or a group of functions.
- ▶ Testing against the unit should be significantly faster than other kinds of tests.
- ▶ Unit test script is usually written by the developers who built the code that is under test.

This testing is where we hope to detect most of the bugs in the system, and it is the most frequently run testing compared to other kinds of tests. The primary goal of unit testing is to give us fast feedback about whether the implemented piece of code is good in isolation.

This test is also critical in the code refactoring activities, where small-scoped tests can prove timely to help ensure the quality of the code restructuring as you proceed with development. Unit testing is also a principal tool in a test-driven development approach, helping to obtain fast feedback about the system we build from the beginning.

Integration testing

Integration testing is a testing method that collects relevant modules together, and verifies that they collaborate as expected. Integration testing accomplishes this by exercising communication paths among the relevant modules to detect any incorrect assumptions that each module has about how to interact with other modules under test.

In microservice architectures, this testing is typically used to verify interactions between the layers of integration code, and any external components that they integrate to. The external components might be data stores, other microservices, and so on.

The integration test against data stores' external components, or *persistence integration test*, ensures that the schema used by the code matches the real one in the external data store. In many cases, the object-relational mapping technique might be used to convert data. This might create more complications in the persistence integration test, which needs to be structured to consider the dependency of the test with the configuration on the tool to ensure that the data makes a full round trip.

Also, data stores can exist across a network partition that might cause timeouts and network failures. Integration tests should attempt to verify that the integration modules handle these failures gracefully.

Another important external integration test is the test against communication between two microservices. Very often, a proxy component is used to encapsulate messages passing between two remote services, marshalling requests and responses from and to the modules that actually process the request. The proxy is usually accompanied by a client that understands the underlying protocol to handle the request-response cycle.

The integration testing in this case needs to detect any protocol-level errors, such as incorrect Secure Sockets Layer (SSL) handling, missing HTTP headers, or mismatches of request-response body. Special case error handling should also be tested to ensure that the service and the client employed respond as intended in exceptional circumstances.

Because the integration testing relies on a certain set of data being available, one technique that can be used is that both parties agree on a fixed set of harmless representative data that is guaranteed to be available as needed.

Sometimes it is difficult to trigger abnormal behaviors, such as a slow response from an external component. In this case, the test might use a stub version of the external component as a test harness that can be configured to fail at will.

Component testing

A *component* is part of a larger system that is encapsulated, coherent, and independently replaceable to other parts. A *component test* limits the scope of the exercised software to a portion of the system under test, manipulating the system through internal code interfaces, and using techniques to isolate the code under test from other components.

By limiting the testing scope within a single component, and isolating it from its peers, you can eliminate “noise” introduced by any possible complex behavior of the external components that would affect the test result. This enables you to focus thoroughly on testing the behavior encapsulated by that particular component. That isolating also results in a faster test, and provides a controlled testing environment where error cases can be retriggered at will.

In a microservice architecture, the components are considered the services themselves. The isolation for testing a service is usually achieved by representing the associated external collaborators with their simulators (for example, fake, spies, dummy, and stubs), through a set of internal API endpoints to probe the service.

The following list includes two common component testing options in microservices architecture:

- ▶ **In-process component tests.** Allow comprehensive testing while minimizing moving parts.
This test creates a full in-memory microservice, with in-memory service simulators and data stores, that ultimately helps to eliminate any network traffic, lead to faster test execution time, and minimize the number of moving parts to reduce build complexity. The downside of this technique is that the work products under test must be altered accordingly for testing purposes.
- ▶ **Out-of-process component tests.** Exercise the fully deployed work products.
This test is against a microservice deployed as a separate process. It enables exercising for more layers and integration points. The interactions are through real network calls so that there is no need to alter the component under test with test-specific logic. This approach pushes the complexity into the test harness, which is responsible for starting and stopping external stubs, coordinating network ports, and configuration.
The test execution time consequently increases due to the use of the real data store and network interaction. However, this approach is more appropriate for testing microservices, which have a complex integration, persistence, or startup logic.

Contract testing

A contract is conceptually a proliferation of interfaces between two services. A *contract test* is a test at the boundary of an external service to verify that it meets the expectation of the contract formed between the service and its consuming services. The contract consists of expected requirements for input and output data structures, performance, concurrency characteristics, and so on. A component can change over time, and it is fundamental that the contracts of its consumers continue to be satisfied after the changes.

In microservice architecture, the contract is fulfilled through a set of APIs exposed by each service. The owner of its consuming services is responsible for writing an independent test suite, which verifies only the aspects of the producing service that are in the contract. That contract test suite should be packaged and runnable in the build pipelines for the producing services, so that the owner of the producing service might be informed of the effect of the changes on their consumers in a timely manner.

The overall contract can be defined by summing up all of the consumer contract tests. The consumer-driven contracts form a discussion point with the service owner, based on which the test can be automated to provide an indication of the readiness of the API.

Several tools exist to help writing contract tests, such as Pact and Pacto. You can find more information about these tools on the following websites:

- ▶ Pact: <https://github.com/realstate-com-au/pact>
- ▶ Pacto: <https://github.com/thoughtworks/pacto>

End-to-end testing

An *end-to-end test* is a test verifying that the system as a whole meets business requirements and achieves its goals. The test accomplishes this by exercising the entire system, from end-to-end.

Because of the naturally autonomous nature of a microservice architecture-powered system, it usually has more moving parts for the same behavior as other systems. End-to-end tests can help to cover the gaps among the services. This test provides additional confidence in the correctness of messages passing between the services. The test also ensures that any extra network infrastructure, such as firewalls, proxies, and load-balancers, are properly configured, as well.

Over time, a microservice architecture-powered system usually evolves, with services being merged or split to address the business problem domain, that also evolves. End-to-end tests also add value by ensuring that the business functions provided by that kind of system remain intact during such possible large-scale architectural refactorings.

The test boundary for end-to-end tests is much larger than for other types of tests, because the goal is to test the behavior of the fully integrated system. End-to-end testing interacts at the most coarse granularity possible, and is the most challenging testing type to build and maintain. As the test scope increases, so does the number of moving parts, which can introduce test failures. These problems do not necessarily show that the functionality under test is broken, but rather that some other problems have occurred.

Ideally, you can manage all of the external services of the system to be included in your end-to-end testing. However, in many cases, the services are managed services from a third party, which limits you from building a side-effect-free end-to-end test suite. The results are that the test fails for the factors that are out of your control. In such cases, it can be more beneficial to stub the external services and sacrifice some end-to-end confidence, but in return gain stability in the execution of the test suite.

Due to the additional complexity of writing end-to-end tests, the test team should selectively decide how extensively this test should be used in their testing strategy. The following list includes some guidelines to make an effective end-to-end test suite:

- ▶ Make the end-to-end test as small as possible. The main goal of performing end-to-end testing is to make sure that all services that make up the system instrumented work well together as a whole. An adequate level of confidence in the behavior of the system might be more beneficially achieved by focusing on other types of tests rather than putting too much effort into end-to-end testing.
- ▶ Avoid dependency on pre-existing data. Because data is most likely going to be changed and accumulated as the testing goes on, relying on a set of pre-existing data might cause nonsense failures. These breakdowns are not an indication of the failure of the system under test. A solution for this is to automate the data management so that the system can initialize its data upon the actual test execution.
- ▶ Mimic production environment in a fully automated manner. The deployment complexity of a microservice architecture system might result in many difficulties in replicating the environment for an end-to-end test. One solution might be using an infrastructure-as-code (IaaS) approach by using some automation frameworks, such as Chef or Puppet, which might help build environments as needed, in a reproducible manner.
- ▶ Make the test consumer-driven. Model the tests around the consumers of the service in terms of their personas, in addition to the patterns that they navigate through the system under test. You might even test in production with feature switches.

Performance testing

Performance tests should run against service-to-service calls, and against a particular service in isolation, to track microservices individually and keep performance issues under control. Usually, a good way to start is with identifying major journeys in the system, put each of the services and also all services under gradually increasing numbers of requests.

This way we can track how the latency of the calls varies through increasing load. Ideally, the component or system under performance test should be placed into an environment that is as similar to production as possible. In this way, the test result might be more indicative of what the production environment would look like in a similar situation.

Performance testing also should be done jointly with monitoring the real system performance. It should ideally use same tools as the ones in production for visualizing system behavior, because it might make it easier to compare the two. A comprehensive centralized logging tool is a good example for that with which we can see precisely how the microservices are acting.

The following list describes some outlines to consider for completing performance testing against a microservice system:

- ▶ Use as much real data as possible.
- ▶ Load the test with real volume similar to that expected on production.
- ▶ Push the test through in as close to production conditions as possible.
- ▶ Do the performance test early, and regularly, not just once immediately before moving to production.
- ▶ Consider continuing to gain performance insights and monitoring performance in production as appropriate.

3.4.2 Building a sufficient testing strategy

Automated testing has advanced significantly over time as new tools and techniques keep being introduced. However, challenges remain as for how to effectively and efficiently test both functional and non-functional aspects of the system, especially in a distributed model. With more independent components and patterns of collaborations among them, microservices naturally add a new level of complexity into the tests.

It is obvious that identifying problems before going to production is a critical need, but it is the fact that we cannot spot and detect all problems before we move our products to production. Performing testing jointly with an appropriate deployment model and tool, might push the number of problems when the system is operated on production closer to zero. Smoke test suite and blue/green deployment are good examples of techniques for achieving it.

Understanding what different types of tests we can run, and what their advantages and disadvantages are, is fundamental to help us build a sufficient testing strategy. This enables us to balance the needs of getting our services into production as quickly as possible versus making sure that they are of sufficient quality. Bringing all of them together, we can highlight the following considerations in order to have a holistic approach for testing a microservice system:

- ▶ Build tests toward the consumer-driven contracts, to center tests around consumers personas and navigation journeys through the system.
- ▶ Optimize tests for fast feedback for any type of testing.
- ▶ Optimize your test strategy so that the number of required end-to-end tests is as few as possible.
- ▶ Consider performance and other non-functional tests regularly in earlier stages.



Developing microservices in Bluemix

This chapter describes building a microservice application on IBM Bluemix. Using a well-rounded platform for building, running, and managing your applications (apps) can alleviate much of the operational, networking, and other infrastructural resource costs of the microservice architecture.

A well-rounded platform also enables continuous delivery capabilities for the entire *application lifecycle management* (ALM). Bluemix aims to simplify software development and management on the cloud by providing a complete set of flexible run times, integrated services, and DevOps tools that include development, testing, scaling, monitoring, deployment, and logging.

The aim of this chapter is to first provide an introduction to Bluemix, describe developing and deploying microservices using Bluemix DevOps capabilities, and outline some considerations when leveraging Bluemix for microservices.

This chapter has the following sections:

- ▶ 4.1, “Bluemix introduction” on page 58
- ▶ 4.2, “Developing microservices using Bluemix DevOps” on page 68
- ▶ 4.3, “Deployment, testing, monitoring, and scaling services in Bluemix DevOps” on page 76
- ▶ 4.4, “Communication, session persistence, and logging in Bluemix” on page 81
- ▶ 4.5, “Considerations and opportunities” on page 86

4.1 Bluemix introduction

Bluemix is an open-standards cloud platform for building, running, and managing applications. It provides developers access to IBM and third-party services for integration, security, transaction, and other key functions.

Providing a platform as a service (PaaS) environment as one of its run times, along with containers and virtual machines (VMs), Bluemix leverages the Cloud Foundry open source technology to accelerate new application development and DevOps methodologies. Additionally, Bluemix provides optimized and elastic workloads, enables continuous availability, and simplifies delivery and manageability of an application by providing pre-built services and hosting capabilities.

As an application developer, Bluemix enables you to focus on developing your application without concerning yourself with the infrastructure required to host it. There is a wide variety of runtime environments to enable several different application types, including Java, JavaScript, GO, PHP, Python, Ruby, and others.

In addition to these runtime environments, you can use pre-built services to rapidly compose applications from a marketplace of IBM and third-party services. The catalog of services on Bluemix includes several different categories, with specific services for each category. There are services for Big Data, Integration, IBM Watson™, Analytics, Data Management, and so on. The catalog includes over 100 services, and is still continuing to grow.

Figure 4-1 shows the Bluemix service categories.

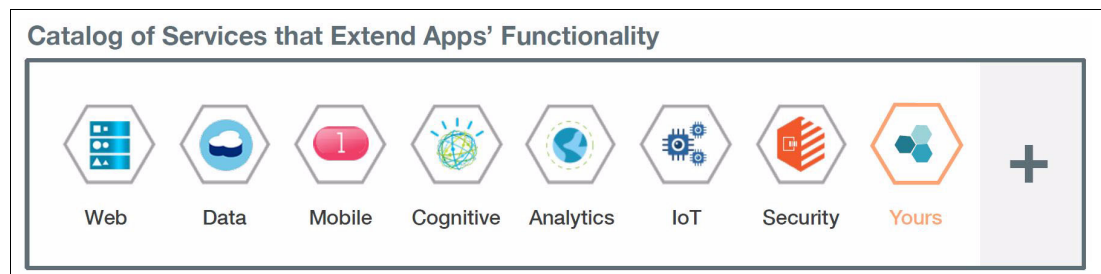


Figure 4-1 Bluemix service categories

Bluemix encourages *polyglot practices*, and enables you to develop different services using different languages and run times.

Polyglot programming: A polyglot is a computer program written in multiple programming languages, which performs the same operations or output independent of the programming language used to compile or interpret it.

You can also employ polyglot persistence to store data in different data storage technologies, depending upon the varying needs of your applications. Through composition of these consumable services and run times, Bluemix enables rapid development of your application.

Bluemix also provides tools for testing and rapid deployment of your application. When deployed, there are more tools for monitoring, logging, and analytics, in addition to the ability to easily scale up or down based on changes in the usage or load of your application. Bluemix provides a platform for application development, deployment, DevOps, and monitoring. We describe specifics of the Bluemix experience in the following section.

4.1.1 Bluemix deployment models

Bluemix enables you to choose a deployment model to use based on the following factors:

- ▶ Location or privacy requirements of your application and data
- ▶ Shared or private environment
- ▶ Level of infrastructure abstraction needed

See Table 4-1 for the list of available deployment models.

Table 4-1 Deployment options

Type	Available Options
Access model	<ul style="list-style-type: none">▶ Bluemix Public (Regions: US South and United Kindgom, with more public regions coming soon)▶ Bluemix Dedicated▶ Bluemix Local (coming soon)
Compute infrastructure	<ul style="list-style-type: none">▶ Virtual Images▶ Cloud Foundry Applications▶ Docker Containers

Deployment options based on access models

Bluemix offers several deployment options based on access models.

Public

Bluemix is offered as a PaaS environment leveraging SoftLayer® virtual servers to anyone over the Internet. This public availability means that to become quickly productive with Bluemix, a developer only needs to make an account and start developing and deploying applications.

Figure 4-2 on page 60 shows the basic flow of how Bluemix manages the deployment of applications for public platforms. When an application is developed and deployed, the execution environment for that application is isolated from the execution environment of other applications, even though that application might be running in the same data center as other applications.

Figure 4-2 shows a basic Bluemix deployment.

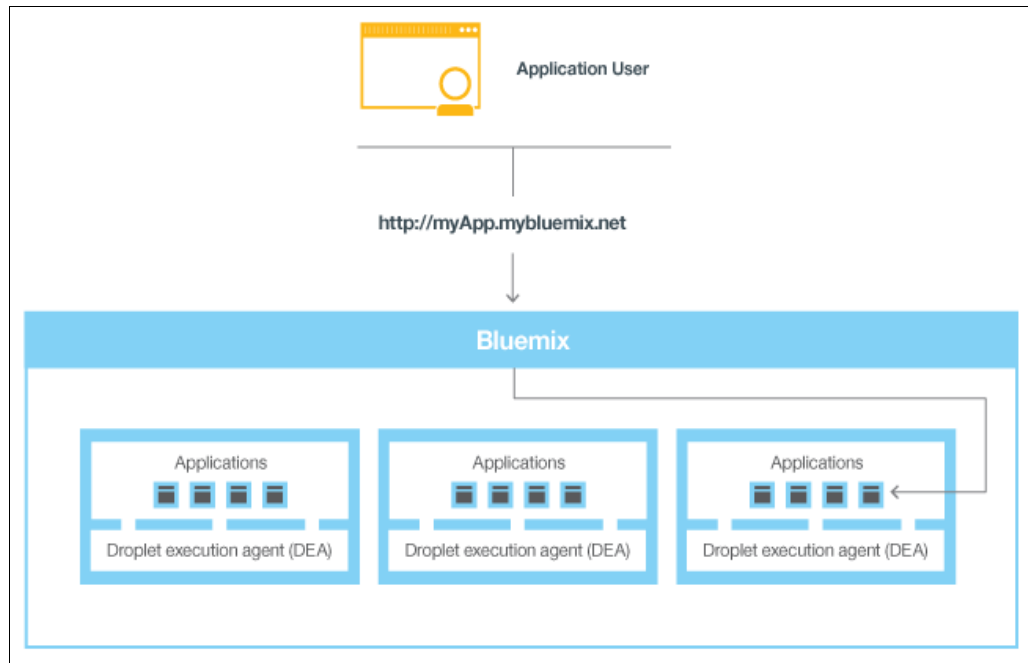


Figure 4-2 Applications on Bluemix

Dedicated

Bluemix Dedicated is your own exclusive SoftLayer environment that's securely connected to both the public Bluemix and your own network. Bluemix Dedicated is connected to your network through a virtual private network (VPN) or a direct network connection. Your single-tenant hardware can be set up in any SoftLayer data center around the world.

IBM manages the dedicated platform and dedicated services, so that you can focus on building custom applications. In addition, IBM performs all maintenance to dedicated instances during a maintenance window selected by you. IBM has several services that are available in your dedicated environment, but you can connect to public services as well. All run times are available in the dedicated environment.

All dedicated deployments of Bluemix include the following benefits and features:

- ▶ Virtual private network
- ▶ Private virtual local area network (VLAN)
- ▶ Firewall
- ▶ Connectivity with your Lightweight Directory Access Protocol (LDAP)
- ▶ Ability to effectively use existing on-premises databases and applications
- ▶ Always available onsite security
- ▶ Dedicated hardware
- ▶ Standard support

Local

Bluemix Local is geared more toward enterprises that want to have their own environments, and brings all the advantages of a PaaS into their private world and own data center. It is delivered as a fully managed service in your own data center, and brings cloud agility and Bluemix features to even the most sensitive workloads. Bluemix Local sits either on OpenStack or VMWare driven infrastructure, or on a Bluemix appliance. It also features a syndicated catalog from the public Bluemix offering so that you can easily locate apps and services based on their specific business or technical requirements.

Regions

A Bluemix region is a defined geographical territory that you can deploy your apps to. You can create apps and service instances in different regions with the same Bluemix infrastructure for application management, and the same usage details view for billing. You can select the region that is nearest to your customers, and deploy your apps to this region to get low application latency. You can also select the region where you want to keep the application data to address security issues.

When you build apps in multiple regions, if one region goes down, the apps that are in the other regions continue to run. Your resource allowance is the same for each region that you use. You can switch between different regions to work with the spaces in that region.

Figure 4-3 illustrates all three deployment models of Bluemix.

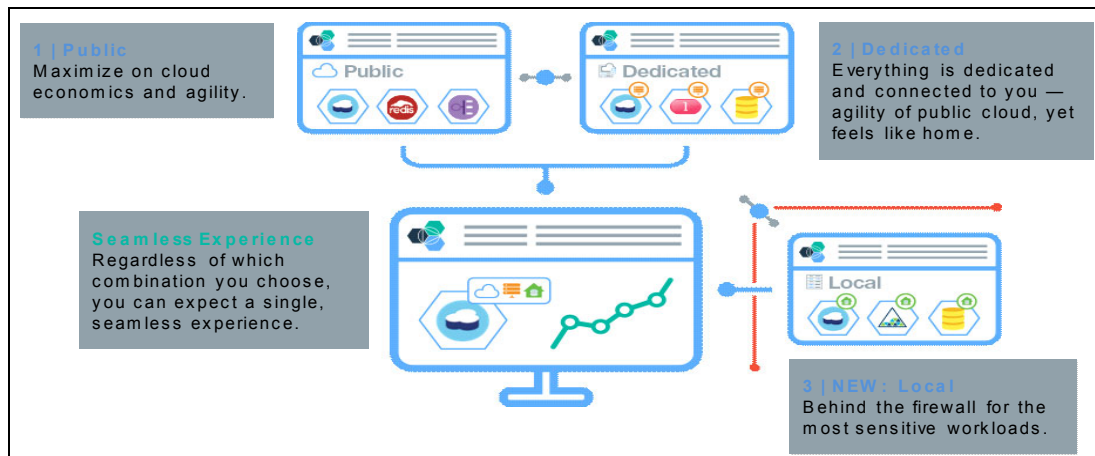


Figure 4-3 Bluemix deployment models

Deployment options based on compute infrastructure

Bluemix enables portability for your workloads by offering three infrastructure options to run your code.

Cloud Foundry

IBM Bluemix is based on Cloud Foundry, which is an open PaaS. In this PaaS model, you can focus exclusively on writing your code. You can leave aspects of running your application code to Bluemix, because all infrastructure, including the application servers, are provided by Bluemix. Bluemix takes care of the management and maintenance of all of the infrastructure and run time that powers your application.

This enables developers to build applications rapidly in local integrated development environments (IDEs) and push the apps to the cloud. This ease of use, however, comes at the expense of having less control over the infrastructure, which is desirable in certain scenarios.

Containers

With this option, you run your application code in a container that is based on the popular Docker containers. Different from the previous Cloud Foundry model, you not only must provide the applications, but also the application servers or run times, which are both included in the containers.

This comes with the benefit that you can port your applications more easily because you can run these containers locally, on-premises environments, and on cloud. The main disadvantage is that there can be some resource use involved with building and managing these containers. However, there are many images available that are provided by both IBM Bluemix (images for Liberty and Node.js) and public Docker Hub.

Note: With this option, you run pure Docker containers in a hosted environment on Bluemix. These are the same containers that run on-premises in a simple Docker environment, or in any other cloud-provider offering Docker services.

Virtual machines

In this option, you run your app code on your own VM with its own operating system (OS). Therefore, you are not only responsible for providing the application servers and run times, but also the operating system, so you have even more resources to use with building and management of the images relative to the previous options. There are public virtual images available, and the Bluemix virtual machines infrastructure does give you the ability to create and manage virtual machine groups on the IBM public cloud that makes management easier.

You can also create and manage VM groups on your private IBM clouds that you've chosen to make available to Bluemix users, in addition to the ability to connect to your on-premises infrastructure through guided experience. You can deploy and manage your VMs by using either the Bluemix user interface (UI), or the cloud's OpenStack application programming interfaces (APIs).

Choosing a deployment model

Microservices workloads should be co-located with their data to reduce latency between the business logic in your microservice and the data persistence store. You should consider deploying Bluemix local, if you have sensitive data that must remain onsite behind a corporate firewall.

Microservices that must comply with personal data regulations that require that personal data remain within a country can use Bluemix Local or Bluemix Public with choosing the region where the data will reside.

You can build a hybrid environment using Bluemix Public or Dedicated, and securely communicate between them using the IBM Secure Gateway service. The service provides secure connectivity from Bluemix to other applications and data sources running on-premises or in other clouds. A remote client is provided to enable secure connectivity. For example, you can use Secure Gateway to connect Bluemix to an on-premises database, as shown in Figure 4-4.

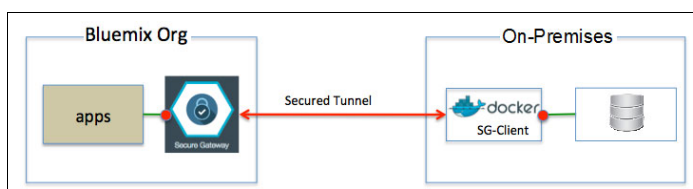


Figure 4-4 Secure Gateway connection

IBM Cloud Integration is another service that aids in building hybrid environments by exposing back-end systems of record as Representational State Transfer (REST) APIs. This enables you to provide a standard way of exposing your systems of record data in a standard, easy-to-use format. You can selectively show only those schemas and tables that you want. See the diagram in Figure 4-5 for a general flow.

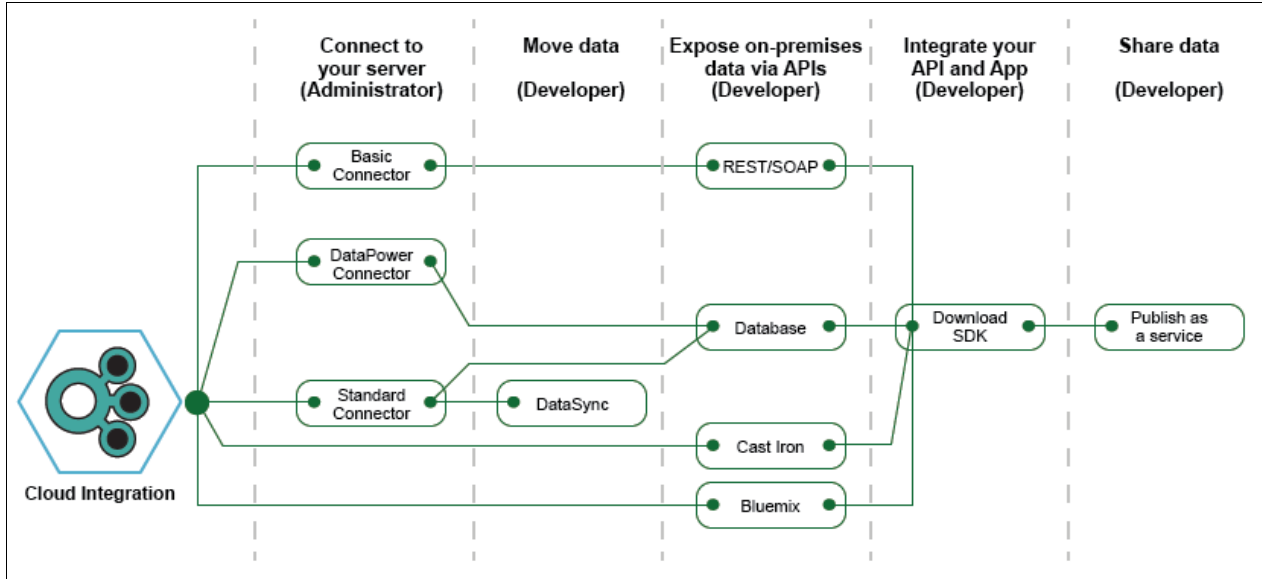


Figure 4-5 Cloud integration flow

4.1.2 Bluemix Cloud Foundry concepts

Bluemix consists of applications, services, buildpacks, and other components. You can deploy applications to different Bluemix regions by using one IBM ID.

Applications

In Bluemix, an application, or app, represents the artifact that a developer is building.

Mobile apps

Mobile client apps can use Mobile services that are provided in the Bluemix Catalog, or that are already deployed to and running in Bluemix. These services typically act in concert, and represent the back-end projection of that application. Bluemix can then run and manage these microservices.

Web apps

Web apps consist of all the code that is required to be run or referenced at run time. Web apps are uploaded to Bluemix to host the application. For languages such as Java, where the source code is compiled into runtime binary files, only the binary files are required to be uploaded. When building microservices, each microservice can be deployed as a Bluemix web application, and can communicate with each other using Hypertext Transfer Protocol (HTTP) or one of the provided services.

Services

In the Bluemix and Cloud Foundry terminology, a *service* is essentially any resource that can help a Bluemix user (developer or operator) develop and run their application. Bluemix services provide functionality that is ready-for-use by the app's running code. Example services provided by Bluemix include database, monitoring, caching messaging, push notifications for mobile apps, and elastic caching for web apps.

Bluemix stands out from the rest of the competition by providing a massive catalog of these pluggable services, ready to be immediately provisioned and used by your application. The Bluemix console displays all the available services in one single catalog view, as shown in Figure 4-6.



Figure 4-6 Bluemix service catalog

Some of the services have a different flow, but for most of them, you simply select a service and click **Create**. Within moments, Bluemix provisions your own instance of the service and provides the credentials for any application that you bind the service to.

Many powerful IBM products like IBM DB2 and Elastic Caching are offered as a service for quick consumption, with no setup. The catalog is populated with not only IBM products, but also third-party and community services like MongoDB and Twillio.

Many options in the catalog: This type of catalog creates a flexible environment that developers love. For example, at the time of this writing there are over 100 services and 12 different types of data management services to choose from.

You can create your own services in Bluemix. These services can vary in complexity. They can be simple utilities, such as the functions you might see in a runtime library. Alternatively, they can be complex business logic that you might see in a business process modeling service or a database.

Bluemix simplifies the use of services by provisioning new instances of the service, and binding those service instances to your application. The management of the service is handled automatically by Bluemix. For all available services in Bluemix, see the catalog in the Bluemix user interface:

https://console.ng.bluemix.net/?ace_base=true/#/store

Read more about the using services in your applications or creating services in the CloudFoundry documentation. Most of these capabilities are also available in Bluemix:

<http://docs.cloudfoundry.org/devguide/services/>

If you want to offer your own service in the IBM Bluemix Catalog as a third-party service, consider joining the IBM marketplace:

<http://www.ibm.com/cloud-computing/us/en/partner-landing.html>

This site shows you how to work with IBM to create and start the Service Broker API calls to enable you to have your service listed in the IBM Bluemix Catalog.

Starters

A starter is a template that includes predefined services and application code that is configured with a particular buildpack. There are two types of starters: Boilerplates and run times. A starter might be application code that is written in a specific programming language, or a combination of application code and a set of services.

Boilerplates

A boilerplate is a container for an application, its associated runtime environment, and predefined services for a particular domain. You can use a boilerplate to quickly get up and running. For example, you can select the Mobile Cloud boilerplate to host mobile and web applications, and accelerate development time of server-side scripts by using the mobile app template and software development kit (SDK).

Run times

A run time is the set of resources that is used to run an application. Bluemix provides runtime environments as containers for different types of applications. The runtime environments are integrated as buildpacks into Bluemix, and are automatically configured for use. Bluemix supports polyglot programming and enables you to use different programming languages, frameworks, services, and databases for developing applications.

You can use Bluemix to quickly develop applications in the most popular programming languages, such as Java, JavaScript, Go, and PHP. Figure 4-7 shows some of the current Bluemix run times.

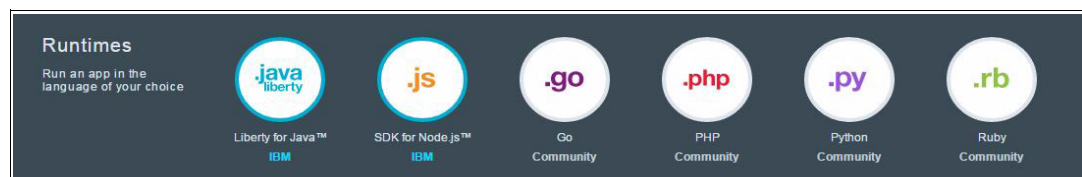


Figure 4-7 Bluemix run times

4.1.3 How Bluemix works

Bluemix (<https://bluemix.net>) is where you run and monitor your application. When building Bluemix applications, look at all of the services that are available in the catalog, and use them to rapidly build and deploy your application. For example, if your application has a workflow structure to it in terms of automating a certain business process, you can explore leveraging the workflow service.

Depending upon the need, you can further add more services, such as email services (for example, SendGrid service) or decision-making services (for example, IBM Watson Question and Answer service) from the Bluemix catalog. If you need a database back-end as a service, you can choose from various relational options (such as DB2/SQL, MySQL, and Postgres) and noSQL options (such as Cloudant and MongoDB). In this way, you can rapidly create Bluemix applications by leveraging the services available in the Bluemix catalog.

When you deploy an application to Bluemix, you must configure Bluemix with enough information to support the application:

- ▶ For a mobile app, Bluemix contains an artifact that represents the mobile app's back end, such as the services that are used by the mobile app to communicate with a server.
- ▶ For a web app, you must ensure that information about the proper run time and framework is communicated to Bluemix, so that it can set up the proper execution environment to run the application.

Each execution environment, including both mobile and web, is isolated from the execution environment of other applications. The execution environments are isolated even though these apps are on the same physical machine. When you create an application and deploy it to Bluemix, the Bluemix environment determines an appropriate virtual machine (VM) to which the application or artifacts that the application represents is sent. For a mobile application, a mobile back-end projection is created on Bluemix.

Any code for the mobile app running in the cloud eventually runs in the Bluemix environment. For a web app, the code running in the cloud is the application that the developer deploys to Bluemix. In each VM, an application manager communicates with the rest of the Bluemix infrastructure, and manages the applications that are deployed to this VM. Each VM has containers to separate and protect applications. In each container, Bluemix installs the appropriate framework and run time that are required for each application.

When the application is deployed, if it has a web interface (such as a Java web app), or other REST-based services (such as mobile services exposed publicly to the mobile app), users of the app can communicate with it by using normal HTTP requests. Each app can have one or more Uniform Resource Locators (URLs) associated with it, but all of them must point to the Bluemix endpoint. When a request comes in, Bluemix examines the request, determines which app it is intended for, and then selects one of the app instances to receive the request.

Bluemix resiliency

Bluemix is designed to host scalable, resilient applications and application artifacts that can both scale to meet your needs, and be highly available. Bluemix separates those components that track the state of interactions (*stateful*) from those that do not (*stateless*). This separation enables Bluemix to move applications flexibly as needed to achieve scalability and resiliency.

You can have one or more instances running for your application. When you have multiple instances for one application, the application is uploaded only once. However, Bluemix deploys the number of instances of the application requested, and distributes them across as many VMs as possible.

You must save all persistent data in a stateful data store that is outside of your application, such as on one of the data store services that are provided by Bluemix. Because anything cached in memory or on disk might not be available even after a restart, you can use the memory space or file system of a single Bluemix instance as a brief, single-transaction cache. With a single-instance setup, the request to your application might be interrupted because of the stateless nature of Bluemix.

A leading practice is to use at least three instances for each application to ensure the availability of your application, where one or more of those instances is also in a different Bluemix region. All Bluemix infrastructure, Cloud Foundry components, and IBM-specific management components are highly available. Multiple instances of the infrastructure are used to balance the load.

Bluemix security

All the same security concerns that apply to applications also apply to microservices. These security risks can also be addressed in similar ways with microservice levels. Bluemix is designed and built with secure engineering practices at the physical layer, software layer, and communication layer. The Bluemix documentation contains more information about this topic:

https://www.ng.bluemix.net/docs/overview/sec_exec_ov.html

Liberty for Java Run Time in Bluemix

Support for Java applications in Bluemix is provided by the Liberty for Java Run Time. Supporting both web archive (WAR) and enterprise archive (EAR) files, the lightweight WebSphere Application Server Liberty Profile and a special small footprint IBM Java runtime environment (JRE) are used to keep disk and memory requirements low. This, combined with fast start-up times and market-leading throughput, makes Liberty an ideal run time for Java-based microservices.

The associated buildpack in Bluemix handles generating the required server configuration based on the services that the application is bound to, enabling the microservice developer to use standard Java Platform, Enterprise Edition (Java EE) APIs for injecting dependencies. By default, this configuration will include all of the features required to support the Java EE 6 Web Profile.

If only a subset is needed (for example, a microservice might only require Java API for RESTful Web Services (JAX-RS) and Java Database Connectivity (JDBC) support) further runtime memory savings can be made by overriding this default set of features with the **JBP_CONFIG_LIBERTY** environment variable.

Many Java EE 7 features are also available, either as fully supported or beta features, the latter enabled using the **IBM_LIBERTY_BETA** environment variable. One of the new features that is likely to be of particular interest to microservice developers is JAX-RS 2.0. This adds a client API for starting REST services, which includes support for handling responses asynchronously. This support is useful for keeping latency low when starting multiple downstream services.

Asynchronous support on the server side also makes it much easier to implement services that support long polling, although Liberty also supports the use of WebSockets, which might be more appropriate. There is both inbound and outbound Java Message Service (JMS) support for those looking to produce a messaging-based solution.

Liberty brings with it a wealth of other connectivity options, including to SQL and NoSQL databases like MongoDB and CouchDB. Although perhaps not what you would want to expose from your microservice, Java API for XML Web Services (JAX-WS) support might be of use in bridging to existing service implementations.

Runtime fidelity between the Liberty Profile and WebSphere Application Server Full Profile, combined with Eclipse-based migration tools, make it easier to port existing monoliths to WebSphere Liberty before decomposing them into microservices to run in Bluemix.

The Liberty run time and the WebSphere Developer Tools, which work with the Bluemix Eclipse tools, make it easy to develop services locally and then deploy them to Bluemix. The Liberty run time is available to download at no initial charge from the following website:

<https://developer.ibm.com/wasdev/>

Finally, Liberty's zero-migration policy means that you should never be forced to make code changes when Bluemix moves to newer versions of the run time.

4.2 Developing microservices using Bluemix DevOps

When first implementing a new business application that uses microservices in Bluemix, it might make sense to identify a low-risk service that isn't as broadly used across multiple applications. This enables you to do a proof of concept and learn about microservices and Bluemix. These services can be created as individual microservices in Bluemix. When building microservices on Bluemix, treat each microservice as an individual application on Bluemix. Each application has its own lifecycle, and is managed independently.

Each application typically should only have a small set of responsibilities, and this principle usually results in a large set of apps. Usually this situation would have presented several operational and deployment challenges, and this is where Bluemix DevOps comes to the rescue. DevOps is an important set of practices and tools to improve efficiencies in software delivery. This section explains how you can begin to develop, build, and test a microservices application using Bluemix DevOps Services and run a microservices application on Bluemix.

Tip: In situations where you are using Bluemix Local, and you also want to manage deployment pipelines for applications within their organizations, you should consider using IBM UrbanCode Deploy.

4.2.1 Bluemix DevOps introduction

IBM Bluemix DevOps Services (<https://hub.jazz.net/>) is where you develop, build, test, and deploy your application code from, run your ALM cycle, and support continuous delivery. Figure 4-8 captures a typical ALM cycle of a microservice application for a continuous delivery model.

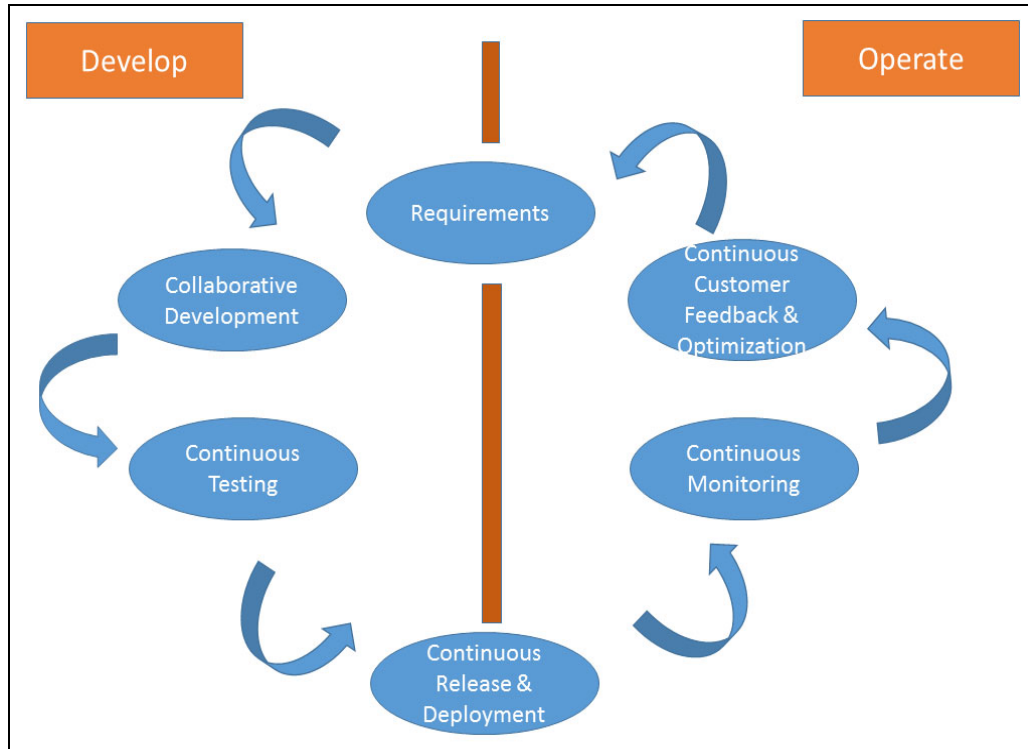


Figure 4-8 Application lifecycle management of typical microservices in Bluemix

The Bluemix DevOps provides a set of services that enables you to adopt DevOps for this ALM cycle, and more specifically an app delivery pipeline as a set of services. With IBM Bluemix DevOps Services, you can develop, track, plan, and deploy software in one place. From your projects, you can access everything that you need to build all types of apps. You can use the services without any concern about how the services are hosted and delivered.

The following list includes some typical Bluemix DevOps Services spanning the ALM cycle:

- ▶ Agile planning, through the Track & Plan service
- ▶ A web IDE for editing and managing source control on a web browser or Eclipse Plug-in for Bluemix (or integrate with existing Eclipse and work locally)
- ▶ Source control management (SCM), through Git, IBM Rational Jazz SCM, or GitHub
- ▶ Automated builds and deployments, through the Delivery Pipeline service
- ▶ Integrated and Load Testing cycles through Testing services
- ▶ Visibility, control, and analytics through Monitoring and Analytics services
- ▶ Automatically increase or decrease the compute capacity through Auto-Scaling services

Getting started with Bluemix DevOps

As a first step, you need to create a Bluemix account on <https://console.ng.bluemix.net/>. After registering for a Bluemix account, you are automatically assigned an organization. In your organization on the Bluemix dashboard, you can invite other members to collaborate with you on your application in your space. Select the **Manage Organization** option in the upper left corner to manage access controls to your spaces, and invite other team members to your space.

Bluemix DevOps Services and Track and Plan are separately charged, development-time services that you must add to your Bluemix account to use. To add the Bluemix DevOps Services to your Bluemix account, go to the Bluemix catalog and add it to your space. You must separately create and invite team members to a Bluemix DevOps Services project.

We suggest creating a separate Bluemix application and Bluemix DevOps Services project for each microservice. See the following options for information about how to get started. There are different flows for getting started, depending on if you are starting with an existing application and you already have code, or are creating a new application. Start by creating your environments on Bluemix.

Create Bluemix spaces

Create a Bluemix space for each microservice application environment. Invite team members to the space. Set access controls for each microservice application environment space. See the instructions in the documentation:

<https://www.ng.bluemix.net/docs/#acctgmt/index.html#acctgmt>

Table 4-2 represents one possible way to set up access controls for spaces. It provides one or more development spaces for each developer. A shared integration test environment enables you to test multiple microservices applications and their dependencies. Additional spaces may be required for other regions or deployment models.

Table 4-2 Deployment environments and team role permissions

Space	Developer			Tech Lead			Manager		
	Mgr	Dev	Aud	Mgr	Dev	Aud	Mgr	Dev	Aud
Development-CarlosF	■	■	■	■	■	■	■	□	■
Integration-Test	□	■	□	□	■	□	■	□	■
Production-Blue-US	□	□	□	□	■	□	■	□	■
Production-Green-US	□	□	□	□	■	□	■	□	■
Production-Blue-UK	□	□	□	□	■	□	■	□	■
Production-Green-UK	□	□	□	□	■	□	■	□	■

Create a Bluemix Cloud Foundry application run time

As described in “Deployment options based on compute infrastructure” on page 61, there are several ways to create your Bluemix compute environments to run your microservices applications on Bluemix:

- ▶ Cloud Foundry Application
- ▶ Containers
- ▶ Virtual Images

If you are starting a new microservice application and do not yet have code, the easiest way to get started is to create your application run time using one of the following approaches. After that, decide how you want to manage and deploy your application code. The paragraphs that follow describe the ways that you can create application run times without coding. Figure 4-9 shows how to create an application run time from the Bluemix catalog.

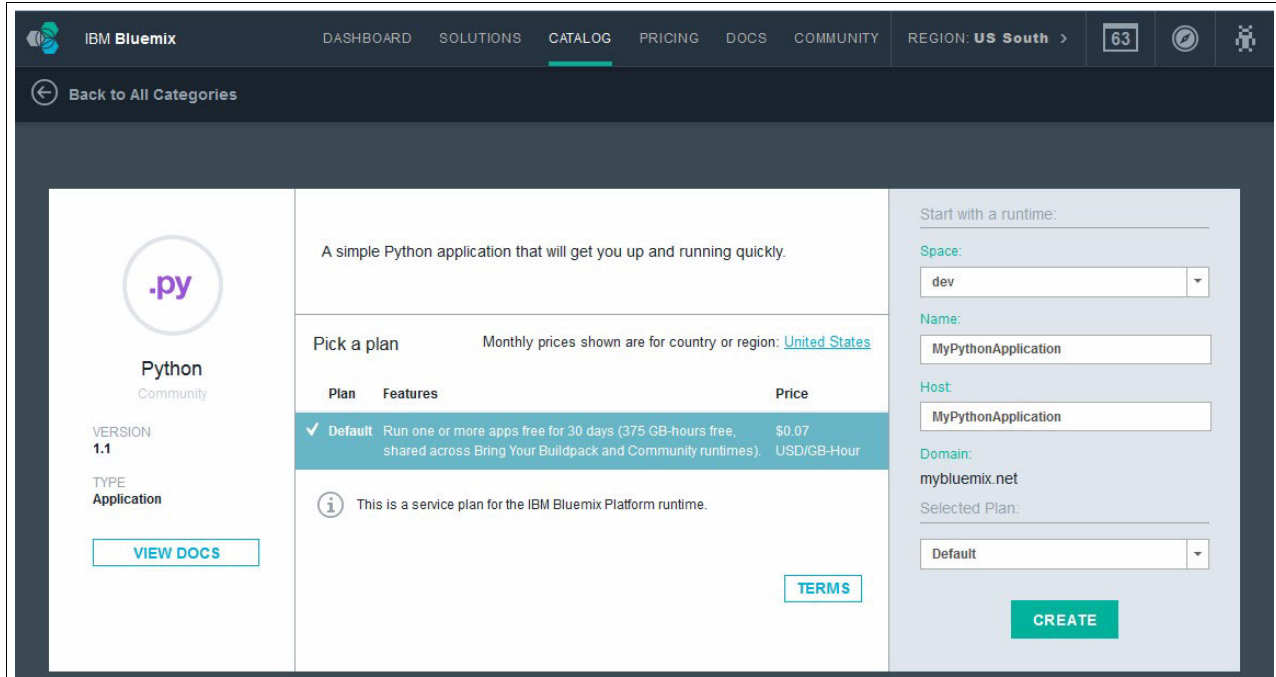


Figure 4-9 Create an application run time from the Bluemix catalog

Create an application run time from the Bluemix catalog

IBM and their IBM Business Partners have created templates, called *boilerplates*, for some of the most common activities to help you get started creating applications. They include an application run time and a bound set of services. To create an application run time from the Bluemix catalog, perform the following steps:

1. From the **Bluemix catalog**, select one of the available application boilerplates that have predefined bounded services or choose a run time.
2. Name your application and chose a space to deploy the application to.
3. Click **Start Coding** submenu of your application to learn more about how to manage code and do deployments to Bluemix.
4. Decide how you want to manage code and do deployments. See the sections that follow for an explanation of the available options.

5. Click the **Overview** menu on the upper left to return to your application overview page. See Figure 4-10.

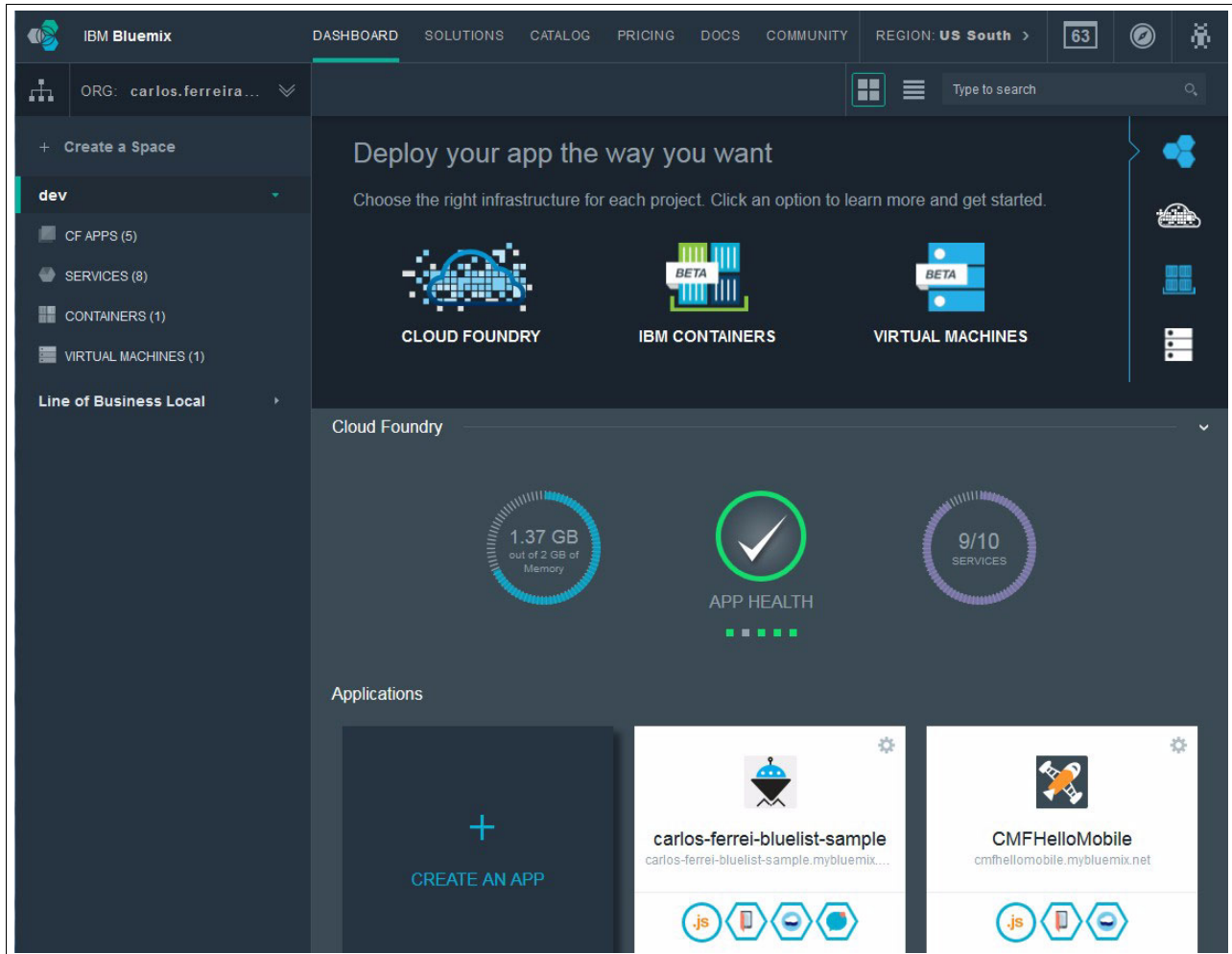


Figure 4-10 Create microservice application in Bluemix dashboard

Create an application run time from the Bluemix dashboard for Cloud Foundry applications

Perform the following steps to create an application run time from the Bluemix dashboard for Cloud Foundry applications:

1. From the Bluemix dashboard. Click the **CREATE AN APP** tile.
2. Select the **web** option and choose the run time that you want to run your microservices on. Choose **Community buildpacks** if you don't see the programming language run time that you want. New run times are being added.
3. Click the **Start Coding** submenu of your application to learn more about how to manage code and do deployments to Bluemix.
4. Decide on how you want to manage code and do deployments. See the sections that follow for an explanation of the available options.
5. Click the **Overview** menu on the upper left menu to return to your application overview page.

Setting up local Eclipse clients to work with Jazz source control

If you want to use Jazz source control management (SCM) for your Bluemix DevOps Services project, you can either do the coding locally and deliver changes through Eclipse, or work by using the integrated web IDE. If you want to do coding locally and deliver changes through Eclipse, you can install the IBM Rational Team Concert plug-in for version control. The following steps are some of the main steps involved:

1. If you do not have Eclipse installed already, download and install “Eclipse IDE for Java EE Developers” from the Eclipse website:
<http://www.eclipse.org/downloads/>
2. Install IBM Rational Team Concert Version 5.0.2 or later. See steps in the following link for further instructions:
https://hub.jazz.net/docs/reference/jazz_scm_client/
3. From Eclipse, you can connect to your projects in the following ways:
 - a. Use the Manage IBM DevOps Services Projects tool.
 - b. Accept a team invitation. See the previous link for more detailed instructions.
4. Load Bluemix DevOps Services projects in Eclipse. For this, the project must have the Track & Plan feature enabled. See the previous link for more detailed instructions.
5. Deliver your changes in your local repository to Bluemix. Again, See the previous link for more detailed instructions.

After the general structure is set up, it is possible to edit the code in the web UI and in the Eclipse client. When you deliver the code to the stream, it gets automatically built and deployed.

Create a Cloud Foundry application using the Deploy to Bluemix button

There might be existing Bluemix applications or starter templates that exist within your organization, or on the Internet using Git repositories. Git provides distributed SCM services, such as versioning. GitHub is a popular Git service provider. One way people share their applications is through GitHub, Bluemix DevOps Services Git project, and other Git repositories using the **Deploy to Bluemix** button.

To make those applications easier to fork and deploy, IBM Bluemix has provided a service that connects to a Git URL that you provide, and automatically deploys the application to one of your selected Bluemix Spaces and Regions. The Deploy to Bluemix button service fetches the code and creates a Git project and pipeline in Bluemix DevOps. After you provide or confirm the application name, the application and services described in the manifest file are automatically deployed to Bluemix.

This service requires that you have a Bluemix and Bluemix DevOps account. Example 4-1 demonstrates how you can also add your Deploy to Bluemix button to your own microservices starter templates to make it easier for your teams to use similar approaches for implementing microservices. Here is a link to the documentation about how you can add the **Deploy To Bluemix** button to your samples in Git repositories:

<https://www.ng.bluemix.net/docs/#manageapps/index-gentopic2.html#appdeploy>

Example 4-1 How to add the Deploy to Bluemix button to your own samples README.md

```
[![Deploy to  
Bluemix](https://bluemix.net/deploy/button.png)](https://github.com/fe01134/2048.g  
it)
```

Figure 4-11 shows you how to deploy the application to Bluemix.

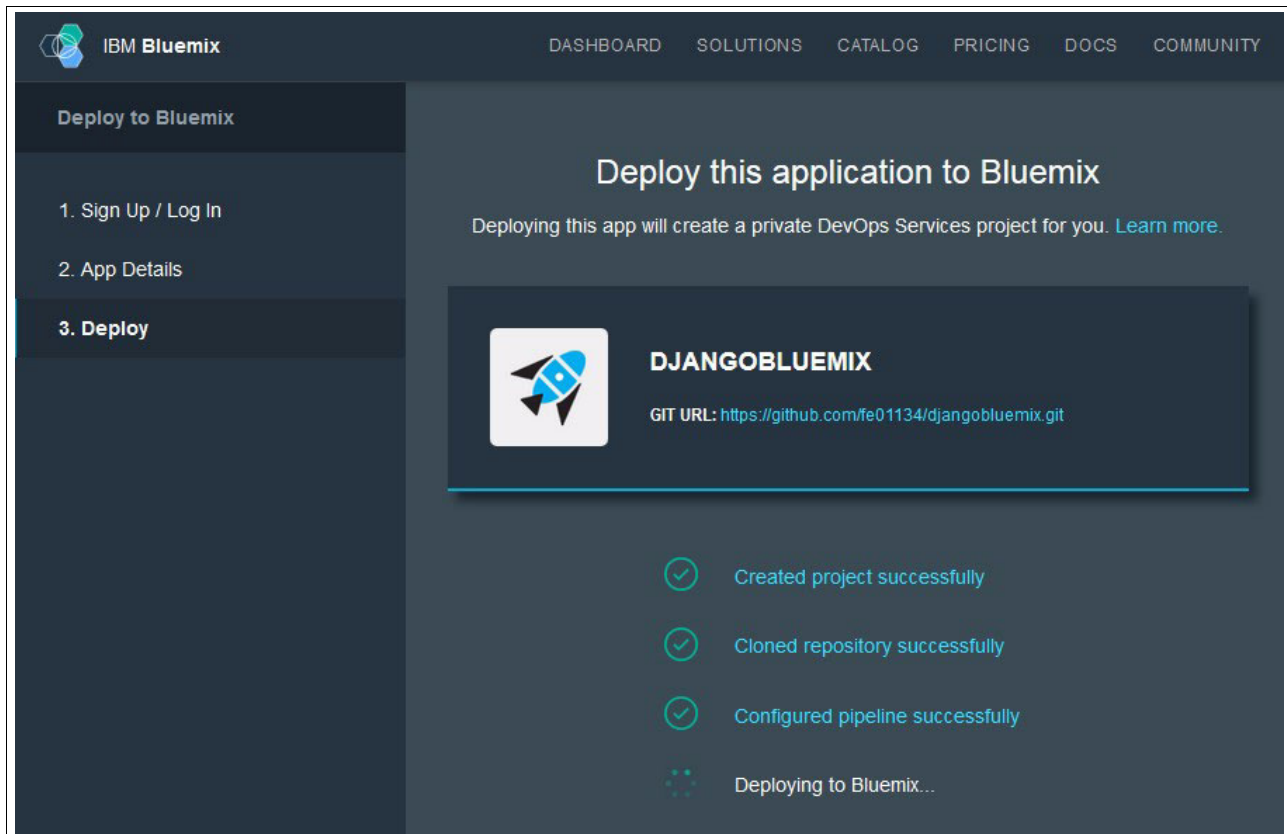


Figure 4-11 Deploying application to Bluemix

Create a microservice container or virtual image application project

If your microservice application already exists as a Docker Container or VM, you can also create an application and deploy it on Bluemix. At the time of this publication, IBM Containers and IBM virtual machines were in Beta. You can create a container app or virtual machine app using a similar approach as described earlier with Cloud Foundry applications.

Because both services are in Beta and likely to change, we suggest that you read the documentation for how to create containers. Read the section about *Creating and deploying a container or container group*:

<https://www.ng.ibm.com/docs/starters/index-gentopic3.html>

For creating Virtual Images, we suggest that you read the section about *Creating a virtual machine* in the documentation:

<https://www.ng.ibm.com/docs/starters/index-gentopic4.html>

4.2.2 Delivery pipeline

Delivery pipeline enables you to create stages and their corresponding jobs for each phase of the deployment of your microservices. It is common to have delivery pipeline delivering to different Bluemix environments, such as development, integration test, system test, and green-blue production and multiple regions or deployment models (public, dedicated, and local). Delivery pipeline is also helpful for delivering different compute-type microservices.

IBM DevOps Services supports deployment of Cloud Foundry Apps and containers:

- Stages** Manages the flow of work through the pipeline. Stages have inputs, job, and properties.
- Jobs** Is where work is done. One or more within a stage. Build, deploy, and test are the categories of jobs. An example job could be a script to start integration testing services described in 4.2, “Developing microservices using Bluemix DevOps” on page 68.
- Triggers** Events that start the execution of a stage. They can be manual or automatic. Automatic triggers include SCM commits and stage completion.
- Inputs** Defined on a stage, and they identify the artifacts used by the jobs within a stage. They can be an SCM branch or the artifacts produced by another job from a previous stage.
- Properties** Defined on a stage, and values are passed to Jobs within the stage A job can override a value and have it passed to a downstream job.

Figure 4-12 shows how Bluemix DevOps Services pipeline is configured to deploy to multiple Bluemix regions. Bluemix DevOps Services on public instances can deploy to your dedicated or local instances of Bluemix if there is accessible network connectivity. Pipeline also supports deployment of containers.

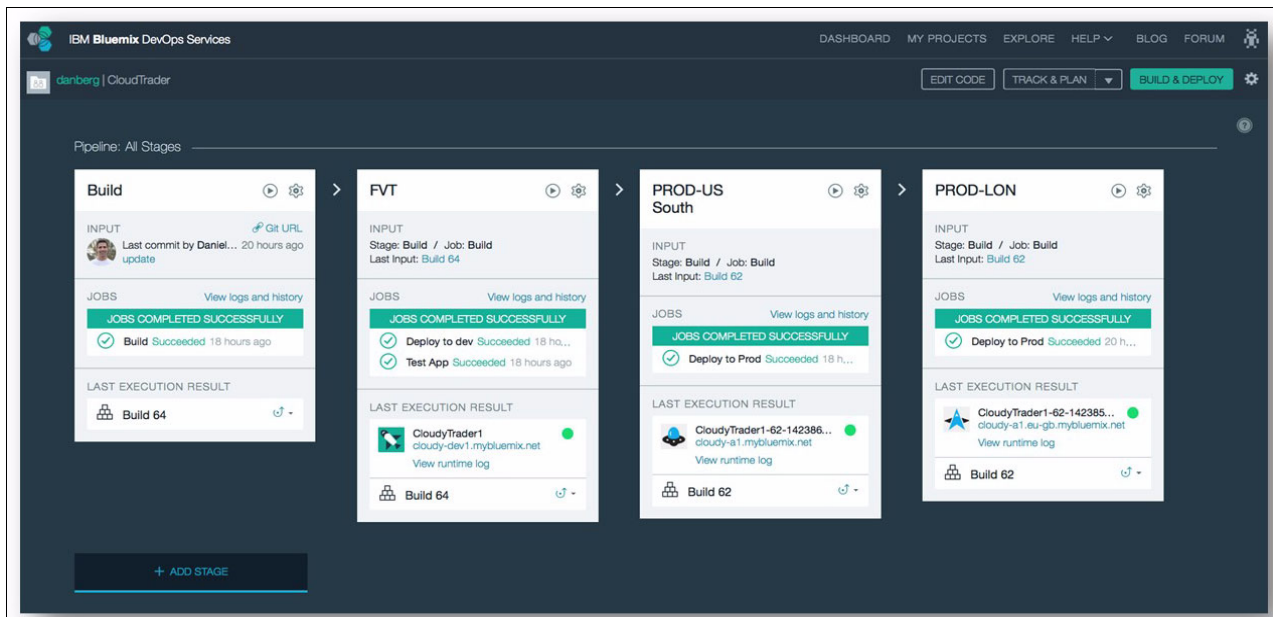


Figure 4-12 Example multi-region IBM Bluemix DevOps Services delivery pipeline

4.3 Deployment, testing, monitoring, and scaling services in Bluemix DevOps

In this section we describe deployment, testing, monitoring, and scaling services in Bluemix DevOps.

4.3.1 Deployment services in Bluemix

This section explains the different options that you must manage and deploy your code to Bluemix with or without Bluemix DevOps Services. You are not required to create a Bluemix DevOps project to use Bluemix. Bluemix DevOps Services were designed to work seamlessly with Bluemix, making it easy to get started and scale the frequency of deployments.

Decide how to control your source code and deploy it to Bluemix

After creating your application run time, you must decide where to manage your code and how to deploy it to Bluemix. Table 4-3 summarizes the available options.

Table 4-3 Available options to deploy to Bluemix

Code is managed on	Deploy to Bluemix using
Local client	Cloud Foundry command-line interface (CLI)
	Local Git to remote push
GitHub	Local Git to remote fetch from GitHub and push to Bluemix
	Bluemix DevOps Pipeline GitHub web hook integration
Bluemix DevOps Services SCM	Bluemix DevOps Pipeline

The Cloud Foundry command-line interface (CLI) enables you to administer your Bluemix applications from your desktop client. There are various functions it provides for creating spaces, applications, services, and users. It also provides functions for deploying and binding services. In addition, you can manage the application run times and their bound services by increasing the number of instances, allocating memory resources and other actions like completing deployments.

You can also start bash scripts to populate or migrate databases. You must separately download and install the Cloud Foundry CLI and Git. You can find instructions by clicking the **Start Coding** menu from your application overview page in Bluemix.

Code is on local client and use Cloud Foundry CLI to deploy

The two options for deploying code from your local client to Bluemix are using Cloud Foundry CLI or using Git, as described in “Code is on local client use Git to deploy” on page 77. Using Cloud Foundry CLI, you can manage your Bluemix application from your local client command window.

Tip: It is best to use the Cloud Foundry CLI option when you have an on-premises or third-party build pipeline that you depend on to build your application code.

If you have already created your application run time, as was described earlier, carefully read the directions on the **Start Coding** page of your application. Some run times, such as Node.js, might have additional options shown for managing and deploying your code to Bluemix.

Bluemix automatically detects what type of application code you have when you deploy your applications. You can also consider using a manifest file in situations where you prefer not to specify additional `cf push` options on the command line when deploying an application.

The manifest file describes how to deploy the application to Bluemix, create services, and bind those services to your application. See instructions about creating manifest files. You can also specify memory and the number of instances used to run your application run time. For more information about the manifest files, see the following website:

<http://docs.cloudfoundry.org/devguide/deploy-apps/manifest.html>

For more detailed instructions about how to deploy your application to Bluemix using Cloud Foundry CLI, follow the instructions on the following website:

https://www.ng.bluemix.net/docs/#starters/upload_app.html

After adding the required manifest file to your application, use the Cloud Foundry CLI `cf push` command to deploy your application from your local client directly to Bluemix. Open a separate Cloud Foundry CLI window to view the logs, and to troubleshoot failed deployments when your application fails to start. Use the Cloud Foundry CLI `help` command to learn other useful commands. You must initiate a `cf push` command each time you want to deploy new tested code to Bluemix.

Tip: The most common deployment errors are caused by an invalid `manifest.yml` file format. Read the documentation or reuse existing valid manifest files from other projects to reduce the chance of errors in your manifest file. Additionally, you can create <http://cfmanigen.mybluemix.net/> or validate your manifest file using online `manifest.yml` validators.

Code is on local client use Git to deploy

To use Git to deploy your application, you must first create your application run times and enable the Git deployment option. To enable Git deployment for each application, click the **Add Git** link on the upper right of the overview page of your Bluemix Application. This provides you a Git URL that you can use from your local client to deploy to Bluemix using Git. It also provides you an optional Bluemix DevOps Git repository project if you later decide that you prefer to use Bluemix DevOps Services to manage your code and deployments.

Use the corresponding Git commands to add, commit, and then push your code to the remote Bluemix Git URL. This results in your application being deployed to Bluemix.

Tip: If you are unsure of the folder structure of your Bluemix application, or where to put your manifest file look at working examples on Bluemix DevOps Services or on Github. You can also do an initial Git fetch from the Bluemix Git URL to see the required folder structure and location of the manifest file.

Use GitHub with Bluemix

GitHub is a popular web hosted Git service provider. There are two options you can use to deploy your code from GitHub to Bluemix. The first option is to use Git locally, remotely fetch your code from GitHub, and then do a remote push to Bluemix. This requires that you set up your Bluemix application with a Git URL, as described in “Code is on local client and use Cloud Foundry CLI to deploy” on page 76.

The second option is to configure the GitHub Web Hooks integration to automatically deploy your code to Bluemix whenever there is an update to your master branch on GitHub. This deployment option requires that you enable and use the Bluemix DevOps Services pipeline.

Make sure that you have created your Bluemix DevOps SCM project as a GitHub integrated project. See the explanation in the next section.

Use Bluemix DevOps Services SCM and pipeline to deploy

There are two ways to initiate creating a Bluemix DevOps Services project. If you have already created your Bluemix run time, you can select **Add Git** option on the upper right. This creates a Bluemix DevOps Services project, Jazz Git repository, and a delivery pipeline. You can then add code to your Jazz Git repository using the web IDE, or use the normal Git commands from your local client to add, commit, and then push your code to the remote DevOps Jazz Git repository. As you add code to your Jazz Git repository, it automatically triggers the default pipeline that builds and then deploys your code to Bluemix.

You can also go to the Bluemix DevOps Services, create an account, and then create a project. You are prompted to choose which services that you want to use, such as Jazz Git Repository or Jazz SCM. You also must decide whether to make the project public or private, and choose which run times to bind your delivery pipeline to. If you are using Jazz SCM, you can use the optional Rational Team Concert SCM Eclipse client plug-in to add your code to a Bluemix DevOps Services Jazz SCM repository.

4.3.2 Testing services in Bluemix DevOps

Testing is an important part of the ALM cycle, and it is advised to test software early, often, and continuously. Bluemix provides for Integration and Load Testing as part of the DevOps capability.

Integration testing in Bluemix

Integration testing focuses on the interaction between two or more *units*. Integration testing tests integration or interfaces between the units, and their interactions to different parts of the system. You can automate this testing in Bluemix with the Integration Testing service.

Even if you don't have coding experience, you can define and run automated tests for publicly reachable, HTTP-based, service endpoints. In a few clicks, you can use the shared dashboard to discover and run integration tests, and view confidence indicators for the quality of the services that are exposed and consumed within the system being tested. With this greater understanding of quality, you can determine where to focus ongoing testing and development efforts.

Load testing in Bluemix

Load testing helps determine the capability of handling a certain number of users (load) on your application. The goal is to see how your application performs when subjected to both an expected and stressful amount of load. Bluemix has several third-party load testing options.

BlazeMeter is a self-service, web, API mobile load-testing platform (PaaS) that provides developers an enterprise grade, ready-for-immediate-use load testing solution that's 100% compatible with Apache JMeter. Load Impact is another alternative Unlimited Load testing, on-demand from multiple geographic locations solution that enables you to create sophisticated tests using a simple GUI.

4.3.3 Monitoring and analytics services in Bluemix

Monitoring is an important component of the *Operate cycle* of ALM. IBM Bluemix provides monitoring and analytics services as part of its DevOps Services offerings. Monitoring is provided at multiple levels in Bluemix:

- ▶ System-level monitoring
- ▶ Application-level monitoring
- ▶ Service-level monitoring

System monitoring is available for the Bluemix Platform:

<https://status.ng.bluemix.net/>

Application-level monitoring is available for Cloud Foundry applications and containers. You can use IBM Monitoring and Analytics service to monitor your Cloud Foundry Application and access your Cloud Foundry application logs. To learn more, read the documentation *Getting started with IBM Monitoring and Analytics for Bluemix*:

<https://www.ng.bluemix.net/docs/services/monana/index.html>

Containers monitoring is provided for basic processor (CPU), memory, network, and disk metrics. By default, when no *Collectd agent* is installed in the containers, the containers host provides basic CPU, memory, network, and disk metrics. If you want to see application-level metrics from inside the container, you must have a machine-mode container that has a *Collectd agent* that is configured with the Bluemix multi-tenant output plug-in. Similarly, to view logs from files inside a machine-mode container, you must install the Bluemix multi-tenant logstash-forwarder agent.

Note: For more information about how to install these agents, see the following references:

- ▶ Agents for Ubuntu and Debian: <https://logmet.opvis.bluemix.net:5443/apt/>
- ▶ Agents for Centos and RHEL: <https://logmet.opvis.bluemix.net:5443/yum/>

Application level monitoring is also possible within containers. You must set up application-level metrics from inside the container. You must have a machine-mode container that has a *Collectd agent* that is configured with the Bluemix multi-tenant output plug-in. Containers use Elasticsearch and Graphite to store logs and metrics. It provides access to that data through the Bluemix Operational dashboard. Access to the data is controlled on a per-space basis.

Service level monitoring is available within various services. For example, if you use the Advanced Mobile Access Service for iOS, you can monitor the number of client connections to your mobile back end. It provides frequency and mobile device iOS versions that are connecting. It also provides monitoring of your Cloudant database. When assessing services also evaluate their monitoring capabilities.

4.3.4 Scaling in Bluemix

Bluemix enables users to scale your Cloud Foundry application, both horizontally and vertically. Based on your application need, you might need to add more memory to existing instances, or you might need to scale the number of instances.

Vertical scaling

Vertical scaling in Bluemix is the act of adding or removing resources for each instance of your application. For example, the memory limit or the disk limit per application instance can be adjusted. Figure 4-13 shows the scaling options in the Bluemix console.

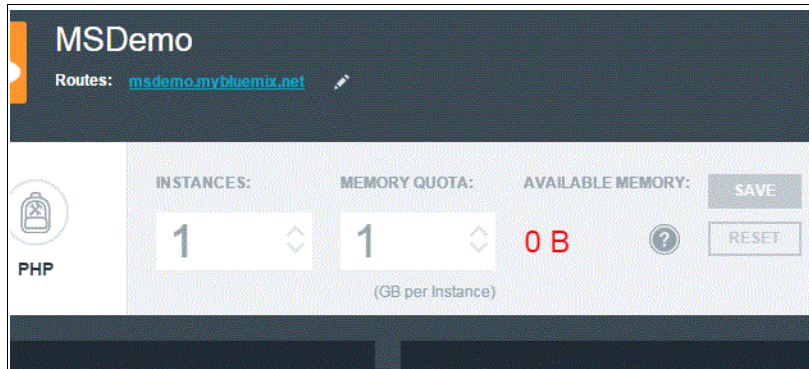


Figure 4-13 Scaling options in the Bluemix console

Horizontal scaling

You can have one or more instances running for your application. When you request multiple instances for one application, the application is only uploaded once. Bluemix deploys the number of instances of the application requested, and distributes them across as many VMs as possible. Each instance runs in its own isolated environment, and does not share memory space or file system. They are identical nodes, and a load balancer is built into Bluemix to route incoming traffic across all instances, as shown in Figure 4-14.

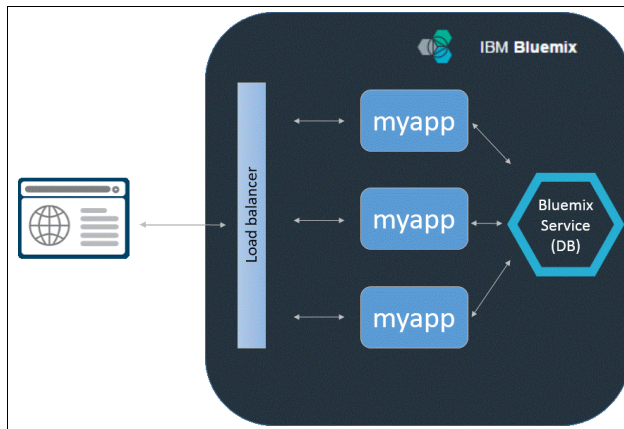


Figure 4-14 Load balance on Bluemix

With a single instance setup, the request to your application might be interrupted because of the stateless nature of Bluemix. A leading practice is to use at least three instances for each application to ensure the availability of your application.

When using horizontal scaling, it is crucial to confirm that your application is designed properly, and does not violate any cloud application rules. Review the following website for leading practices:

<http://www.12factor.net>

Auto-Scaling

The IBM Auto-Scaling for Bluemix is a DevOps service that enables you to automatically increase or decrease the application instances. The number of instances are adjusted dynamically based on the auto-scaling policy that you define. The policy enables you to use metrics, such as CPU, memory, and Java virtual machine (JVM) heap, to define your scaling policy. You can define multiple scaling rules for more than one metric type, as shown in Figure 4-15.

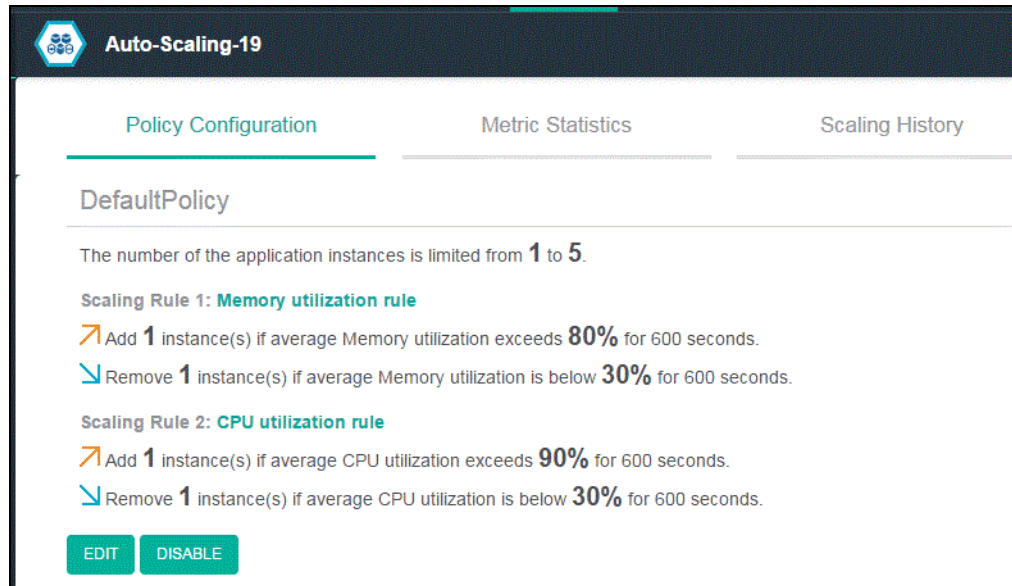


Figure 4-15 Auto-scaling policy on Bluemix

There are many things to consider before deciding to scale your application. The performance benefits of both horizontal and vertical scaling depend heavily on how your application is written, and there is no one policy that can be applied to all. See the following article for a more thorough explanation about, and implications of, scaling on Bluemix:

<http://www.ibm.com/developerworks/cloud/library/cl-bluemix-autoscale>

4.4 Communication, session persistence, and logging in Bluemix

In this section, we describe communication, session persistence, and logging in Bluemix.

4.4.1 Communication

As mentioned earlier, in a microservice architecture, you end up having several independent applications, with each running in their own isolated environments and written in the languages and frameworks best for their individual purpose.

Each of these individual applications is deployed to Bluemix, and is accessed through their own URL. For example, one of these microservices applications might have the following URL:

`myapplication-OrdersService.mybluemix.net`

In Figure 4-16, you can see the microservices communicating with each other. Naturally, microservice architecture supports both synchronous and asynchronous communication between the different applications. Consider how we can implement each of these in Bluemix.

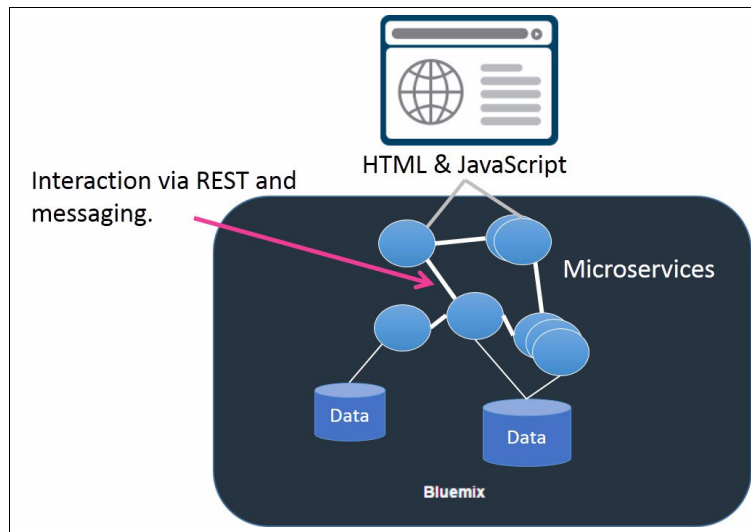


Figure 4-16 Communication between applications in Bluemix

Synchronous

Imagine a scenario where our UI microservice application is querying a catalog microservice application for a list of items to render. The catalog application needs to expose a URL endpoint where the client (UI) can call to get the results. For this type of scenario, the suggested protocol is to use REST over HTTP.

REST is a simple stateless architecture that uses existing web technology and protocols, essentially HTTP. It is a simpler alternative to SOAP. The RESTful API establishes a mapping between **create**, **read**, **update**, and **delete** operations and the corresponding **POST**, **GET**, **PUT**, and **DELETE** HTTP actions.

Being an API, the RESTful interface is meant to be used by developers within application programs, and not to be directly started by users. So in our scenario, the catalog application exposes a RESTful API, and the UI application makes a call using the appropriate HTTP action.

Because every application on Bluemix is on a different host name, if you are attempting to make requests from one application to another using a client-side JavaScript, it is important to know that you will run into Same-Origin Policy (SOP) that a browser imposes for security issues. (Under the SOP, a web browser permits scripts contained in a first web page to access data in a second web page, but only if both web pages have the same origin.)

Therefore, in our situation, this SOP does not permit JavaScript running and associated with one Bluemix application to make requests to another web page related to another Bluemix application. One solution is to make these requests on the server side. Alternatively, you can implement a *cross-origin resource sharing (CORS)* solution, which allows a providing service to define a whitelist of allowable hosts that are allowed to make requests, as outlined in the following link:

<https://developer.ibm.com/bluemix/2014/07/17/cross-origin-resource-sharing-bluemix-apis/>

Asynchronous

Next, consider a scenario where one microservice application needs to start one or many microservice applications, but we don't need to wait for the action to complete. In our store scenario, when a user purchases an item using our UI application, it needs to tell the orders application about the order, but it does not need to wait until the order is processed. The orders app needs to alert an email notification or shipping microservice in a similar way.

Messaging is an effective way to communicate between these applications. Messaging uses an asynchronous model, which means that an application that generates messages does not have to run at the same time as an application that uses those messages. Reducing the requirements for simultaneous availability reduces complexity, and can improve overall availability. Messages can be sent to specific applications or distributed to many different applications at the same time.

We can perform asynchronous work in various ways. It can queue large volumes of work, and that work is then submitted once a day by a batch process. Alternatively, work can be submitted immediately, and then processed as soon as the receiving application finishes dealing with a previous request.

IBM MQ Light for Bluemix is a cloud-based messaging service that provides flexible and easy to use messaging for Bluemix applications. IBM MQ Light provides an administratively light solution to messaging. You can choose which API to use based on your requirements, your system architecture, or your familiarity with a particular technology. You can use IBM MQ Light to enable applications that use different run times to communicate with each other using the AMQP protocol.

AMQP: AMQP stands for Advanced Message Queuing Protocol. It is an open standard app layer protocol for message-oriented middleware, providing message orientation, queuing, routing (both point-to-point and publish-and-subscribe), reliability, and security.

In our store order scenario, every time we receive a new order in the UI application, we can implement a worker offload pattern where we have one or more Orders applications or *workers* to share the work among themselves. As orders increase, we can simply add more workers without having to change anything.

4.4.2 Session persistence

When a user visits your application and establishes a session, how can you ensure that you maintain the same session on subsequent requests when you have multiple instances of the same application serving requests? In Bluemix, when you scale your instances, each instance runs in its own isolated environment. *Memory and disk are not shared among instances.*

If the application is storing user sessions in memory, the same instance of the application needs to handle the request every time. To ensure that requests are routed to the same instance, session affinity (*sticky sessions*) for incoming requests can be established if a `jsessionid` cookie is used.

However, storing session data in your application server memory and relying on session affinity is not a reliable architecture. Instances of your application need to be able to scale up and down dynamically, and each instance should not contain any data that cannot be recovered. If an application instance crashes or gets scaled down, the session data needs to be able to be recovered using another instance. For a more dependable architecture, applications should persist the session data using an external service.

Session Cache

Session Cache is a caching service that stores and persists Java HTTP session objects to a remote data grid. The data grid is a general-use grid that stores strings and objects. Session Cache remotely leverages the caching capabilities of the data grid and enables you to store session data. The session caching service provides linear scalability, predictable performance, and fault tolerance of a web application's session data.

Session Cache provides a NoSQL-style, distributed key-value storage service, and provides the following features:

- ▶ Low-latency data access
- ▶ Transactional semantics
- ▶ Dynamic scalability of both data and transactional capacity
- ▶ Data replication to prevent data loss during system outages
- ▶ Integrated web console to monitor service instance performance and usage

Currently, this service is only supported for Java EE applications on Bluemix running on Liberty for Java. To use this service, simply create and bind the service to your Liberty application on Bluemix, and the necessary configuration is already created for you. No further action is required to achieve session persistence. The Liberty buildpack installs the required data cache code into the Liberty server, and generates cloud variables for the bound Session Cache instance.

After you connect your application to the IBM Session Cache for Bluemix, you can monitor your web application's caching usage. Click a service instance from the Bluemix console dashboard in order to open the charts for your application.

Data Cache

For other types of applications, the IBM Data Cache service can be used to achieve similar session persistence. It is a caching service that supports distributed caching scenarios for web and mobile applications. It uses a data grid in which you can store key-value objects.

The service provides a language-neutral REST interface to access services. Use the HTTP **POST**, **GET**, and **DELETE** operations to insert or update, get, and remove data from the cache. Each REST operation begins and ends as an independent transaction to Data Cache. These operations can be used to store and retrieve user session values.

Figure 4-17 shows the Session Cache contents.

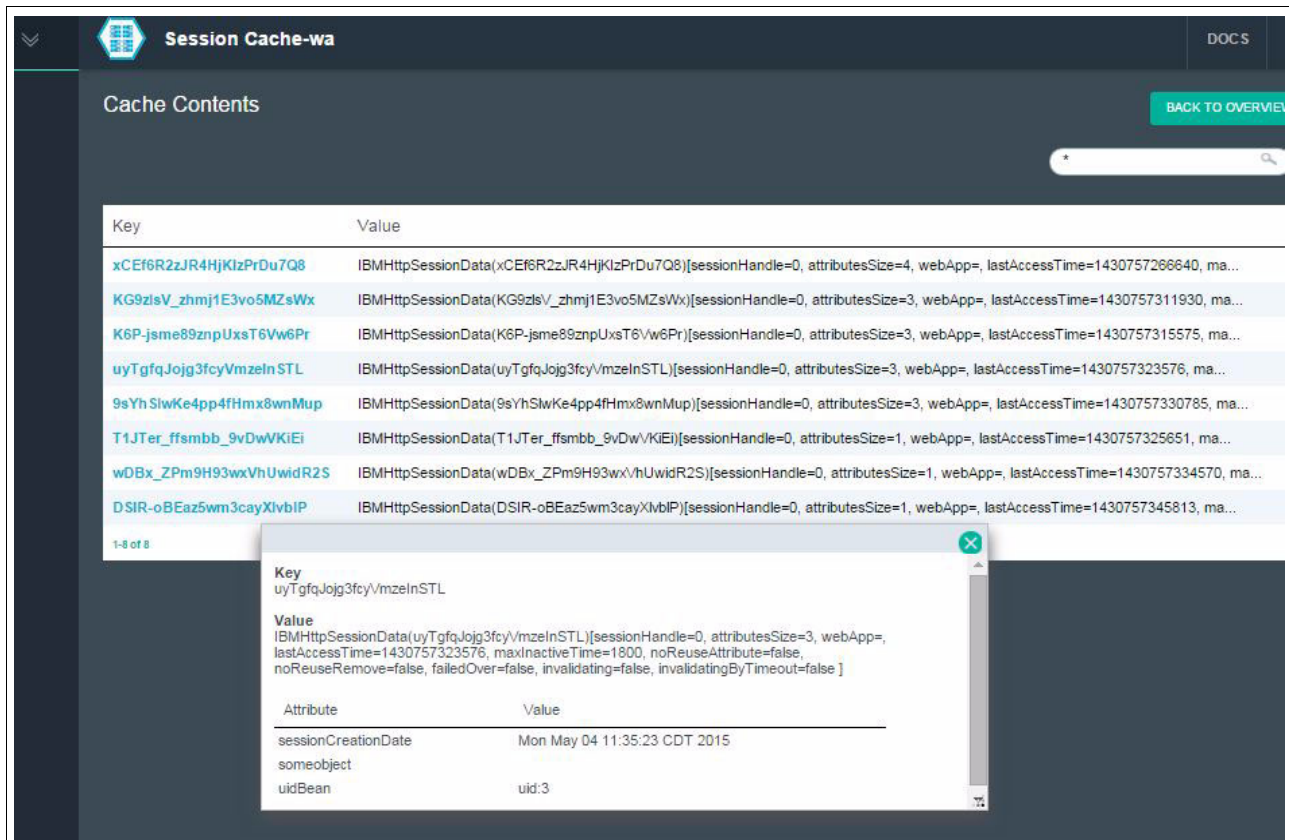


Figure 4-17 Session Cache contents

Bluemix also provides many third-party services, such as Redis, which can be used for storing session data. However, it is currently a community service that is as-is, and is provided for development and experimentation purposes only.

4.4.3 Logging in Bluemix

It is important to view your log files to analyze your application, and use the log statements to gain insight into what is happening in the application throughout the application lifecycle. With Bluemix, that is based on Cloud Foundry, access to logs from an application are exposed using something called the *Loggerator*, and all of the logs from your app are made available through it. The most common way to access log files is using the Cloud Foundry CLI:

\$ cf logs

The **cf logs** command enables you to access the logs from your application deployed to Bluemix directly from your command line, so it is extremely useful when performing development. You can access a snapshot of recent logs or stream logs directly to your command line.

Some basic uses of the `cf logs` command are shown in Table 4-4.

Table 4-4 Basic uses of the `cf logs` command

<code>cf logs appName -recent</code>	This command shows a subset of recent logs from the application deployed to Bluemix with the name <code>appName</code> .
<code>cf logs appName</code>	This command appends the logs from the application deployed to Bluemix with the name <code>appName</code> .

When your application is deployed and somewhat stable, you want to use a logging service. There are a couple of reasons to use a logging service. The first reason is that using `cf logs appName -recent` only enables you to see so far back in your logs. Second is that logs are deleted after the application restarts, and restarts can happen as a normal course of operation, especially if you are practicing continuous integration and delivery. There are several third-party logging services that you can use for Bluemix, including Logentries, Papertrail, Splunk Storm, and SumoLogic.

4.5 Considerations and opportunities

This section provides an overview of important areas you should consider as you begin to design, develop, deploy, and run microservices on Bluemix. These considerations identify areas of existing or emerging needs, and provide suggestions on how to address those needs. In some cases, the technology is emerging or not yet available.

This information can affect your choices for which types of microservices you deploy on Bluemix (Cloud Foundry, containers, or virtual images), and where you deploy them (Bluemix public, dedicated, or local). This section also identifies other areas where future focus makes sense, based on the client microservice adoption trends in the marketplace.

Important: These considerations are a snapshot in time. Given the rapid pace of change, new capabilities are being added all the time to Bluemix. You should review the current version of the documentation to see if there are new capabilities that address these considerations:

<https://www.ng.bluemix.net/docs/>

4.5.1 Microservice trends to consider

This section identifies emerging trends and challenges using microservices. The authors are providing their own opinion of likely areas where new capabilities will be required to address these challenges.

Continued emergence of new compute types

As microservices are implemented across various deployment models and compute type infrastructures, it quickly becomes obvious that a common set of platform services that spans these are required to reduce IT maintenance costs and facilitate their use. Microservices platforms continue to evolve and provide new or improved services for event notification, scaling, logging, monitoring, and single sign-on (SSO) authentication across Cloud Foundry applications, containers, and virtual images.

Increasing need for services to be local and more secure

It is a global economy, with customers in different countries with different privacy and data concerns. This often requires that a user's personal data be local to the country. Also, application response can be negatively affected by latency. By offering microservices that are colocated with your users, these issues can often be addressed. These needs continue to drive demand for support for Bluemix in more geographic regions. See the documentation (https://www.ng.bluemix.net/docs/overview/overview.html#ov_intro__reg) for the list of the currently available Bluemix regions.

Some microservices are only available on a PaaS multi-tenant environment. Continued security needs also drive more customers to demand microservices in a single tenant dedicated environment, where both hardware and software running and managing those services are dedicated to a client. Microservice providers are asked to provide their services in local and dedicated deployment models, too. When choosing a platform, carefully consider where services are available.

Expanding number of services and open source

An ever-increasing number of microservices will be available in the market, making it harder to find differentiating services. Microservices platforms must provide effective keyword and multi-faceted searching and filtering of services. Open source and community-based microservices will continue to grow. Bluemix already addresses many of these needs with the Bluemix catalog. As described earlier, it enables you to find IBM, third-party, and community supported services.

Continued increasing competitive pressures

Relentless competitive pressure is driving demand for more differentiation. Service differentiation is what causes clients to stay with a PaaS, or to leave it for another one. The authors expect that PaaS vendors will continue to acquire or build unique differentiating microservices as a way to entice clients and remain the market leaders. IBM Bluemix provides unique and differentiating cognitive service offerings from Watson.

Another benefit of IBM Bluemix is the openness of the platform to prevent vendor lock-in and provide the ability to move as needed. This is another driver for competitive differentiation, to truly deliver value to the clients and their users.

Note: It is also important to mention that IBM recently acquired Alchemy API (<http://www.alchemyapi.com/>), which enables further enhancements to Bluemix services.

4.5.2 Controlling access and visibility of endpoints

Many clients are offering their APIs and microservices publicly as another way of monetizing their business. Bluemix enables you to expose the routes and endpoints of your services, so that they can be accessed by your clients. In some cases, however, you might not want your microservice endpoints to be publicly visible, or might want to limit access to certain users or applications. *IBM API Management Service* enables you to catalog and control access to your microservice REST APIs.

You can also throttle the number of requests allowed to your microservices by specifying rate limits on the number of calls a client can make to your API. Today the service does not allow you to not make those endpoints visible. Consider using obscure host names and API keys to further restrict access, or consider using a software-defined networking (SDN) third-party service.

You can learn more about IBM API Management Service in the documentation on the following website:

<https://www.ng.ibm.com/docs/#services/APIManagement/index.html#gettingstartedapim>

Another approach that requires more investment is to use Bluemix Local. Services deployed in Bluemix Local are not visible outside your own corporate firewall. So although this option does prevent visibility from the public, it does not prevent access to those people or applications within your organization. You can enable access to your Bluemix Local microservices from publicly hosted Bluemix applications by using the IBM Secure Gateway:

<https://www.ng.ibm.com/docs/#services/SecureGateway/index.html#gettingstartedsecuregateway>

You can also use the IBM Cloud Integration Service:

<https://www.ng.ibm.com/docs/#services/CloudIntegration/index.html#gettingstartedwithcloudintegration>

To prevent access, you can also consider using API keys or tokens to only allow those with API keys to access the microservice.

4.5.3 Avoiding failures

As mentioned in Chapter 2, “Elements of a microservices architecture” on page 19, one of the microservice architectural patterns is detecting a failure condition. Following that, the pattern circumvents the service failure using a circuit breaker approach that reroutes similar types of future requests to alternative microservice providers. Bluemix provides some services to proactively prevent some types of conditions that could cause microservice failures.

For example, you can enable Bluemix auto-scaling service to prevent microservice applications from running out of memory or CPU. It is also worth reiterating the information in 2.3, “REST API and messaging” on page 33, about using IBM MQ Light to decouple your microservices so that they do not hang when one service becomes unavailable.

Service monitoring

Bluemix provides a Bluemix service status page (<https://status.ng.ibm.com/>) to see the current state of services and information regarding event outages. You can subscribe to the Really Simple Syndication (RSS) feed. You could occasionally call this RSS feed to detect error conditions and make corresponding mitigation actions in your application. A useful capability would also be to be notified using email or text message when services become unavailable.

Another approach to further improve resiliency in your application architecture is to consider including Hystrix as part of your application architecture pattern. Yet another approach is to monitor log files using service for error conditions, and send an email notification using SendGrid or text message using Twilio. Finally, also consider using service monitoring and notification services like Pingdom.

Incremental deployment rollouts

The best approach is to avoid failures completely by completing rolling deployments of new microservices, and then evaluating their effect on the overall system. By incrementally providing access to more and more clients and closely monitoring the microservice, you can detect anomalies and revert the deployment. A similar approach to avoiding microservice failures is preventing problems during deployment.

Use a blue-green deployment approach where you have two production environments. The live one is on blue servers, and the new microservice that will replace the existing service is on the green server. The environments should be as identical as possible. After the green server has tested, it becomes the live server by switching traffic to the green server.

Load balancing

Another approach to avoiding application failures is to providing load balancing across the data center regions. If a data center goes down, being able to route requests to the same applications deployed in another available data center can ensure continuity of microservices.

At the time of publication, IBM Bluemix does not provide a way to do application failover across regions or public and local instances of Bluemix, but third-party services can provide these services for use with Bluemix. For example, global load balancing can be handled using domain name server (DNS) providers, such as Dyn or NSONE, which support geographic routing and failover policies. You make one of these offerings as your authoritative name server, and configure the policy that you want.

4.5.4 Versioning

After microservice versions are released into production, client feedback results in new versions of the microservice being made available. Microservice developers are responsible for providing clear direction on microservice versions as they go from beta to production and providing notifications when they are sunset. Services listed in the Bluemix catalog typically show if a service is experimental, beta, or generally available as part of their metadata.

Service versioning, however, is the responsibility of the service providers. The service version of the APIs should be described in the documentation of the service. Bluemix does not surface microservice versions in the user interface.

One approach for clearly communicating microservice versioning in Bluemix is to add version identifiers to the microservice. Clients can then reach your microservice API's endpoints using one of the following methods:

- ▶ URL path
- ▶ Header
- ▶ Accept_version_header
- ▶ Parameters

Example 4-2 shows versioning using the URL path.

Example 4-2 Versioning microservice using URL path

```
https://www.googleapis.com/youtube/v3/playlists?key={YOUR_API_KEY}
```

The easiest approach is simply using a URL parameter, as shown in Example 4-3. In cases where a version is not provided, you use the current version.

Example 4-3 Versioning microservice using URL parameter

```
https://www.googleapis.com/youtube/playlists?version="3beta"&key={YOUR_API_KEY}
```

Tip: It is generally a good practice to avoid using version identifiers in the names of a service, and to ensure that new changes do not break compatibility with an earlier version.



Microservices case studies in IBM

This chapter provides the implementation examples of microservice architecture in some IBM products as case studies, in order to capture the application of microservice architecture in building real-life, enterprise grade solutions. We share our experiences and lessons learned when redesigning these products using the microservices approach.

We describe the following case studies:

- ▶ 5.1, “Microservices implementation in IBM DevOps Services” on page 92
- ▶ 5.2, “Microservices case study in Bluemix console” on page 96
- ▶ 5.3, “Microservices case study in IBM Watson services” on page 103

5.1 Microservices implementation in IBM DevOps Services

IBM DevOps Services is a website designed to enable developers to write applications (apps) *in the cloud, for the cloud*. It has support for the following functions:

- ▶ Projects and source control, such as Git, GitHub, and Jazz SCM
- ▶ Source code editing using the provided web integrated development environment (IDE)
- ▶ Tracking and planning
- ▶ A delivery pipeline that builds and deploys the code to IBM Bluemix

IBM DevOps Services can be accessed on the following website:

<http://hub.jazz.net>

Because IBM DevOps Services started as an experiment aimed at academic use, it was originally designed as a monolithic Java Platform, Enterprise Edition (Java EE) application for expedience. As such, it follows the usual path we have seen over and over: An application that grows beyond a manageable size, and eventually reaches a point where redesign is required.

The baseline system before the microservices transformation consisted of two major parts:

- ▶ A JEE-based application for most of the project and user management tasks
- ▶ Another system based on the open source Orion project for web IDE, source control, and simple deployment to Bluemix

Because all of the servers, databases, and other applications were running in virtual machines (VMs), their internal Internet Protocol (IP) addresses were not visible from the outside. The externally accessible paths into the system were exposed using the Apache server running as Hypertext Transfer Protocol (HTTP) or Hypertext Transfer Protocol Secure (HTTPS) reverse proxy.

Running a reverse proxy in front of a system is a standard practice, and is not particular to microservices architecture, but it comes useful when evolving a system. It provides for a Uniform Resource Locator (URL) architecture as an application programming interface (API), a site map that is the only thing users see. Paths of this site map can incrementally be moved to microservices without disruption.

Figure 5-1 shows an IBM DevOps Services starting point, as describe previously.

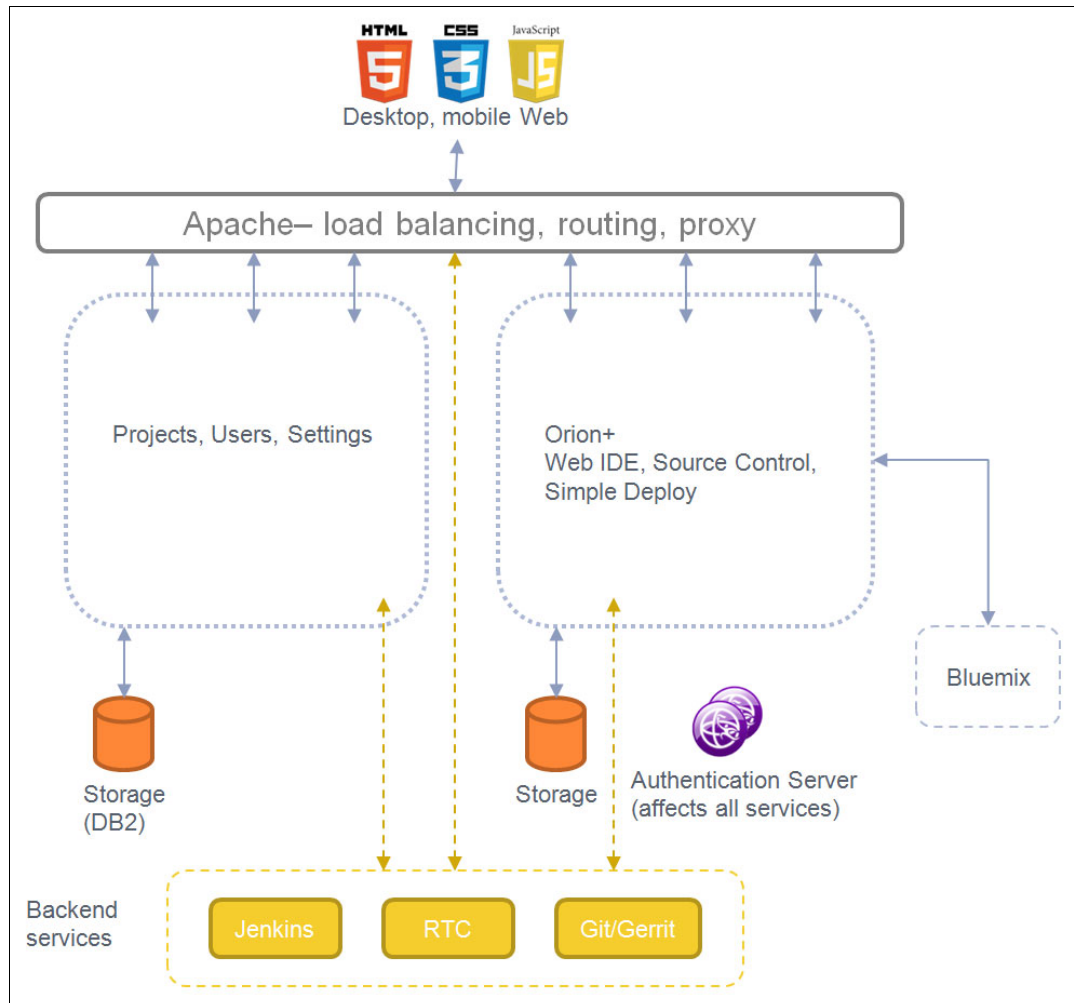


Figure 5-1 IBM DevOps Services starting point

5.1.1 North Star Architecture

When assessing how to go about the microservices transformation, it was imperative for us to keep the system running and operational at all times. However, even though evolution rather than wholesale switch was in order, we had to know where we were going. To that end, we created a *master plan*, or what we used to call *The North Star Architecture*.

Figure 5-2 on page 94 shows the The North Star Architecture for the DevOps Services system, where various Node.js-based microservices were designated to deliver various features, with web user interface (UI) and Representational State Transfer (REST) API tasks clearly separated. The following microservices were designated to provide the system “glue”:

- ▶ UI composition
- ▶ Usage metering
- ▶ Activities
- ▶ Asynchronous messaging

Figure 5-2 shows the master plan for DevOps Services.

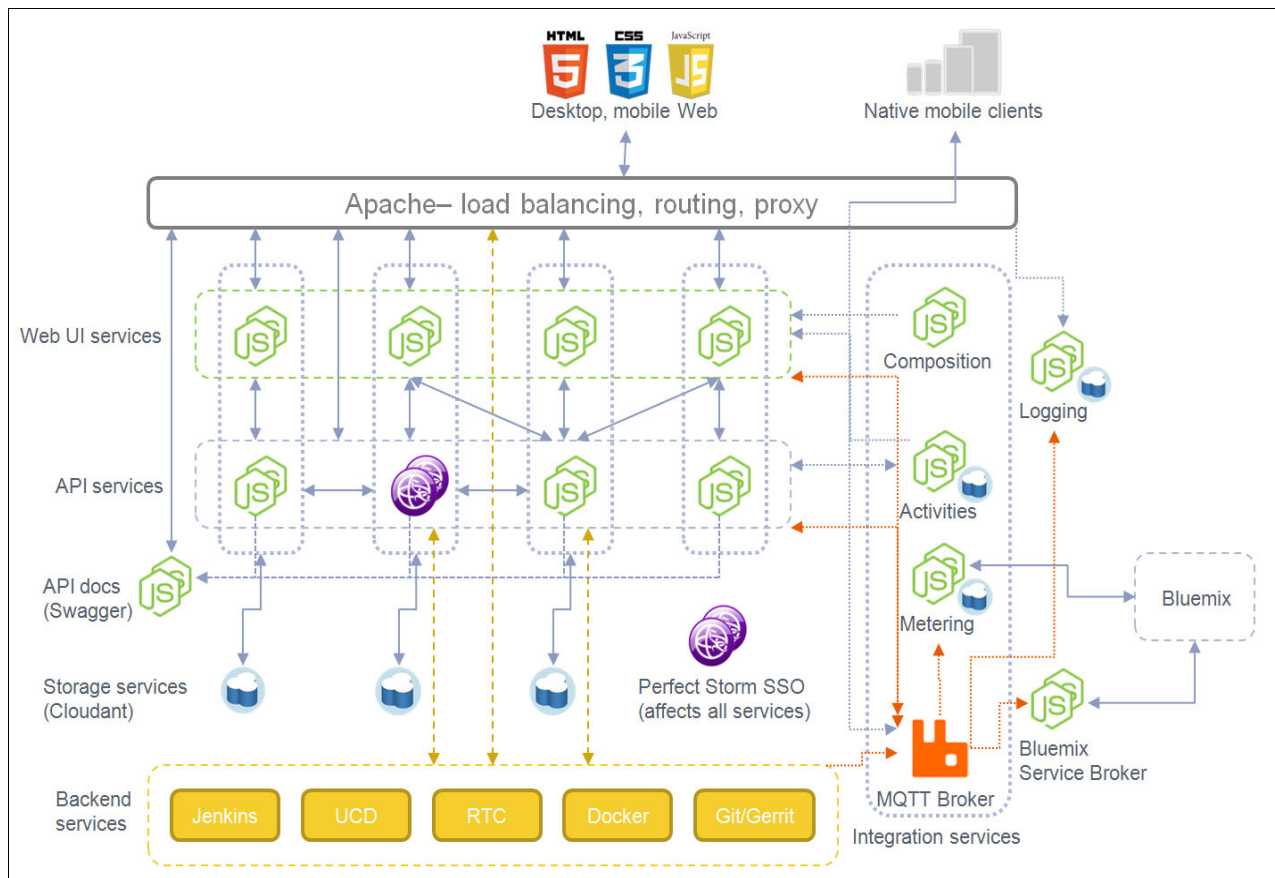


Figure 5-2 The North Star Architecture for the DevOps Services system

The plan was to have this architecture in mind as the end goal, and use it to inform all tactical steps toward its full realization. In order to ensure this goal, we devised a set of rules for code transformation:

1. When adding an entirely *new feature*, write it by following the North Star architecture to the fullest.
2. When attempting a refactoring of an *existing feature*, assess the extent of the rewrite. If the cost of rewrite is comparable to writing a new feature, take the opportunity to implement it as a microservice using the new architecture, and retire the old feature in the monolith.
3. Repeat step 2 until the monolith dissolves.

Our first microservice followed Rule 2. We wanted to redesign the way we served tutorials to make it easier for content authors to be productive, and to update the content without the slow and costly monolith redeployment. We stood up a Node.js microservice serving tutorial content authored in Markdown, and pointed at it using the Apache reverse proxy.

This is an example of the incremental evolution that all existing systems must undergo. Our users did not notice any disruption, except that new tutorials arrived at a faster cadence, and we were able to react to typographical errors and technical errors with much more agility.

5.1.2 Delivery pipeline

Our first example following Rule 1 was the entire delivery pipeline. In a sense, it was our “poster child” of the *new way of working*. It was designed entirely around the North Star architecture, using Node.js for the UI microservice, Java for the REST API service, and several back-end systems (Jenkins, Docker) for running builds and deployments. It was also our first use of the message broker (RabbitMQ using the MQ Telemetry Transport (MQTT) protocol).

Combining MQTT asynchronous messaging and web sockets between the Pipeline UI microservice and the browser allowed us to create a dynamic user experience. As a useful anecdote, we got into a situation where build failures were showing in our Pipeline UI faster than in the native Jenkins web UI (because the MQTT/web sockets server push was sending events as they were happening, where Jenkins relied on periodic server polling to update the UI).

Delivery pipeline can be a useful example for microservice sizing. Neither pipeline UI nor pipeline REST API microservices are particularly *micro* (we like to call them *mini-services*). Nevertheless, they are a great example of *right-sizing*. The UI microservice serves several pipeline pages, and reuses many Dust.js partials. Similarly, the REST API service is a logical owner of all pipeline API endpoints. These endpoints do many things, but there is still a sense of a single purpose for each of these microservices.

5.1.3 REST and MQTT mirroring

One of the techniques we found quite useful in our use of the message broker is something we dubbed *REST and MQTT mirroring*. This technique is applied to all REST API services that manage resources that can change state. We found that in a microservice system, starting REST APIs and caching the state is not a good way to go on its own. The system is always in a situation where trade-offs need to be made between operating on the stale state and degrading system performance with overzealous polling.

For this reason, we realized that microservices need to know when the state they are interested in changes without the polling. Because this is a universal need, we thought that coming up with a standard approach would help the consistency of the architecture.

The mirroring simply uses the end-point URL as an MQTT topic. A downstream microservice using a REST API endpoint can also subscribe to the MQTT topic that matches the end-point syntax. A microservice starts by making an HTTP **GET** request to obtain the initial state of the resource. Subsequent changes to the resource made by any other microservice are tracked through the MQTT events.

Events were designed to transfer the complete information about the change (*created*, *modified*, or *deleted*). They also contained the new state of the resource for created and modified events, so that an additional REST request was not needed for the downstream microservice to handle the event.

We found that with this technique, we were able to make the system more up to date without the strain that more frequent polling brings to the system.

5.1.4 Deployment considerations

IBM DevOps Services was our first experience with the microservice architecture. As such, we felt the need to fully control all aspects of the architecture and the infrastructure. For example, we felt that the MQTT protocol in its simplicity and ease of use is a great companion to the REST APIs. We chose the open source RabbitMQ message broker for this task, and wanted to have full control over its installation and configuration.

DevOps Services system operated on IBM SoftLayer, as an infrastructure as a service (IaaS), from day one. When transforming the system to microservices, we simply added more VMs, and started deploying and running new services on them.

One of the conclusions we made from our first microservice system deployment was that IaaS might be too low an abstraction level. Because we did not have the luxury of the Delivery Pipeline (because we were just building it), we had to run and manage the following components:

- ▶ IBM UrbanCode Deploy for extracting code from IBM Rational Jazz source control management (Jazz SCM) or Git, building it, and deploying to Bluemix
- ▶ Open source utility PM2 for running and clustering Node.js instances (and restarting them when they occasionally crash)
- ▶ In-house monitoring service designed to work with PM2 and track app memory and processor (CPU) usage
- ▶ RabbitMQ for messaging
- ▶ Apache reverse proxy for the external routes
- ▶ Several VMs for the existing parts of the system, running a collection of Java EE containers and databases

This all added up to a somewhat costly and high-maintenance system, particularly considering the fact that a large portion of this functionality was already available maintenance-free in products as a service (PaaS) such as Bluemix. As a result, our next implementation of a microservice system took advantage of what Bluemix had to offer.

5.2 Microservices case study in Bluemix console

Bluemix console (the name of the single-page app that is powering the web experience of IBM Bluemix) started its life as a monolith. Again, in most cases this is the most practical thing to do initially. It allowed us to make quick progress with a few parts to manage and worry about.

However, as Bluemix grew in scope and complexity, some things became apparent:

- ▶ Bluemix is really two things in one:
 - A website that can be freely accessed, and which contains large amounts of searchable and indexable content.
 - A dynamic web app (Dashboard) that is accessed by logging in.These two portions have different goals and priorities and, over time, it became costly to maintain both in a single large application.
- ▶ As both of these parts of Bluemix grew in size, it became harder for teams to work on portions of the app and deploy these portions independently.

One of the lessons of any large web property is that more often than not we deal with a site and an app (or multiple apps) folded into one. Such an arrangement is ideal for microservice architecture, so that different parts can be evolved at their own speed, and use the best technology for their intended use. See the following website for more information:

<http://dejanglozic.com/2015/01/26/should-i-build-a-site-or-an-app-yes/comment-page-1/>

5.2.1 Assessing the task

As with DevOps Services, we start by establishing the baseline (the system before the microservice evolution). This is shown in Figure 5-3.

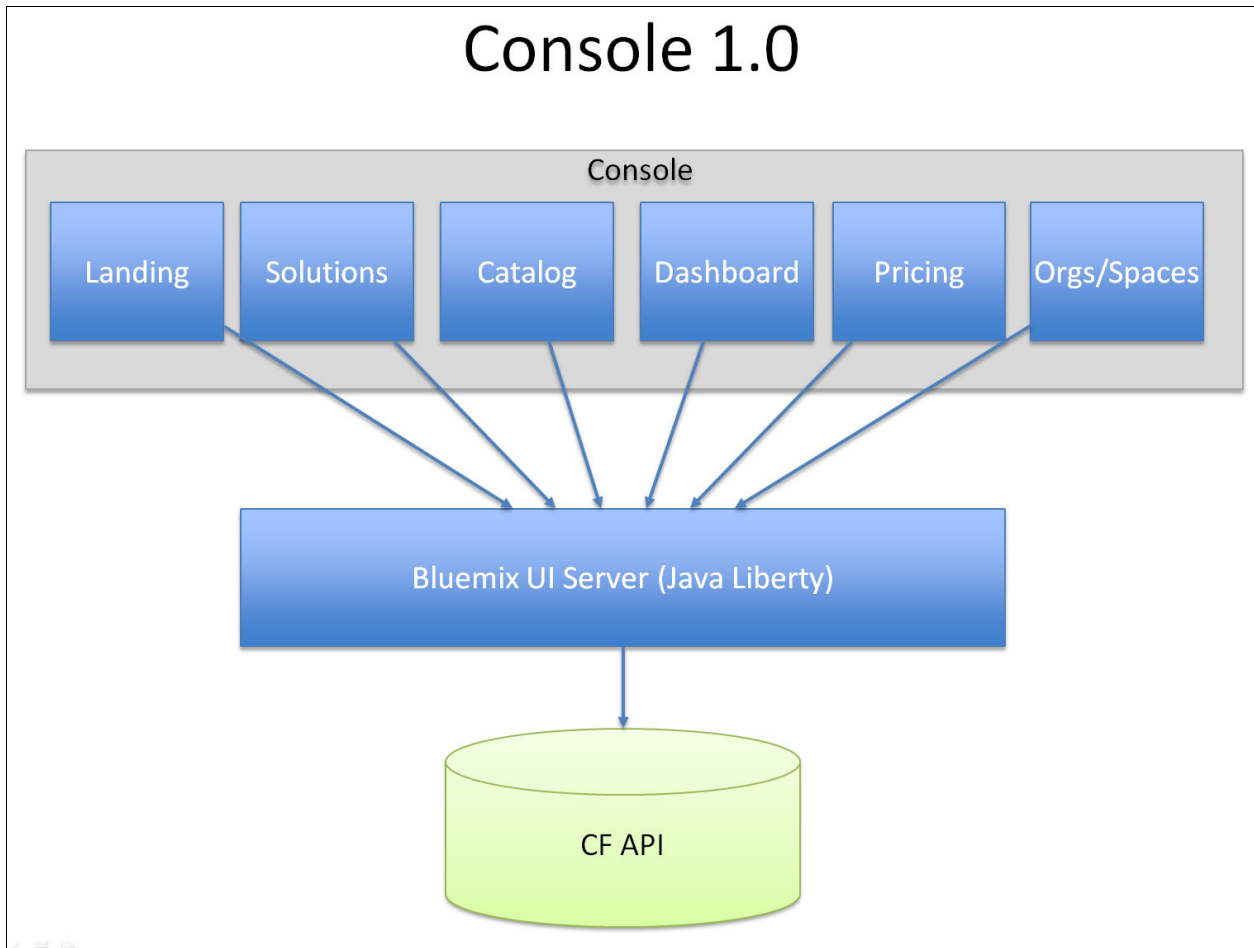


Figure 5-3 High-level architecture of the Bluemix console before the microservice evolution

All of the individual features of Bluemix were part of the single-page app, served by a single Java EE server, and calling the Cloud Foundry API for most of the state. When a change was made to the home page or one of the solution pages (all publicly accessible, crawlable content), the entire application had to be redeployed. After a while, this becomes a significant detriment to further evolution of the platform.

Of course, Bluemix is a production system that many applications rely on for continuous availability, and it had to be evolved with continuity of operation. We have therefore picked a portion that is relatively straightforward to peel off: Public content.

You might notice that this parallels our experience with IBM DevOps Services. The first feature to peel off there were Tutorials. High-churn, frequently updated content is typically first to create a bottleneck, because more sensitive, dynamic parts of the system are normally updated less often and with greater care.

5.2.2 Peeling off Landing and Solutions

As we have said numerous times in previous chapters, reverse proxies are the cornerstone of microservice architecture. They maintain the illusion of a single entry point while dealing with a highly federated system. We similarly had to set up our own for the Bluemix console. After that, we simply stood up two microservices:

- ▶ One for the landing page
- ▶ Another for several marketing pages accessible through the Solutions section in the common header

Being able to optimize microservices for the current task has been proven again in this exercise. Because both Home and Solutions were designed to serve relatively static content, there was no reason to accept the consequences of client-side frameworks, which would make no sense in this use case. We are not saying that sophisticated client-side framework, such as AngularJS, Backbone, or EmberJS, are not without merit, only that they would be an incorrect choice to serve public content that needs to be crawlable by search engines.

Relatively simple Node.js microservices combined with Express framework and Dust.js rendering engine (and a tasteful use of jQuery and JavaScript for dynamic behavior) was all that was needed. Figure 5-4 on page 99 shows the Bluemix site in one of many intermediate states. Although the landing and solution pages are served by the new microservices, the rest of the site is still provided by the old Console app. The process continues until the box on the right is empty and all of the features are handled by the microservices.

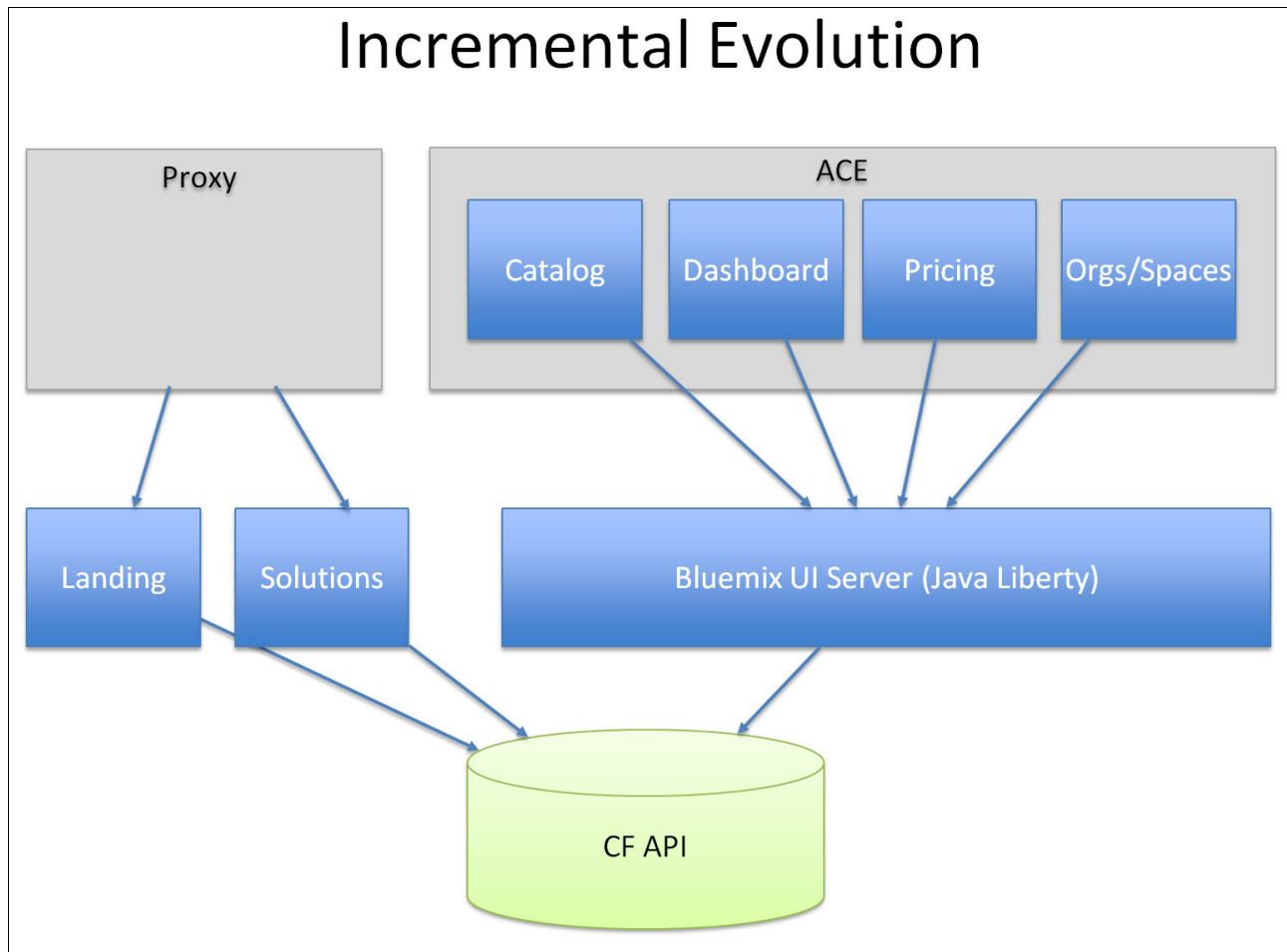


Figure 5-4 Bluemix site in one of many intermediate states

5.2.3 Composition of the common areas

Starting with the microservices architecture for Bluemix console forced us to solve an interesting problem: How to handle common areas that all of the microservices responsible for serving web pages should use. A large application has an advantage of running as a single process. All of the pages can share common templates using simple partial inclusion. When these pages are served by separate applications, this inclusion involves network requests.

We ended up solving this problem by introducing a microservice dedicated to serving common elements. It hosts Cascading Style Sheets (CSS) and JavaScript (JS) needed by these common elements, and using these resources is trivial thanks to `link` and `script` tags in Hypertext Markup Language (HTML).

However, stitching the HTML together required network requests by the service that served the page. The common service was designed to serve the common areas as HTML snippets, not full pages. These snippets would be inserted into the final page HTML on the server. Because service-to-service requests are low-latency calls between peers in the same data center, the network request resource usage is minimized.

To further minimize UI composition resource usage, callers are encouraged to cache the result for a short period of time (5 - 10 minutes) to improve performance and reduce pressure on the common microservice.

The advantage of this approach is that it is stack-neutral. It is reduced to basic HTTP and HTML, which ensures that any microservice, irrespective of the chosen technology, can use it while serving pages. In addition, it uses the normal OAuth2 bearer token authentication to pass the user profile information. This ensures that common areas that are customized to the user can be served by a single common service.

Of course, the service can also serve a public (unauthenticated) version of the common element. Figure 5-5 shows the UI composition of the Bluemix landing page. The common microservice provides header and footer snippets, which are combined with the content and served as a complete page by the Landing microservice. This process is repeated for all of the pages using these common elements. Styles and JavaScript are sourced using normal HTML mechanisms.

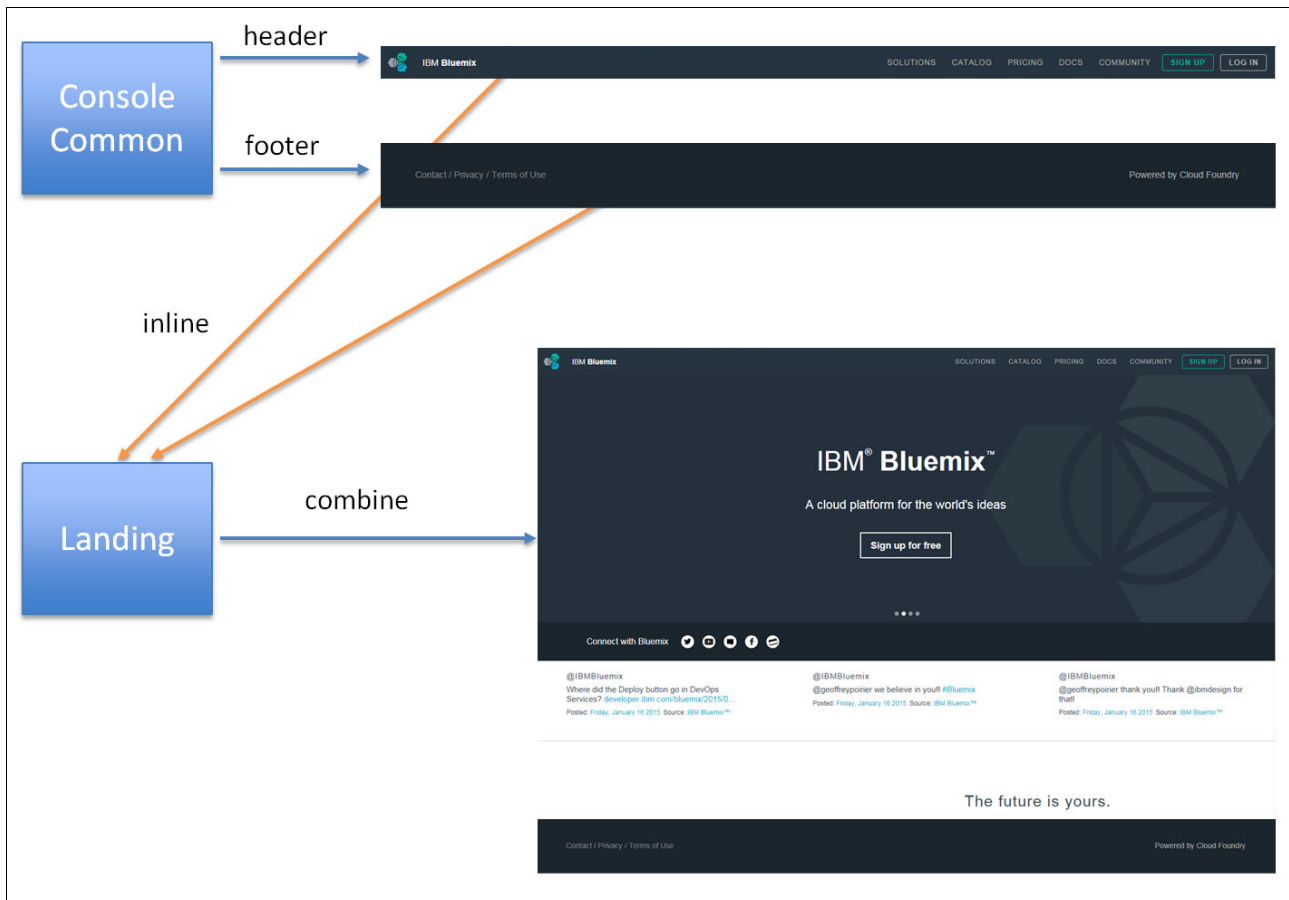


Figure 5-5 UI composition of the Bluemix landing page

5.2.4 “Lather, rinse, repeat”

An old adage says that *the first step is the hardest*. Starting the microservice evolution is often the hardest part, because certain elements that will be shared by all microservices need to be put in place before microservice #1 can be pressed into service. When the first couple of services are up and running, it is easy to follow the established practices and continue.

In our case, we had to solve authentication sharing, UI composition, and reverse proxy issues before proceeding. After these problems were solved, we were able to address areas with the most churn, such as Home and Solutions. With the architecture and DevOps mechanisms in place, the continuation of the evolution is simply a matter of identifying the next area of the system to address, and switching the proxy to the newly erected microservices. At the time of writing of this book, this process is still ongoing, but we can easily predict where it will end (after all, we are now used to the North Star architecture approach that we continue to follow).

Figure 5-6 shows the wanted end-goal of the Bluemix console microservice transformation. All features of the site are served by dedicated microservices and can follow their own development and deployment schedule. In addition, they can be optimized for the particular task (serving content, or assisting users with apps, services, containers, and VMs in a highly dynamic and interactive fashion). There is no need for a *one size fits all* technology approach any more.

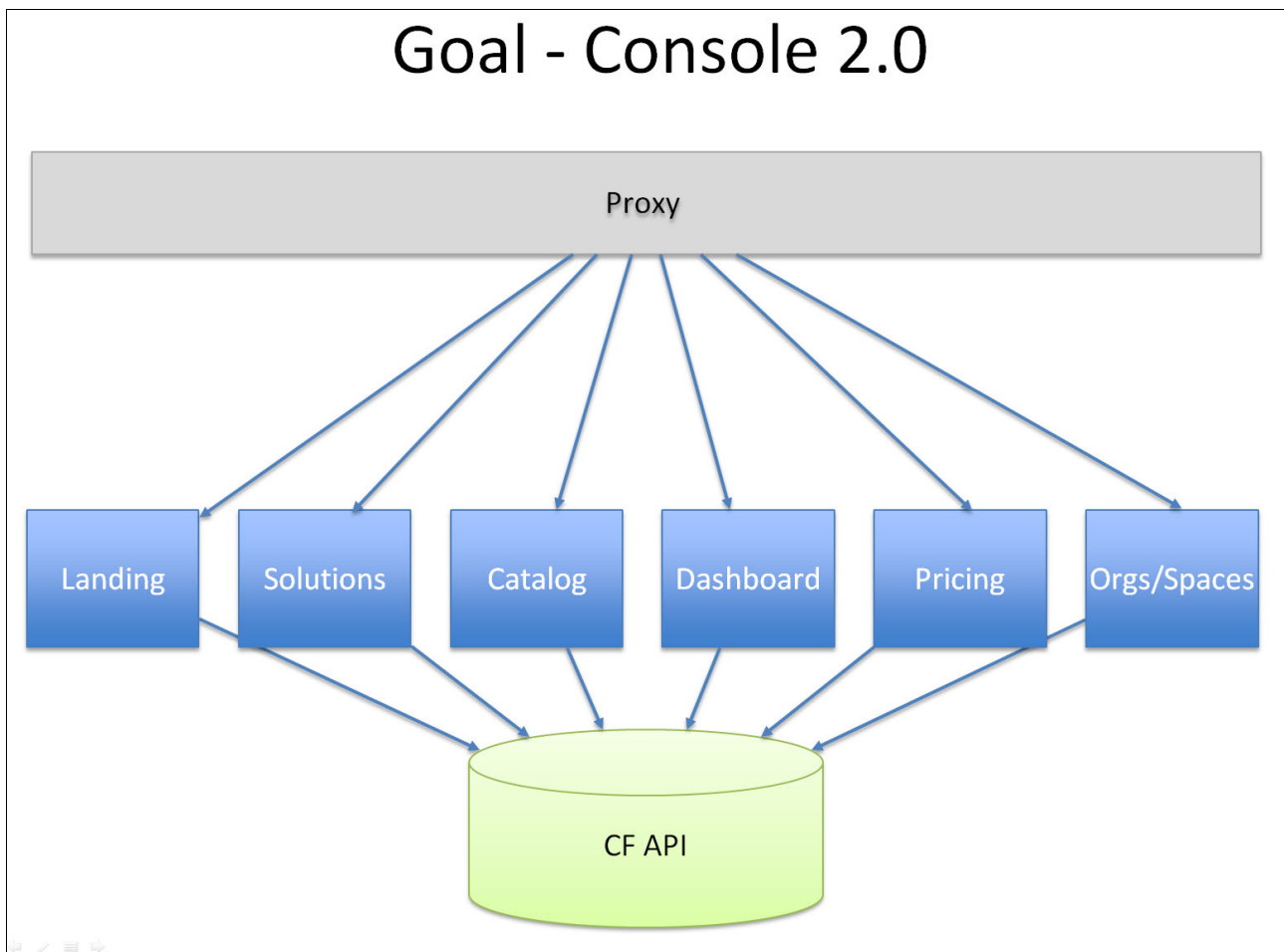


Figure 5-6 The wanted end-goal of the Bluemix console microservice transformation

5.2.5 Deployment considerations

We are repeating the title of the subsection from IBM DevOps Services for a reason. We would like to contrast how our second attempt to build a microservice system changed based on our initial experiences.

Because Bluemix is a PaaS, and because the technologies we selected for our microservice evolution are highly compatible with running in a PaaS environment, it was a foregone conclusion that we were to run Bluemix in Bluemix (also a useful self-hosting opportunity). To contrast this approach with IBM DevOps Services, we list the improvements we were able to put into place:

- ▶ We had no need for PM2 any more, because Bluemix gave us the immediately available ability to run, load-balance, and keep the apps running.
- ▶ We had no need for an in-house monitoring tool, because Bluemix comes with the monitoring services.
- ▶ We were able to employ inter-service single sign-on (SSO) using a shared session store quite trivially using a Cache Store available in the Bluemix service catalog.
- ▶ We didn't have to manage our own message broker any more, because we had the IBM MQ Light messaging service available in the Bluemix service catalog. In addition, we switched to Advanced Message Queuing Protocol (AMQP) 1.0 protocol that turned out to be a better match for our needs than MQTT, while still being an open standard managed by the same standard body, Oasis:
<https://www.oasis-open.org/>
- ▶ We were able to put into practice the auto-scaling service available in the Bluemix catalog, which enables our microservices to add or remove instances according to demand, reacting to surges while keeping the operating costs down in quiet periods.
- ▶ This time around, we had the benefit of the delivery pipeline provided by the DevOps Services. With the delivery pipeline, we were able to automate our deployment in a much more convenient and flexible way.

In the end, we are glad that we did not start with Bluemix, because if we had, we would not be able to appreciate the differences between IaaS and PaaS abstraction levels. Figure 5-7 on page 103 depicts a detail of the Bluemix console showing microservices powering the console (dev space).

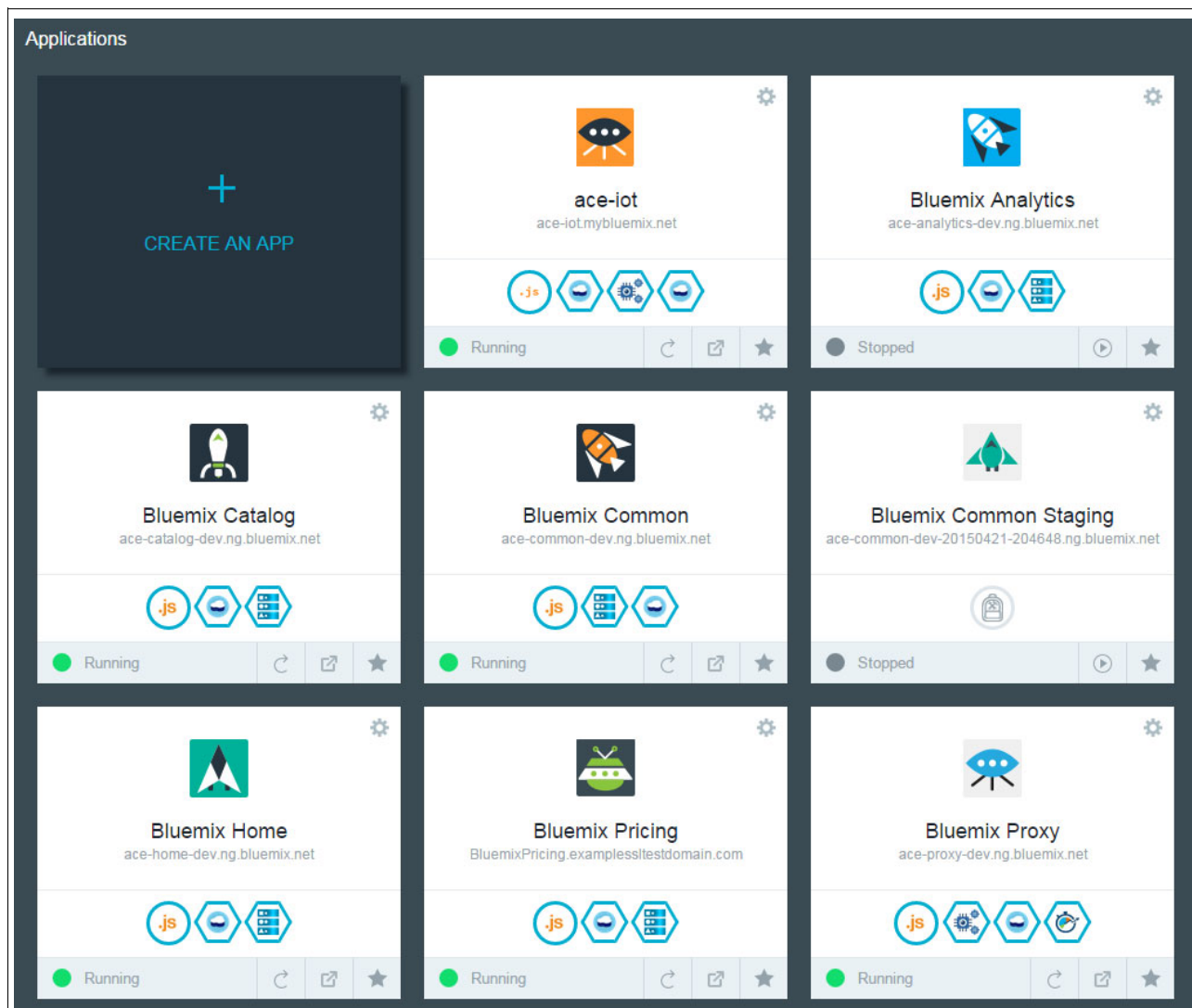


Figure 5-7 A detail of the Bluemix console showing microservices powering the console (dev space): A great example of self-hosting

5.3 Microservices case study in IBM Watson services

Watson is a cognitive computing system capable of understanding natural language, adapting and learning, generating, and evaluating hypotheses to interactively answer questions posted in natural languages. Watson Developer Cloud (WDC) is a collection of REST and software development kits (SDKs) that uses cognitive computing to solve business problems by radically expanding access to IBM Watson services to services consumers.

IBM has been applying microservice architecture methodology into the process of boarding IBM Watson services into the cloud environment. In this section, we describe some major components of WDC, and how microservices architecture has been used to rapidly and effectively make Watson services available in a cloud environment, specifically the IBM Bluemix platform.

5.3.1 IBM Watson Developer Cloud

Watson Developer Cloud is a collection of APIs and SDKs that use cognitive computing to help solving business problems. Figure 5-8 shows the platform architecture of IBM WDC.

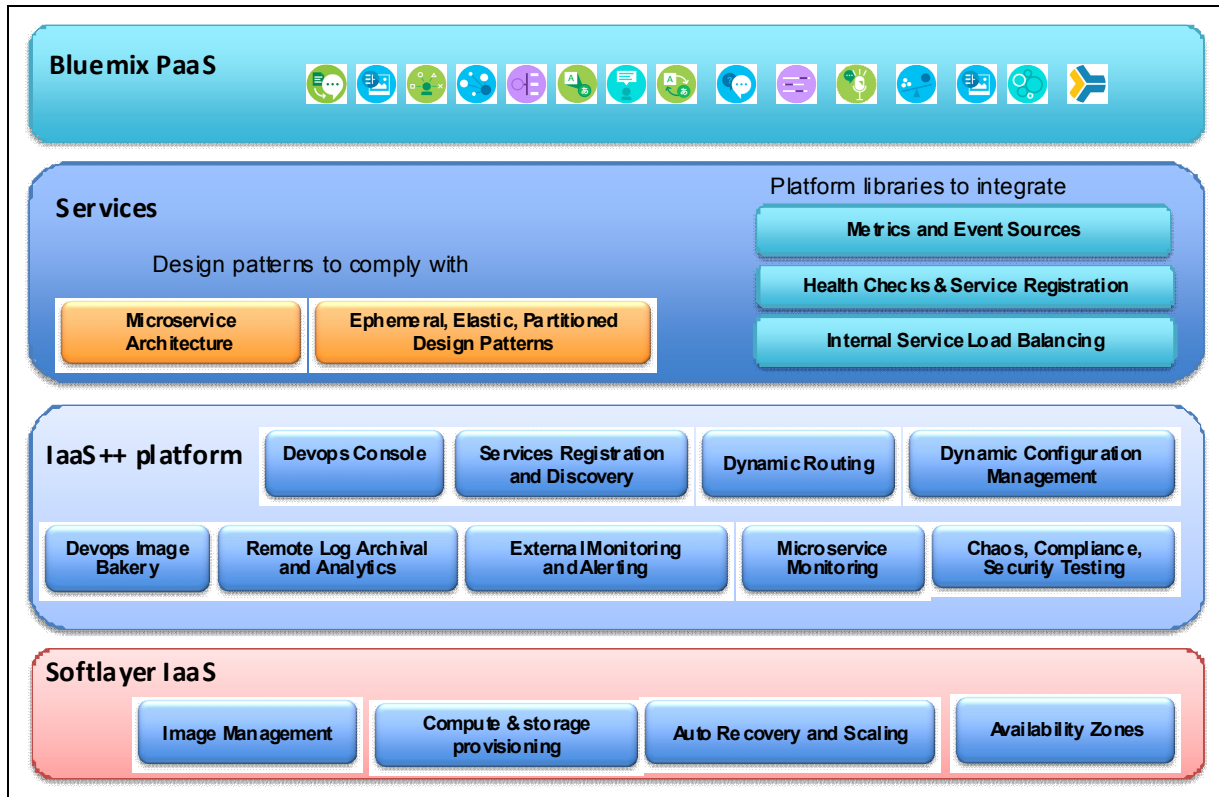


Figure 5-8 IBM Watson Developer Cloud platform architecture

WDC platform is composed of three major layers:

- ▶ SoftLayer infrastructure as a service (IaaS)
- ▶ IaaS++ platform (Cloud Services Fabric, also known as CSF)
- ▶ Services layer

SoftLayer infrastructure as a service

SoftLayer is a cloud IaaS offering from IBM that provides high-performance cloud infrastructure, such as servers, storage, network, and so on. SoftLayer comes with a set of powerful tools to manage and control the cloud computing resources in a self-service manner.

More information about IBM SoftLayer can be found on the following website:

<http://www.SoftLayer.com>

WDC is deployed on SoftLayer infrastructure, and uses several of its APIs and utilities to operate. The following list includes some of these utilities:

- ▶ Image management
- ▶ Compute and Storage provisioning
- ▶ Auto recovery, auto scaling
- ▶ Availability zones

IaaS++ (Cloud Services Fabric)

IaaS++ is a cloud operating environment that IBM developed by leveraging industry-standard, open source software.

More details about the IaaS++ can be found in 5.3.2, “IaaS++ platform (Cloud Services Fabric)” on page 105.

Services layer

On top of the CSF layer, the Watson innovation team continues leveraging the open source software for services-level orchestration:

- ▶ Client-side load balancing, such as Ribbon and HAProxy
- ▶ Services registration, such as Eureka, Etcd
- ▶ Uptime for service health checks

Microservices architecture and relevant cloud design patterns have been applying extensively, such as circuit breaker, event sourcing, and so on.

5.3.2 IaaS++ platform (Cloud Services Fabric)

IaaS++ is a cloud operating environment developed by IBM based on open source technologies. IaaS++ provides functions, such as service registry and discovery, dynamic routing, operational visibility, dynamic configuration management, logging management, centralized monitoring stack, and so on. Most of the components came from industry-standard open source software that IBM adapted to run on SoftLayer infrastructure. The diagram in Figure 5-9 shows the current architecture overview of IaaS++.

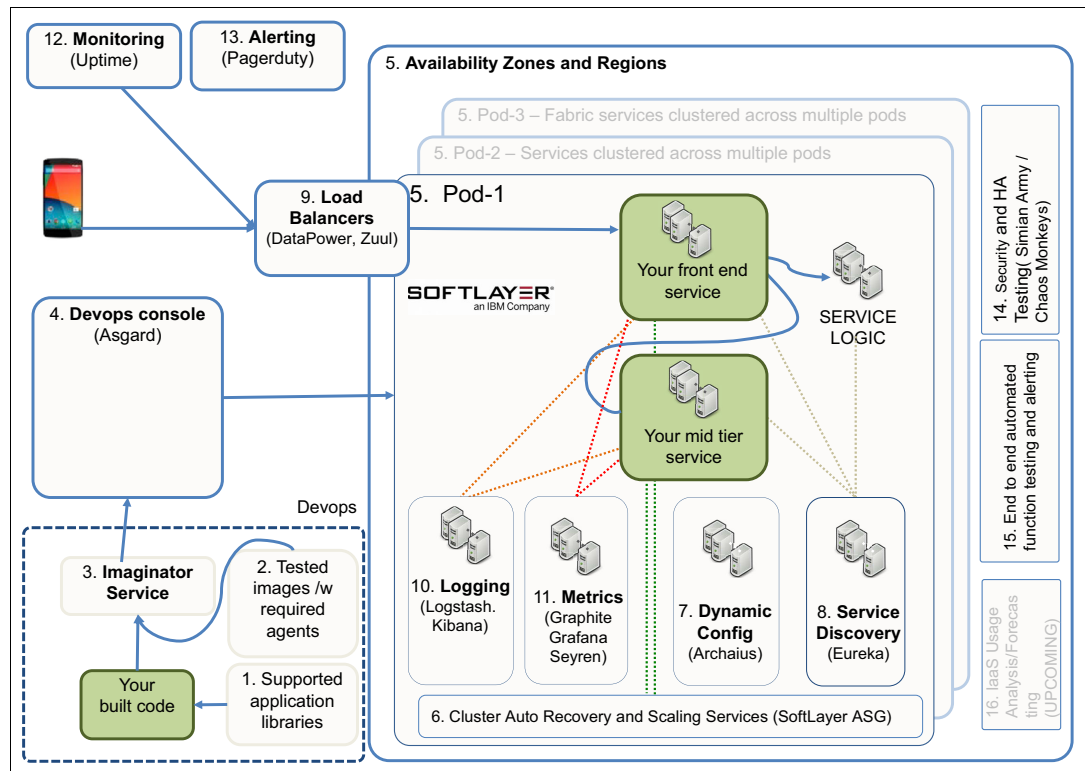


Figure 5-9 IBM IaaS++ architecture

Important: IaaS++ is a cloud operating environment built by IBM and used as an initial internal offering. There are plans to make this available in external offerings as appropriate.

Next, we describe in detail each component that makes up IaaS++ in the following sections.

5.3.3 Main components of IaaS++

Some components in IaaS++ originally came from industry-standard open source ecosystems, which run on Amazon cloud infrastructure. The IBM team ports them to adapt the components into a SoftLayer infrastructure, and develops new components to make up the IaaS++. The following list provides details about each component and its responsibility in the journey of moving to microservice architecture:

- ▶ DevOps console

DevOps console enables you to implement the continuous DevOps delivery mechanism. This console enables you to visually deploy multiple service instances by adding them into auto scaling group (ASG), and then performing a patch deployment. The tool also provides the ability to configure for high availability (HA) and instance recovery, and to define auto-scaling policies as needed.

- ▶ DevOps image bakery

Image bakery is used as a DevOps image *bakery* for immutable approach, to make sure that all of the changes to any services must be made available through a DevOps process, which does not allow any direct access to the server side to make changes, but through an image. The service development team is provided with a base image with necessary run time settings and hardened implemented policies.

To this, they add their code, which they developed, and all of its dependencies. The bakery packages the code and provisions a deployable image, which is basically a SoftLayer CloudLayer® Computing Instance (CCI) image for deployment through a DevOps console.

- ▶ Service registry and discovery

In the monolithic approach, services start one another through language-level mechanism or procedure calls. Services in a traditional distributed system deployment are easily located using hosts and ports, and consequently it's easy for them to call one another using HTTP/REST or some Remote Procedure Call (RPC) mechanism.

In microservice-style architecture, it is critical to make the interactions between services reliable and failure-tolerant. Services fundamentally need to call one another. Microservices typically run in containerized environments, where the number of instances of a service, and its location, dynamically changes. That leads to the requirement to implement a mechanism that enables the consumers of a service to make requests to a dynamically changing set of ephemeral service instances.

Service registry is primarily used in IaaS++ for locating services to fulfill that requirement. As a service registry, it is used with other components for the purpose of load balancing, failover, and various other purposes, such as checking the health state of each service, the ability to take service instances in or out without taking them down, and so on.

- ▶ ELK stack

Elasticsearch-Logstash-Kibana (ELK) works as a centralized logging infrastructure. *Elasticsearch* is a REST-based distributed search engine that is built on top of Apache Lucene library, and is where all of our logs will be stored. *Logstash* is a tool to collect log events, parse them, and store them for later use. *Kibana* is a web-based analytics and search interface that enables you to search and display data stored by Logstash in Elasticsearch.

- ▶ **Dynamic configuration management**

In the journey of applying DevOps methodology into microservice architecture implementation, it's fundamental to enable the ability to effect changes in the behavior of any deployed services dynamically at run time. Availability is also of the utmost importance, so you need to accomplish the changes affected without any server downtime.

Also, you want the ability to dynamically change properties, logic, and behavior of a service based on a request or deployment context. For example, the behavior changes from development to the production deployment environment. This framework supports a hierarchical property structure, and overrides by environment context (for example, development, staging, and production). Apache Zookeeper is used as the distributed data store behind this configuration management.

- ▶ **Dynamic routing**

This component is an edge service application and used in IaaS++ to enable dynamic routing, monitoring, resiliency, and security. It is in the front tier for all consuming requests to CSF to dynamically route HTTP requests and response based on deployed filters. The filters can be pre-routing, routing, or post-routing filters, and are written in Groovy and dynamically compiled into a running server. Also, this component can route requests to multiple ASGs.

- ▶ **Operational visibility**

Operational visibility with real-time insights about a system enables us to deeply understand how the system works in order to proactively detect and communicate system faults, and identify areas of improvement. IaaS++ uses several open source solutions to fulfill that requirement, such as Graphite for time series data store, Hystrix for circuit breaker dashboard, and Seyren and PagerDuty for alert threshold checking and triggering.

- ▶ **High availability testing**

HA testing against a system is a mechanism for identifying high availability weaknesses in the system in order to ensure the ability to automatically recover from failures of that system in real life operation. In the microservice world, where the complexity is typical and fault tolerance is a critical requirement, it is fundamentally important to identify the weaknesses. Watson team has been applying several open source approaches to perform the test. Some of them are Chaos monkey, Security monkey, Latency monkey, and so on.

5.3.4 IBM Watson Developer Cloud services

Watson Developer Cloud (WDC) services is a set of services deployed on IBM Bluemix platform for developers to use to develop Watson-powered solutions on Bluemix. This section describes how the IBM Watson Innovation team makes WDC services available by using a microservices architecture approach, powering IBM Watson and Bluemix.

Figure 5-10 shows the Watson services that are currently available in a Bluemix platform.

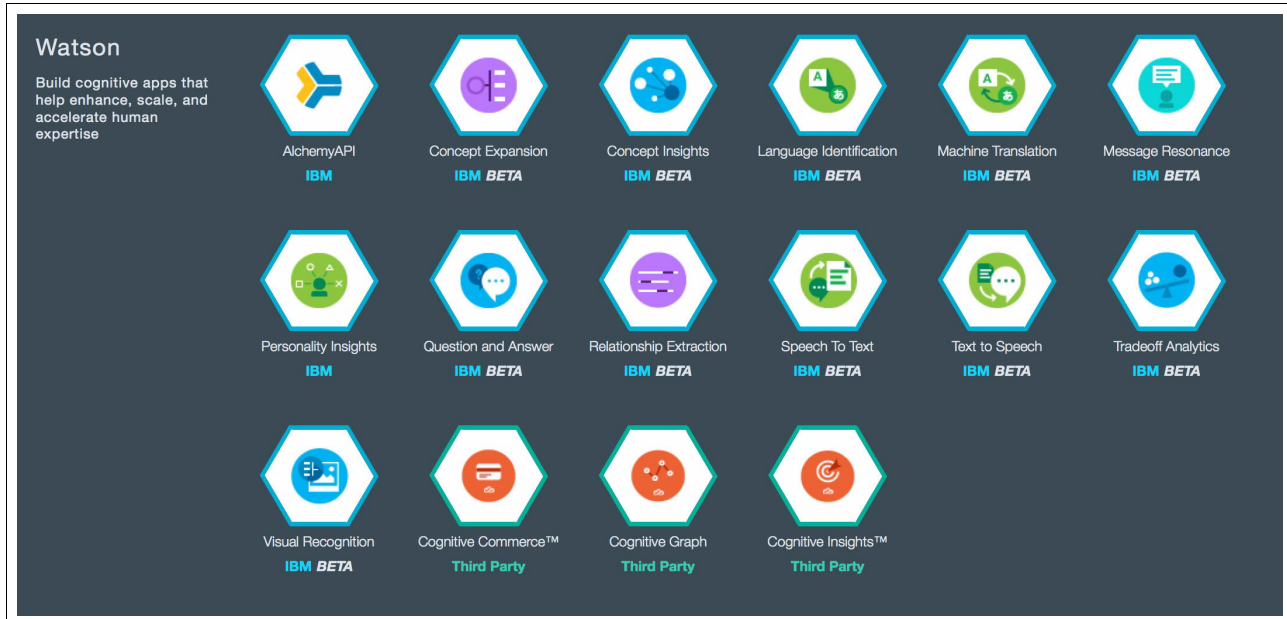


Figure 5-10 Watson services in IBM Bluemix

There are currently 13 Watson services available in Bluemix, most of which are brought and orchestrated to the cloud environment through a process that's centered around a microservice architecture approach.

Migrating to microservice architecture

Some of Watson services were originally developed and made available based on a monolithic approach, so they have a REST interface for consumers to interact with. However, most of the back-end services are built on a monolithic approach, and use a traditional scale-up model.

The approach that has been taken for those services is to not apply a microservice architecture approach from the beginning. The phased approach was used instead, to board all services that were operating into the SoftLayer cloud infrastructure, even with a bare metal deployment model, and ensure that they are working well on the new cloud infrastructure.

After they have been boarded and started serving the consumers well, the Watson innovation team started the next phase to create a more distributed computing architecture. For example, the bare metal servers are moved to a SoftLayer VM instance. Some IBM DB2 databases are moved to *CSF Cassandra* as a service, and some stand-alone ActiveMQs are moved to *CSF Kafka message broker*.

The next phase is to start breaking up the dependencies that the services have, which services are mostly built in a monolithic fashion, and then migrate them into a more microservices architecture approach.

Decomposition to microservices

Overall, the system has been partitioned to several building stacks, and then split up to smaller services using the *single responsibility* principle (where each service has a small set of responsibilities). The following list provides some examples:

- ▶ Logging stack contains loosely coupled services, such as ELK, which internally interact with each other to perform logging activities of the whole system. This is a centralized logging framework, so that we no longer need to `ssh` onto individual nodes and inspect each log. With Kibana, users can query and filter against all nodes and see log results from a single Kibana dashboard.
- ▶ Monitoring stack contains a bunch of microservices, which choreographically interact with each other to visually gain insights about the health status of all Watson services in a real-time manner that consequently reduces the time needed to discover and recover from a bad operational event. *Grafana* is used as a centralized monitoring dashboard to view metrics for all nodes in the system.
- ▶ *Eureka* service is being used as a service registry where all other services self-identify and register themselves as a specific service, and a service discovery responding to queries about location of a specific service instance.
- ▶ In order for services to onboard onto IaaS++, they need to integrate with the *Common Service Broker* microservice for provision, bind, and catalog requests from Bluemix.
- ▶ *Authentication* service is responsible for authenticating requests coming from a Bluemix application into Watson developer platform.

Communication and load balancing

For external Bluemix apps, traffic to WDC goes through IBM DataPower®, routed to Dynamic Router, and then reaches the appropriate Watson service using a client-side load balancer. This load balancer is a client for all outbound requests from the router, which provides detailed information into network performance and errors, and handles service load balancing and load distribution.

For interprocess communication, WDC uses both synchronous HTTP-based messaging, such as REST, and asynchronous messaging mechanism, such as *Apache Kafka*. IaaS++ components can discover services as needed using the service registry, as a service registry (for example, client side load balances based on the healthy instances it receives from the service registry during service discovery).

One example flow in detail is when there is a binding request from a Bluemix application to a Watson service on Bluemix, the Common Service Broker is started at the service bind time. This invocation sets up unique credentials tied to a specific service instance for Bluemix. The user uses these credentials on every request that is made to the Watson Services. The initial request reaches the DataPower gateway, which routes all of the traffic to *Zuul*.

Zuul uses Eureka as the service registry, and client-side load balancing routes the traffic to the necessary service. Zuul supports pre-filtering for pre-processing, routing filtering for routing, and post-filtering for post-processing. These features are written in *Groovy*, and can be dynamically added or updated at run time.

Scaling and service recovery for high availability

IaaS++ uses the SoftLayer autoscale API and offers the auto scaling ability through the DevOps console. The user can create an ASG, and add deployable SoftLayer CCIs (which contains the ready code and all dependencies, run on top of an Ubuntu image). The user then defines the scaling attributes, such as min, max, and wanted number of instances on the Asgard console, in addition to auto scaling policies if needed. Next, the DevOps stack in IaaS++ handles the auto scaling based on the resource consuming status.

For service recovery, each service instance runs an agent (*Karyon* to self register to the Eureka service directory, and *Eureka client* with a caching mechanism to locate a service instance and keep track of its healthy status). *Asgard* is responsible for calling APIs on the SoftLayer autoscale, to create clusters with the wanted number of instances. If any instance stopped working (stopped “heart beating”), the auto-scaler created a new instance and place into the cluster.

The client-side load balancer gets information about a target service from the service registry, and routes and load balances the request to the appropriate service instance in the target service cluster. Figure 5-11 shows an example in detail of the interaction mechanism between services through service registry and discovery.

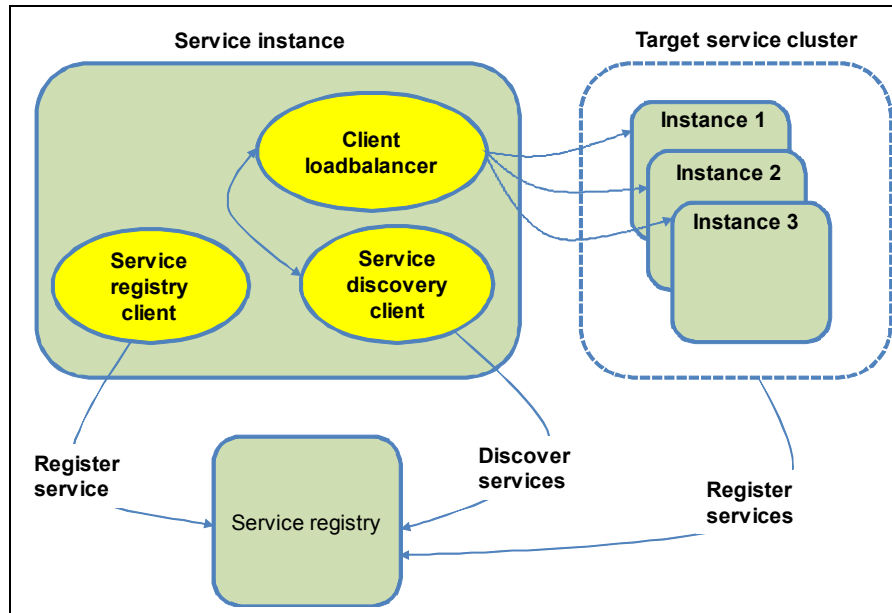


Figure 5-11 Service register and discovery

Deployment

To protect anybody from going directly into a server to modify anything, the immutable approach has been used, which means that the deployment needs to start from an image. The service development team is provided with an Ubuntu base image with the necessary setting to harden the security and other runtime environments.

The development team has to bring in a Debian package, which contains the code and all dependencies that the code that runs on top of the base image might have. The image baker then combines those parts, provisions a deployable image, and provides a SoftLayer CCI image for deployment through the DevOps console.

From a deployment perspective, IaaS++ leverages the *Canary red/black deployment mechanism*. Deployable code is placed in a new ASG alongside the existing one, and the traffic to the services starts being ported to the new ASG after the code is pushed out. This approach helps to build an automated confidence level about the readiness of the new code, and provides the ability to quickly switch back to the old version if needed, which ensures that there are no destructive or in-place upgrades.

Figure 5-12 shows the process of how to onboard a Watson service to a cloud environment, along with the major IaaS++ components' interaction.

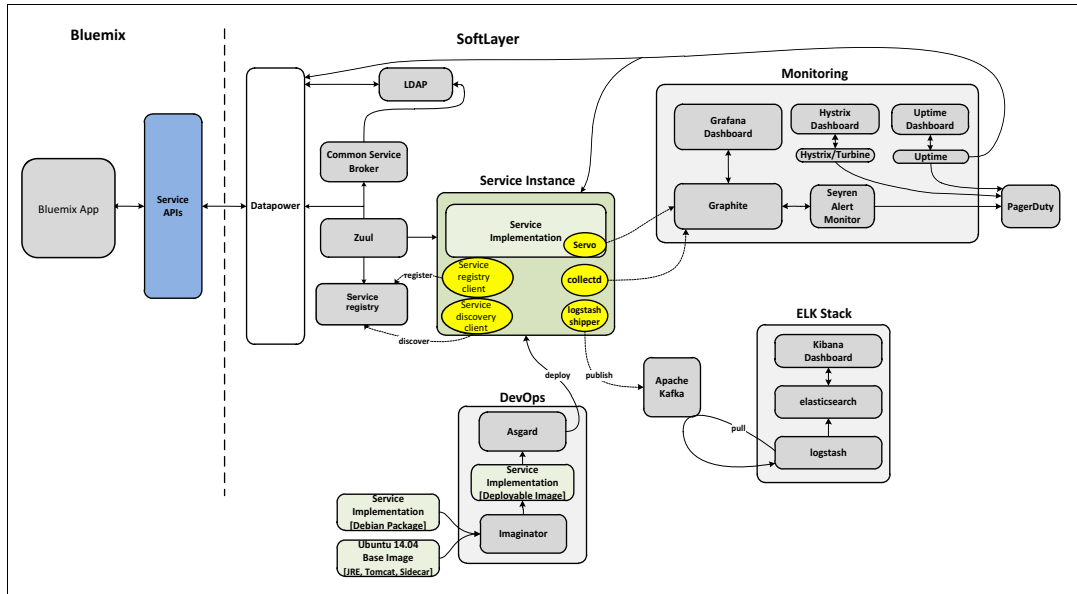


Figure 5-12 Onboarding a Watson service on to cloud environment

Example scenarios using the microservices approach

In this part, we describe three example scenarios that were developed using the microservices approach. The following scenarios are described:

- ▶ Chapter 6, “Scenario 1: Transforming a monolithic application to use microservices (CloudTrader)” on page 115
- ▶ Chapter 7, “Scenario 2: Microservices built on Bluemix” on page 123
- ▶ Chapter 8, “Scenario 3: Modifying the Acme Air application and adding fault tolerance capabilities” on page 131



Scenario 1: Transforming a monolithic application to use microservices (CloudTrader)

In this chapter, we demonstrate how to bring code from IBM DevOps Services into your Eclipse workspace, build and deploy the application to IBM Bluemix, and use database and other services in the cloud. We also externalize a portion of the business logic from a monolithic application into a new microservice developed entirely in Bluemix. To demonstrate this process, we work with the CloudTrader sample application.

This scenario exemplifies a typical modernization of an existing application by selecting a relatively small component or piece of functionality, isolating it, and provisioning a new component in a language and platform more conducive to constant changes. To increase the agility of delivering new features, we set up a Bluemix Delivery Pipeline for the new microservice.

This chapter has the following sections:

- ▶ 6.1, “Introduction to the sample” on page 116
- ▶ 6.2, “Refactoring the application” on page 117
- ▶ 6.3, “The CloudTraderAccountMSA microservice” on page 118
- ▶ 6.4, “The CloudTraderQuoteMSA microservice” on page 118

6.1 Introduction to the sample

CloudTrader is a Java Platform, Enterprise Edition (Java EE) application that simulates an online stock trading system. This application enables users to log in, view their portfolio, look up stock quotes, and buy or sell shares. It is built primarily with Java servlets, JavaServer Pages (JSP), and JavaBeans. To showcase migrating existing applications to Bluemix and revitalizing them with microservices, CloudTrader was created by making changes to a ten-year-old day trading application. Figure 6-1 shows the CloudTrader user interface.

The screenshot displays the CloudTrader user interface. At the top, there is a navigation bar with tabs for Overview, Trading & Portfolio, Configuration, and FAQ. Below this, there are sub-tabs for Home, Account, Portfolio, Quotes/Trade, and Logoff. The main content area shows a welcome message for user 'uid:0' and the current time 'Fri Apr 24 15:32:10 UTC 2015'. The interface is divided into several sections: 'User Statistics' (account ID: 1000, account created: 2015-04-22 01:31:13.346, total logins: 5, session created: Fri Apr 24 15:32:10 UTC 2015), 'Account Summary' (cash balance: \$ 998382.05, number of holdings: 1, total of holdings: \$ 2093.00, sum of cash/holdings: \$ 1000475.05, opening balance: \$ 1000000.00, current gain/(loss): \$ 475.05 (+0.00%)), and 'Market Summary' (Trade Stock Index (TSIA) 86.97 (+0.00%)). There is also a 'Your Friends' section with two entries: 'Bought IBM @ 184.61' and 'Should I keep YRD??'.

Figure 6-1 CloudTrader user interface

The CloudTrader application can use the following Bluemix services:

- ▶ **SQLDB (required).** CloudTrader uses the SQLDB on-demand relational database service powered by IBM DB2 as its main database. The database is created and available for CloudTrader to use in seconds. The data source connection information is created as a Java Naming and Directory Interface (JNDI) namespace entry, ready for the application to look up and start using.
- ▶ **Session Cache.** When bound to CloudTrader, this service enables persistence for all sessions across multiple instances of the application. If there is a failure, sessions are seamlessly recovered.
- ▶ **Auto-Scaling.** Auto-Scaling enables you to automatically increase or decrease the computing capacity of your application. The number of application instances are adjusted dynamically based on the Auto-Scaling policy you define.

Figure 6-2 shows the CloudTrader components overview.

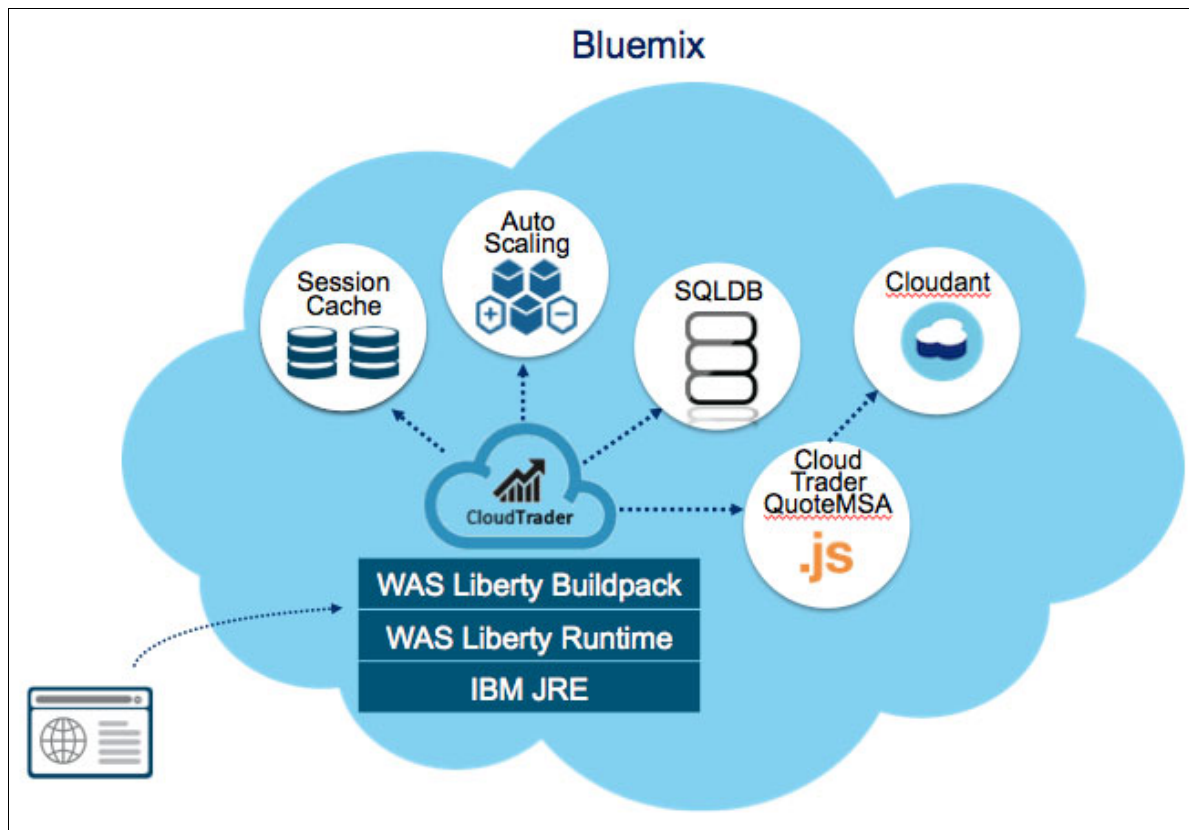


Figure 6-2 CloudTrader components overview

The steps involved in deploying the original CloudTrader to Bluemix, including the source code, can be found on the following website:

<http://www.ibm.com/developerworks/cloud/library/cl-cloudtrader-app/index.html>

6.2 Refactoring the application

The bulk of the business operations provided by the application is in the class TradeDirect, which implements the interface TradeServices. This is a typical monolithic component, because this *single class* in this *single application* is responsible for the main entities used in a stock trading scenario:

- ▶ Orders
- ▶ Quotes
- ▶ Accounts
- ▶ Holdings

In this scenario, we externalize the related business logic of two components from the previous list to demonstrate how a monolithic application can be modernized and refactored in small increments. The quotes and accounts related logic is coded into a Quote microservice and an Account microservice.

The same approach we use for these two business functions can be used for the other entities listed.

In a typical scenario faced by actual organizations, we chose to maintain the accounts microservice in the same original language and platform (Java on IBM WebSphere Application Server Liberty). That choice was made because it is likely that the organization maintaining this application already has a wealth of Java and WebSphere Application Server skilled resources. The advantage of externalizing the smaller scope business function is achieved by the gain in agility afforded by a microservice with reduced scope and accelerated testing and release cycles.

We chose to write the new Quote microservice in Node.js, to exemplify the flexibility of Bluemix, which enables an organization to source each of the microservices using the languages best suited for that particular microservice, still running in a single platform. Figure 6-3 shows the applications involved in this scenario.

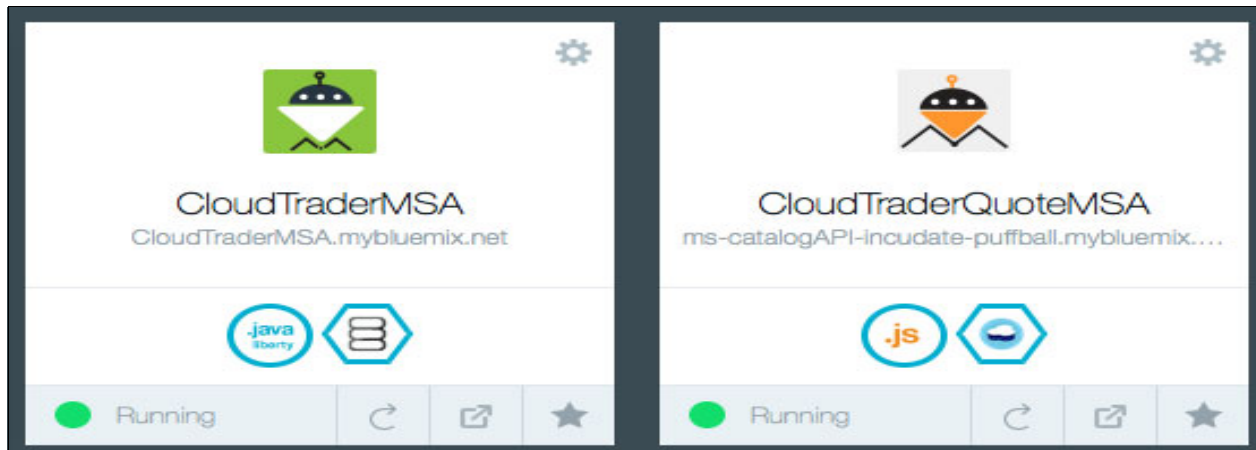


Figure 6-3 Applications involved in this scenario

6.3 The CloudTraderAccountMSA microservice

The code for this microservice was extracted from the original CloudTrader application but, for simplicity, we leveraged the same data store used by the CloudTraderMSA main application. This is accomplished by binding the existing TradeDataSource SQLDB service to the new CloudTraderAccountMSA.

The code for the CloudTraderAccountMSA is maintained in Eclipse, and we use the Eclipse Tools for Bluemix to publish the code to the Bluemix platform.

6.4 The CloudTraderQuoteMSA microservice

The sample code is available as a Git project on IBM DevOps Services. The CloudTraderQuoteMSA microservice requires an IBM Cloudant service instance to function. The Cloudant repository is initialized with a few quote items when it gets instantiated.

For simplicity, the CloudTraderQuoteMSA microservice receives Representational State Transfer (RESTful) requests from the CloudTraderMSA modified monolith, and returns the requested quote.

Note: In a real life scenario, the CloudTraderQuoteMSA microservice likely forwards a request to an external stock quoting system.

Figure 6-4 shows the CloudTrader components overview with an externalized Quote microservice.

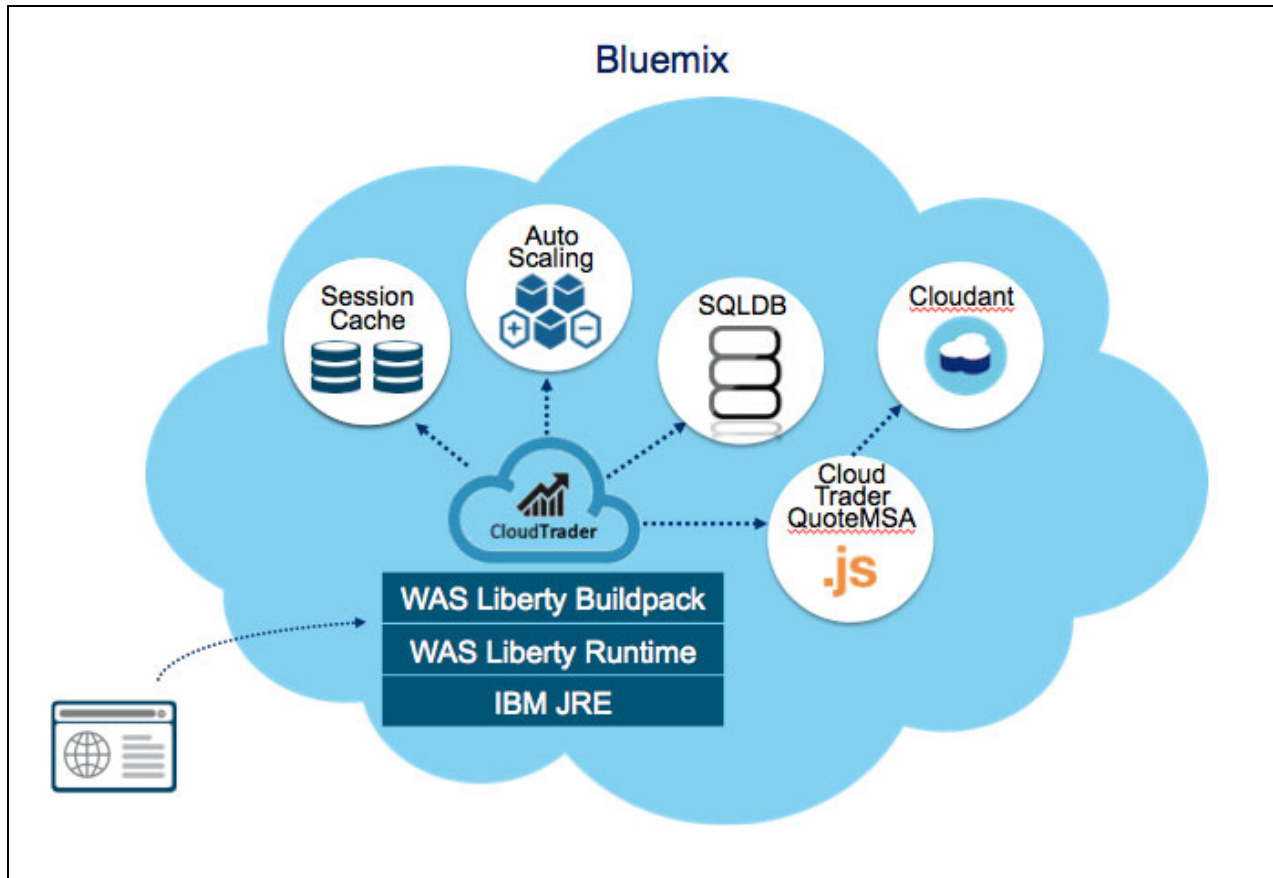


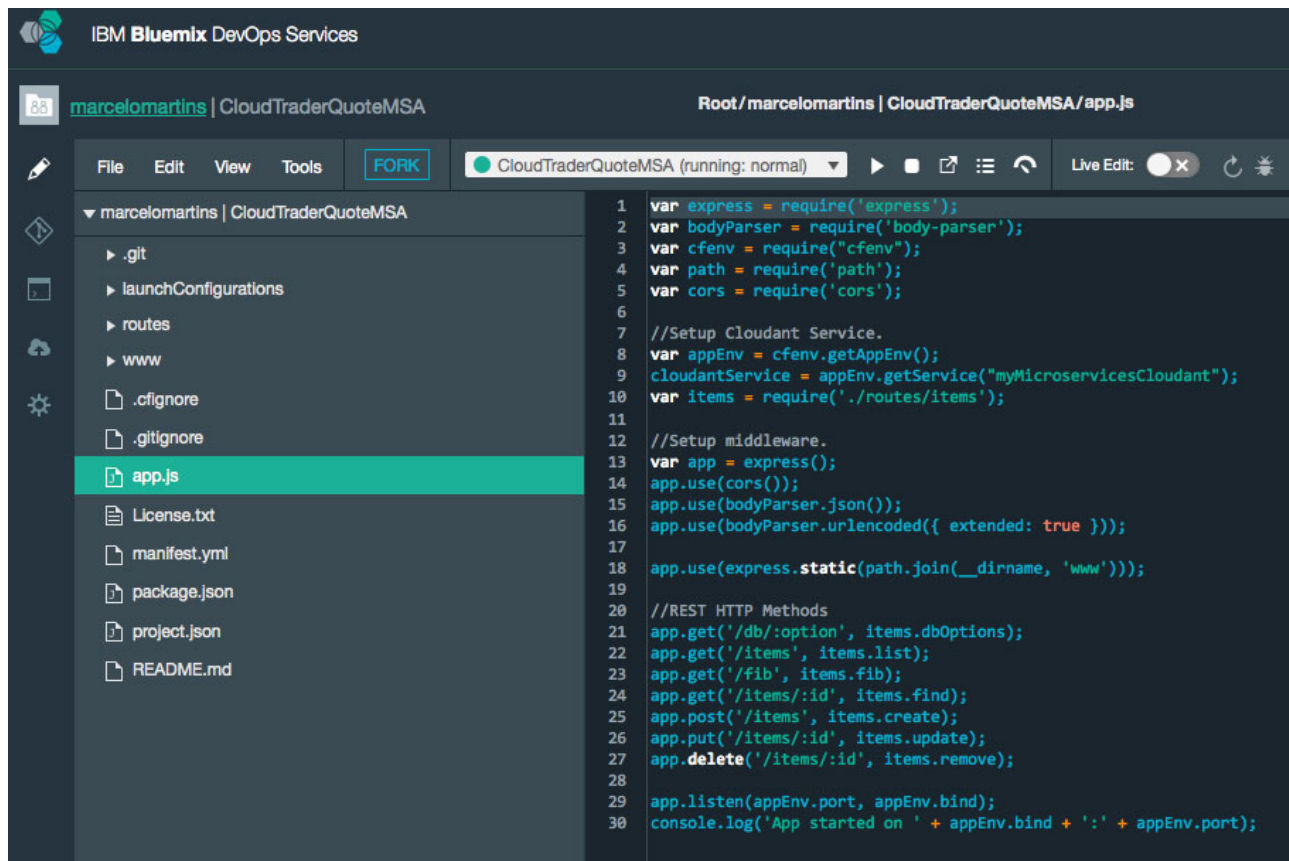
Figure 6-4 CloudTrader components overview with externalized Quote microservice

6.4.1 DevOps Services

Bluemix DevOps Services provides a software as a service solution on the cloud to support continuous delivery. We wrote the CloudTraderQuoteMSA code directly in the browser using the built-in web integrated development environment (IDE), and set up a pipeline that sets up stages to build, test, and deploy to Bluemix. See Figure 6-5 on page 120.

You might have a build stage where your code is built and unit tests are run. Alternatively, you might have a stage that deploys your application, and then run functional tests.

Figure 6-5 shows code being edited in Bluemix DevOps Services.



The screenshot displays the IBM Bluemix DevOps Services interface. At the top, the user 'marcelomartins' is logged in, and the current project is 'CloudTraderQuoteMSA'. The file path is 'Root/marcelomartins | CloudTraderQuoteMSA/app.js'. The interface includes a menu bar with 'File', 'Edit', 'View', and 'Tools', along with a 'FORK' button. Below the menu is a toolbar with various icons for running, stopping, and refreshing the application. The main area is split into a file explorer on the left and a code editor on the right. The file explorer shows the project structure with folders like '.git', 'launchConfigurations', 'routes', and 'www', and files like '.cfignore', '.gitignore', 'app.js', 'License.txt', 'manifest.yml', 'package.json', 'project.json', and 'README.md'. The 'app.js' file is selected and highlighted. The code editor shows the following JavaScript code:

```
1 var express = require('express');
2 var bodyParser = require('body-parser');
3 var cfenv = require('cfenv');
4 var path = require('path');
5 var cors = require('cors');
6
7 //Setup Cloudant Service.
8 var appEnv = cfenv.getAppEnv();
9 cloudantService = appEnv.getService("myMicroservicesCloudant");
10 var items = require('./routes/items');
11
12 //Setup middleware.
13 var app = express();
14 app.use(cors());
15 app.use(bodyParser.json());
16 app.use(bodyParser.urlencoded({ extended: true }));
17
18 app.use(express.static(path.join(__dirname, 'www')));
19
20 //REST HTTP Methods
21 app.get('/db:option', items.dbOptions);
22 app.get('/items', items.list);
23 app.get('/fib', items.fib);
24 app.get('/items/:id', items.find);
25 app.post('/items', items.create);
26 app.put('/items/:id', items.update);
27 app.delete('/items/:id', items.remove);
28
29 app.listen(appEnv.port, appEnv.bind);
30 console.log('App started on ' + appEnv.bind + ':' + appEnv.port);
```

Figure 6-5 CloudTraderQuoteMSA code being edited in Bluemix DevOps Services

To automate the builds and deployments to IBM Bluemix, we use the IBM Continuous Delivery Pipeline for Bluemix (the Delivery Pipeline service). This is shown in Figure 6-6 on page 121. As you develop an app in the cloud, you can choose from several build types. You can provide the build script, and IBM Bluemix DevOps Services runs it; you don't need to set up build systems. Then, with one click, you can automatically deploy your app to one or many Bluemix spaces, public Cloud Foundry servers, or Docker containers on IBM Containers.

Build jobs compile and package your app source code from Git or Jazz source control management (SCM) repositories. The build jobs produce deployable artifacts, such as web archive (WAR) files or Docker containers for IBM Containers. In addition, you can run unit tests within your build automatically. Each time the source code changes, a build is triggered.

A deployment job takes output from a build job and deploys it to either IBM Containers or Cloud Foundry servers, such as Bluemix.

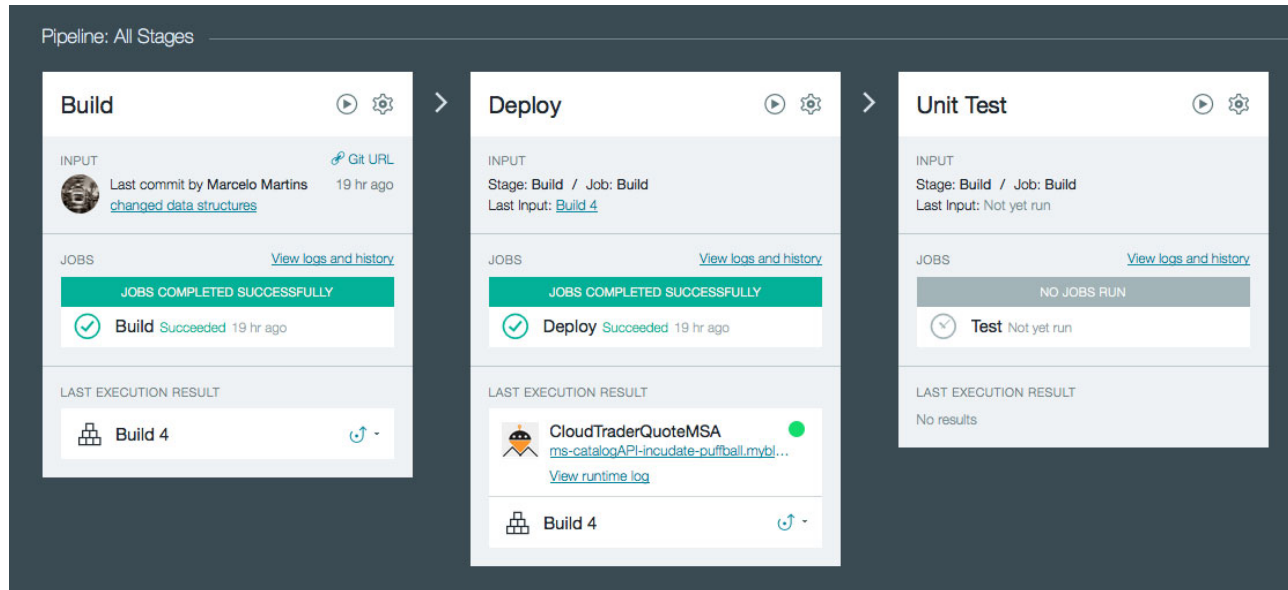


Figure 6-6 Bluemix DevOps pipeline stages for the CloudTraderQuoteMSA microservice

In our sample, we built a simple Bluemix DevOps pipeline with three stages, to build, deploy, and run an automated battery of unit tests. The pipeline can be as complex as necessary, to suit the needs of the testing organization.



Scenario 2: Microservices built on Bluemix

In this chapter we describe a simple microservices architecture online store sample built entirely on IBM Bluemix platform. This chapter introduces a *born on the cloud* microservice application. It demonstrates how to develop microservices, communicate between them, and perform scaling on Bluemix.

This chapter has the following sections:

- ▶ 7.1, “Introduction to the sample” on page 124
- ▶ 7.2, “Breaking it down” on page 124
- ▶ 7.3, “Additional Bluemix services” on page 129

7.1 Introduction to the sample

A modern online e-commerce application has many software requirements that compel us to thoroughly consider the architectural design and plan development accordingly. Users are expecting an interactive experience with the website, and they expect to access it from a wide variety of devices. The application needs to be able to scale on demand so that it can react to dynamic shopping behavior patterns.

Your application might need to roll out constant updates to react to changes in browsers, devices, APIs, security, and so on. Your developers need to be able to deploy updates in minutes or hours, not days. These types of continuous delivery demands require us to design our application differently than in the past. Microservice architecture aims to address this requirement.

In this scenario, we create a mock online store application with some basic functionality to demonstrate microservice concepts. The store is a web application that neatly shows a catalog of items to an online shopper. The shopper is able to pick an item and place an order.

We apply the microservice patterns learned so far in this book to decompose this application into multiple smaller microservice applications. Being a polyglot platform, Bluemix enables you to push your application code written in many different languages, and have it up and running in around a minute. Bluemix enables developers to choose from many different languages and run times. This flexibility ultimately enables us to optimize each microservice app, because we are not restricted to the language or run time of the other microservices.

This gives the application developer the ability to choose the programming language that is best suited for each microservice. For example, the best technology stack for building a slick responsive UI might not be the same stack that is optimal for creating a highly secure, synchronous order processing service that must handle a large volume of transactions.

We cover source control, deployment, scaling, and communication among each of the microservice applications using Bluemix.

7.2 Breaking it down

If we decompose this application using the methods described in this book, we end up with the following discrete microservices:

- ▶ Online Store Catalog. Responsible for keeping track of all the items for sale in the store.
- ▶ Online Store Orders. Processes new customer orders, and provides details about past orders.
- ▶ Online Store Shipping. A worker application that asynchronously receives message triggers when a new order is created, to perform shipping-related processing.
- ▶ Online Store user interface (UI). A web front end for our store, which enables shoppers to browse the catalog and place an order.

This example of the microservices is outlined in Figure 7-1.

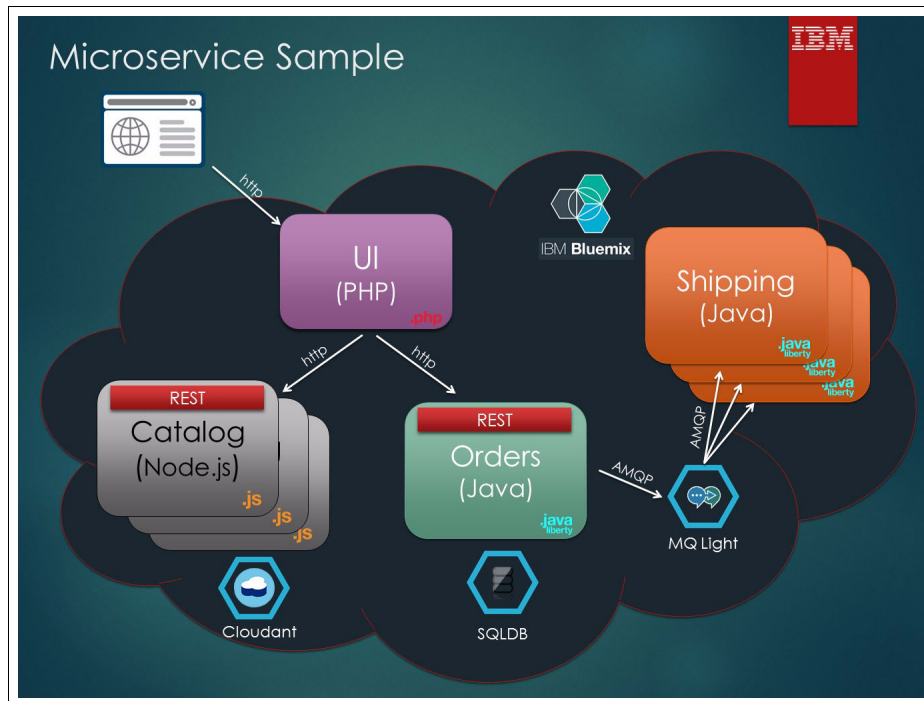


Figure 7-1 Online Store sample

Next, we more closely examine how we implement each of these microservices. These microservices are implemented as independent stand-alone applications in Bluemix. So, we are developing four different applications. Each of the four applications provides different business capabilities, and has been written in a different language. The variability in programming languages demonstrates the options available in Bluemix. The decision about which language to use can be based on several factors to simplify the development tasks, such as the availability of skills or the suitability of the language.

Bluemix offers other types of middleware services, such as database, caching, and monitoring. Our applications can consume one or many of these services. Next, we look at each of the microservice applications in more detail. This variability in language choice does not prevent the microservices from running in a single platform and using common features, such as single sign-on (SSO) or central logging.

7.2.1 Online Store Catalog service

This application's role is simple. It holds information about all of the items we want to sell in our store. It should enable the user to add a new item, delete an item, and return information about items.

Language

We need a language that will help us build a high-throughput, easily scalable application that can handle thousands of simultaneous connections. We won't be performing CPU-intensive operations, but simply a layer to help us interact with a database. Node.js uses a non-blocking, event-driven input/output (I/O) model, and provides an extremely lightweight container that is suitable for what we are doing. The WebSphere Application Server Liberty buildpack on Bluemix offers us an enterprise-grade Java Platform, Enterprise Edition (Java EE) application server, while still being extremely lightweight and dynamic.

Interface

The application needs a way to receive requests for items, and also to provide the ability to modify the items to the catalog. Creating a Representational State Transfer (REST) application programming interface (API) sounds like an optimal choice for this type of interface, because it provides a widely accepted standard for interacting with resources.

Node.js with the Express framework provides a way to create a minimalist web framework for us to create this interface. Express is a module for Node.js that enables you to create a REST API by enabling us to bind functions to incoming requests using Hypertext Transfer Protocol (HTTP) methods. We use the following HTTP methods:

- ▶ GET. Get all items, or one particular item.
- ▶ POST. Create a new item.
- ▶ DELETE. Remove an item.

Database

Obviously, the application needs to be persisting these items into a database. Each item in the store can be represented as a JavaScript Object Notation (JSON) data type. An asynchronous JSON handling NoSQL database matches perfectly with our app, because it provides a flexible, schema-less data model to handle our items. One of the many databases that Bluemix has to offer is Cloudant, a distributed, scalable NoSQL database as a service.

Get it running

Now that we selected the language, framework, and database, we are ready to start writing some code and deploy our application. The sample code is available as a Git project on IBM DevOps Services. The following link outlines detailed steps to deploy this online store application, and provides access to all of the source code:

<https://developer.ibm.com/bluemix/2015/03/16/sample-application-using-microservices-bluemix/>

Before we get into the deployment steps, examine the `manifest.yml` file at the root of the project. A `manifest.yml` file defines the application to Bluemix. It can contain various information about the application, such as the name, URL, amount of memory, and number of instances. It also contains the names of any Bluemix-provided services needed by our app. For example, we decided that this catalog application requires the Cloudant service.

The documentation in the previous link contains a **Deploy To Bluemix** button that facilitates the deployment from source code directly to a running application in Bluemix. The button creates your own private DevOps Services project with a copy of the sample source. It also uses `manifest.yml` to deploy the application to your Bluemix.

7.2.2 Orders

The next application we tackle is the order management application. When a user buys an item from the UI (or any other future channel), the order service is expected to receive and process the order.

Language

Due to the nature of this application handling customer orders and billing information, failures are not acceptable for this service, and security is of utmost importance. We have selected to use Java EE, because this app needs a technology stack that is robust, well-supported, standards-driven, able to provide transactional support, and is industry-proven.

Database

Again, to match our run time, we want to pick a proven, secure, relational database that can handle the transactional workloads. Bluemix offers the SQLDB service, an on-demand relational database service powered by IBM DB2.

Interface

The consumer of this service needs to be able to notify this orders application when a new order is generated, and also provide information about past orders. Similar to the Catalog application in the previous section, we create a RESTful interface using Java EE Java API for RESTful Web Services (JAX-RS). The WebSphere Liberty buildpack on Bluemix supports Enterprise JavaBeans (EJB) applications that are written to the EJB Lite subset of the EJB 3.1 specification, which is ready for immediate use without needing any further configuration.

Get it running

Again, use the following documentation to clone the sample source and deploy to Bluemix. To understand how the application is being deployed, review the `manifest.yml` file:

<https://developer.ibm.com/bluemix/2015/03/16/sample-application-using-microservices-bluemix/>

7.2.3 Shipping

This is our worker application, which waits for new orders and then proceeds to do shipping-related processing. This processing can take an undetermined amount of time to complete. The Orders application needs to asynchronously inform this Shipping application about a new order. However, it does not wait for the Shipping application to complete its work.

We implement the messaging worker offload pattern mentioned in 2.3, “REST API and messaging” on page 33, where this worker uses a messaging service to know when something interesting (a new order) happens, and then proceed to do the work.

Several instances of this worker application could exist, as in this example, which distributes the work among each of the instances. If the load increases, additional worker applications can be created by simply increasing the number of instances, and without the need to alter the application.

Because this application receives its input from a messaging service, and not a web or REST interface, there is not a Uniform Resource Locator (URL) route to access this background worker application.

Messaging

The IBM MQ Light for Bluemix is a cloud-based messaging service that provides flexible and easy-to-use messaging for Bluemix applications. IBM MQ Light provides an administratively light solution to messaging. We use this feature to handle the message communication between the Orders application and this Shipping application.

Language

To make this sample stretch some of the wide run time and language framework capabilities of Bluemix, we decided to implement this worker application using Java EE message-driven Enterprise JavaBeans, which integrates seamlessly with the messaging service. This framework can be easily extended to solve larger business domain problems as needed, due to the self-contained and scalable nature of EJB components.

Get it running

Again, use the following documentation to clone the sample source and deploy to Bluemix. To understand how the application is being deployed, review the `manifest.yml` file:

<https://developer.ibm.com/bluemix/2015/03/16/sample-application-using-microservices-bluemix/>

7.2.4 UI

This is the face of our store. When developing a UI, many aspects, such as JavaScript libraries, usability, globalization, translation, and security, need to be carefully considered. However, for the sake of keeping things simple, this sample has a minimalist web page.

Language

It is no surprise that PHP Hypertext Preprocessor (PHP) is a common language for writing web user interfaces. It gives the developers control over how they want to structure their server-side and client-side code. However, the reason for picking PHP might also simply be that your UI developer is only comfortable with PHP.

Communication with other services

This service needs to call the Online Store Catalog service that we created earlier to get information about all of the items that it needs to render. Because our Catalog service exposes a REST interface, we only need to know the HTTP endpoint of the service. Hard-coding the endpoint URL directly in our code is obviously a bad practice.

If the catalog service route changes, we don't want to have to update our UI application. Rather, Bluemix enables you to provide the service endpoint information to applications using the `VCAP_SERVICES` environment variable, in the same way that you get credentials for services offered by the platform.

First, you create a Bluemix user-provided service for the Catalog API by issuing the following command:

```
cf cups catalogAPI -p "host"
```

This prompts you for the host. Enter the URL for your Catalog API application, for example, `http://ms-catalogAPI-0123abc.mybluemix.net`.

This creates a new service on Bluemix called *catalogAPI*, visible only to your Bluemix org. Applications can now bind to this service and receive the host information in the `VCAP_SERVICES` environment variable. You need to do the same for the `ordersAPI`.

Again, detailed instructions are in the documentation:

<https://developer.ibm.com/bluemix/2015/03/16/sample-application-using-microservices-bluemix/>

Notice that the `manifest.yml` file now references the two user-defined services, `catalogAPI` and `ordersAPI`, that you have just created.

7.3 Additional Bluemix services

Bluemix provides many services that aid in the application development, deployment, and monitoring aspect of a continuous delivery lifecycle.

7.3.1 DevOps Services

Bluemix DevOps Services provide a software as a service (SaaS) solution on the cloud to support continuous delivery. When you click the **Deploy to Bluemix** button, the source code is cloned in a new private project. You can now start making changes to the code directly in the browser using the built-in web integrated development environment (IDE), and establish a pipeline that sets up stages to build, test, and deploy to Bluemix. You might have a build stage where your code is built and unit tests are run. Alternatively, you might have a stage that deploys your application, and then run functional tests.

7.3.2 Monitoring

Bluemix provides the Monitoring and Analytics service, which provides insights about how your application is performing. Availability and performance metrics are crucial when running a business, and having the platform provide this data is extremely valuable. You can bind the Monitoring and Analytics service to an application using the steps provided in the Bluemix documentation. The service can provide dashboards, such as the ones shown in Figure 7-2 and Figure 7-3 on page 130.

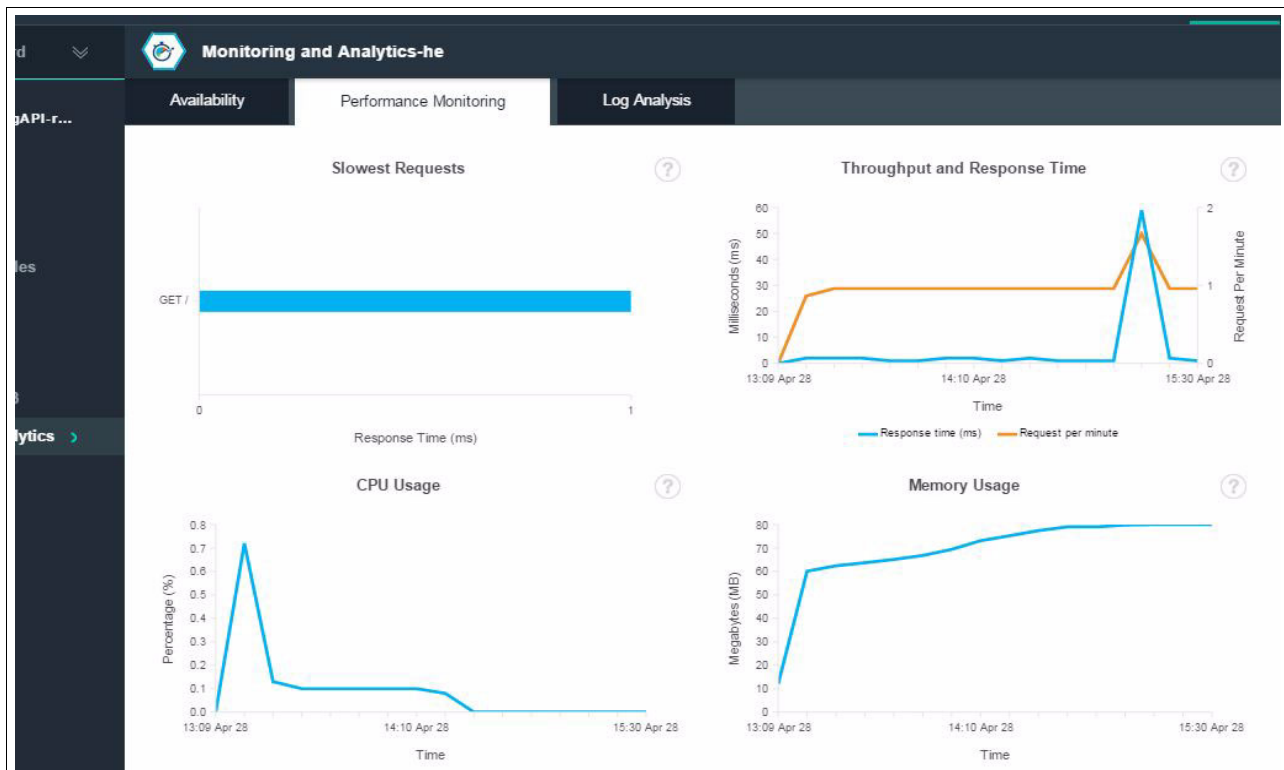


Figure 7-2 Monitoring and Analytics - Performance Monitoring

Figure 7-3 shows the Availability dashboard.

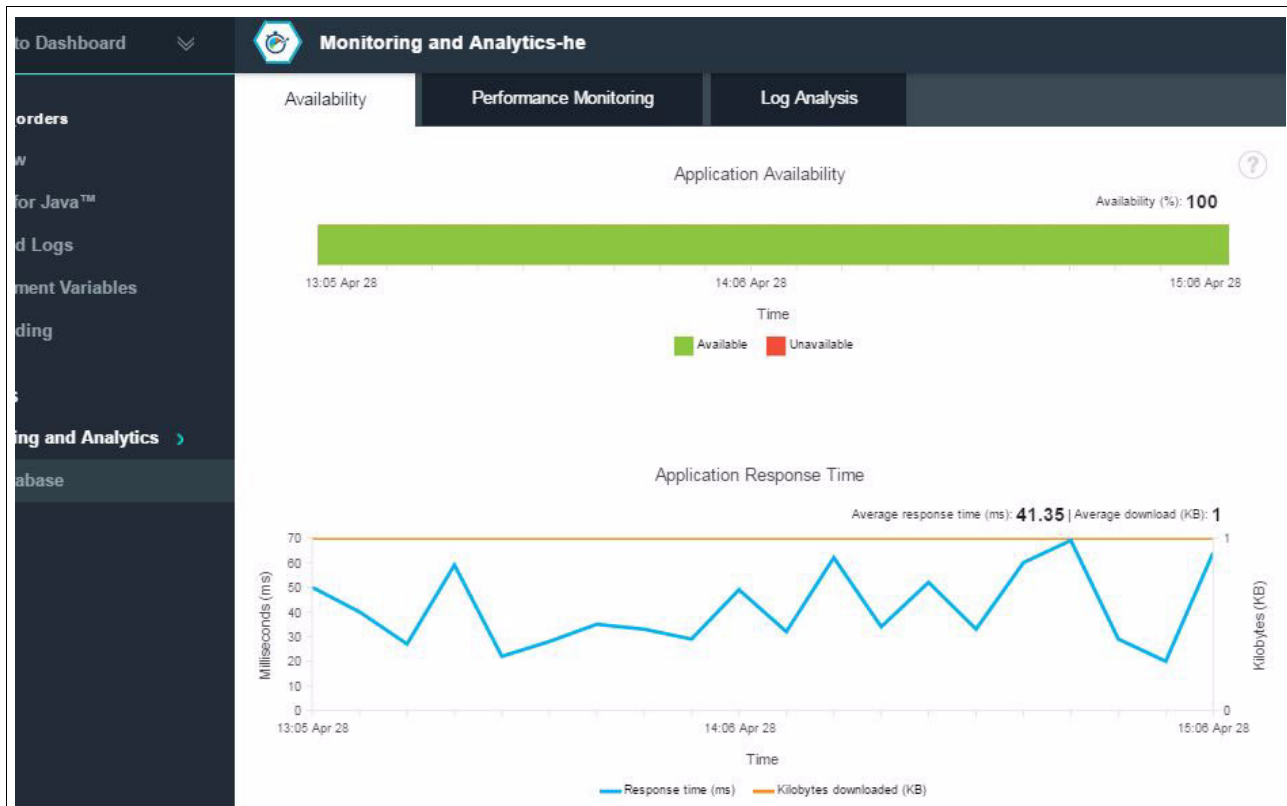


Figure 7-3 Monitoring and Analytics - Availability

If the online store application is running sluggishly, you can see if the root cause is from the response time of the REST endpoints of the Catalog application. From here, you can determine if you need to investigate a performance problem, or scale the application using manual or an auto-scaling policy (horizontal scaling). The memory usage chart can be used to evaluate if more memory needs to be allocated per instance (vertical scaling).

7.3.3 Scaling

One of the great benefits of running your applications on the cloud (when your application is written properly) is the ease of scaling. In our sample scenario, as our store grows in popularity, the UI application makes many requests to our Catalog service. If you add another mobile front-end, that further increases the calls to our service. Bluemix enables you to easily scale the application horizontally by simply increasing the number of instances.

You can increase your instances from one to five and, in about a minute, you should have all five instances serving at the same endpoint behind a load balancer. Each of the instances runs in its own lightweight container, and they do not share the same memory or file system. However, all five instances are bound to the same database service. The scaling of the database service is handled by the provider of the service and the service plan you select.

Bluemix also provides an Auto-Scaling service, which can automatically increase or decrease the instances of your application by the use of the policy that you define. For example, you can protect your application instance from failing due to running out of memory by creating rules to scale up when memory or Java virtual machine (JVM) Heap reaches 80% of capacity.



Scenario 3: Modifying the Acme Air application and adding fault tolerance capabilities

This chapter describes how the functions within an existing monolithic application can be decomposed into finer-grained microservices, and how those individual microservices can be monitored to prevent overall application failure.

Microservices architecture is emerging as the new standard, enabling developers to more easily keep pace with the demand for new application features and reduce maintenance time. In this chapter, we delve further into the Acme Air application that was introduced in 1.5.3, “Large brick-and-mortar retailer” on page 16, and explore how redesigning the features of the Acme Air application to efficiently use microservices provides more flexibility.

As we transform the application from monolithic to a microservices cloud architecture, the application is enhanced with a command pattern and circuit breaker for greater fault tolerance and resilience through fail-fast. We then extend the overall architecture to include Netflix OSS Hystrix Dashboard for monitoring. By adding monitoring such as Hystrix Netflix OSS, we can avoid having the whole application crash when some specific services fail.

This chapter has the following sections:

- ▶ 8.1, “The original Acme Air monolithic application” on page 132
- ▶ 8.2, “Redesigning application to use microservices” on page 139
- ▶ 8.3, “Adding Hystrix to monitor services” on page 142

8.1 The original Acme Air monolithic application

The Acme Air application was designed to represent a fictitious airline that needed the ability to scale to billions of web application programming interface (API) calls per day. The application would handle flights, customer account information, authentication, and baggage service. To accomplish all of this, the application would need to be deployed in a cloud environment, and be able to support user interactions using a normal browser, mobile devices, and business-to-business interfaces.

The application originally had a monolithic design. For the Acme Air application to run on a scale that enables it to service the vast number of customers anticipated for a real airline, it was best for the application to be deployed in a cloud environment.

The application is versatile, such that it can be implemented in various environments. For this scenario, we are deploying in the IBM Bluemix cloud offering. The Bluemix cloud is readily available for deployment, and offers the elasticity to scale the application as needed. This environment enables the application to meet the performance requirements, and be able to expand to meet future demands. The application stores information in a database for customer authentication and flight schedules.

Figure 8-1 shows the architecture when the application is deployed to run in a monolithic mode on the Bluemix cloud environment.

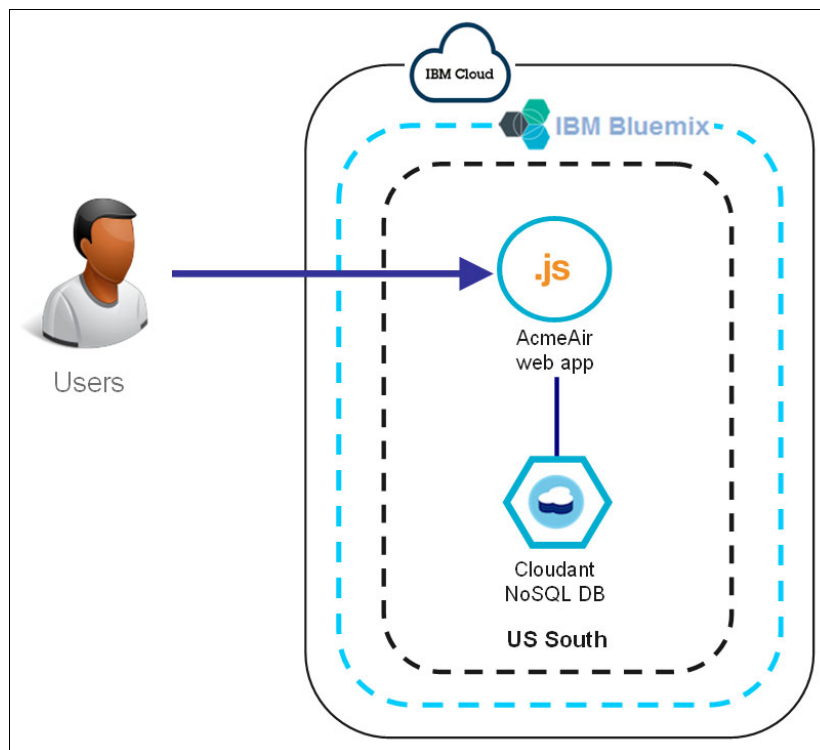


Figure 8-1 Acme Air application architecture for monolithic version

The trend for web API calls has changed in the past few years to include many more mobile users and business-to-business web APIs in addition to the traditional browser-facing clients. The redesign philosophy used for optimizing the Acme Air application was to embrace the different devices that would be used to access the application.

To meet these changing business requirements, a mobile interface feature was added. This feature enables the application to reach users on their remote devices and tie *systems of engagement* to the back-end system. The IBM Worklight server technology (now known as IBM MobileFirst™ platform) was used to provide a secure, scalable, native mobile-facing environment.

8.1.1 Deploying the Acme Air application in monolithic mode

To deploy the Acme Air application to Bluemix, follow the steps described in the following sections.

Download the source code

The source code and information for the monolithic Acme Air application can be downloaded from the following location:

<https://github.com/acmeair/acmeair-nodejs>

Log in to Bluemix

You should have previously signed up for Bluemix at <http://bluemix.net>. You should have also installed the Cloud Foundry command-line interface (CF CLI), which is available on the following website:

<https://github.com/cloudfoundry/cli#downloads>

Log in to Bluemix using the CF CLI. The CF commands to do this are shown in Example 8-1.

Example 8-1 CF commands to login to Bluemix

```
cf api api.ng.bluemix.net
cf login
```

The screen capture in Figure 8-2 shows a **cf login**. You likely have only one organization and one space, and don't need to select these from a list, as is the case in Figure 8-2.

```
C:\Users\IBM_ADMIN>cf api api.ng.bluemix.net
Setting api endpoint to api.ng.bluemix.net...
OK

API endpoint: https://api.ng.bluemix.net (API version: 2.19.0)
Not logged in. Use 'cf login' to log in.

C:\Users\IBM_ADMIN>cf login
API endpoint: https://api.ng.bluemix.net

Email> sdaya@ca.ibm.com
Password>
Authenticating...
OK

Select an org (or press enter to skip):
1. osowski@us.ibm.com
2. bhaskar_bj@yahoo.com
3. shahir.day@gmail.com
4. MobileQualityAssurance
5. sdaya@ca.ibm.com
6. XcelPOC-org

Org> 5
Targeted org sdaya@ca.ibm.com

Select a space (or press enter to skip):
1. CloudIntegration_Demo
2. TimmyMe MSI
3. TwitterInfluencerAnalyzer_Demo
4. CloudTrader_Demo
5. CloudBootcamp
6. Pipeline_Demo
7. Container_Demo
8. Temp
9. AcmeAir
10. Big Data Lab

Space> 9
Targeted space AcmeAir

API endpoint: https://api.ng.bluemix.net (API version: 2.19.0)
User: sdaya@ca.ibm.com
Org: sdaya@ca.ibm.com
Space: AcmeAir

C:\Users\IBM_ADMIN>
```

Figure 8-2 Using the CF command-line interface to login to Bluemix

Push the Acme Air web app

Make sure that you are in the directory that contains the `app.js` file before running the following CF command from a command prompt:

```
cf push acmeair-nodejs-<uid> --no-start -c "node app.js"
```

You should replace `<uid>` with a unique identifier, because this application name will be used for the host name, which must be unique. In this scenario, we used the author's name, for example, `acmeair-nodejs-sdaya`.

The screen capture in Figure 8-3 shows the result of the previous **cf push**.

```
C:\Users\IBM_ADMIN\Documents\0.WIP\0.Code\acmeair-nodejs-public-v2.git>cf push acmeair-nodejs-sdaya --no-start -c "node app.js"
Creating app acmeair-nodejs-sdaya in org sdaya@ca.ibm.com / space AcmeAir as sdaya@ca.ibm.com...
OK

Creating route acmeair-nodejs-sdaya.mybluemix.net...
OK

Binding acmeair-nodejs-sdaya.mybluemix.net to acmeair-nodejs-sdaya...
OK

Uploading acmeair-nodejs-sdaya...
Uploading app files from: C:\Users\IBM_ADMIN\Documents\0.WIP\0.Code\acmeair-nodejs-public-v2.git
Uploading 318.8K, 70 files
OK

C:\Users\IBM_ADMIN\Documents\0.WIP\0.Code\acmeair-nodejs-public-v2.git>
```

Figure 8-3 Pushing the Acme Air web app

In your Bluemix dashboard, you should see the Acme Air web app you just pushed, as shown in Figure 8-4.

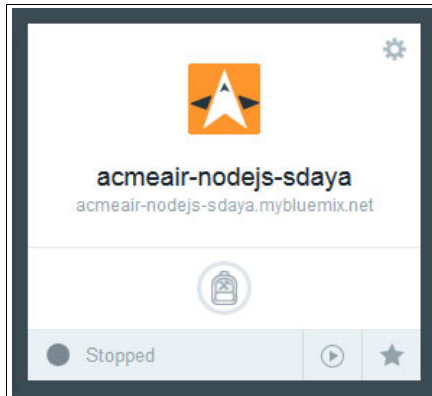


Figure 8-4 Acme Air web app in Bluemix

Note that the web app is not started yet. We need to create a data source for it first, before starting it.

Create a database service

The Acme Air application can work with either MongoDB or IBM Cloudant. The sections that follow describe the steps necessary for each of these.

MongoDB

The following command creates a mongodb service with a name of acmeairMongo:

```
cf create-service mongodb 100 acmeairMongo
```

The screen capture in Figure 8-5 shows the execution of this create-service command.

```
C:\Users\IBM_ADMIN\Documents\0.WIP\0.Code\acmeair-nodejs-public-v2.git>cf bind-service acmeair-nodejs-sdaya acmeairMongo
Binding service acmeairMongo to app acmeair-nodejs-sdaya in org sdaya@ca.ibm.com / space AcmeAir as sdaya@ca.ibm.com...
OK
TIP: Use 'cf restage' to ensure your env variable changes take effect
C:\Users\IBM_ADMIN\Documents\0.WIP\0.Code\acmeair-nodejs-public-v2.git>
```

Figure 8-5 Creating a MongoDB service

In your Bluemix dashboard, you should see the MongoDB service that you just created, as shown in Figure 8-6.

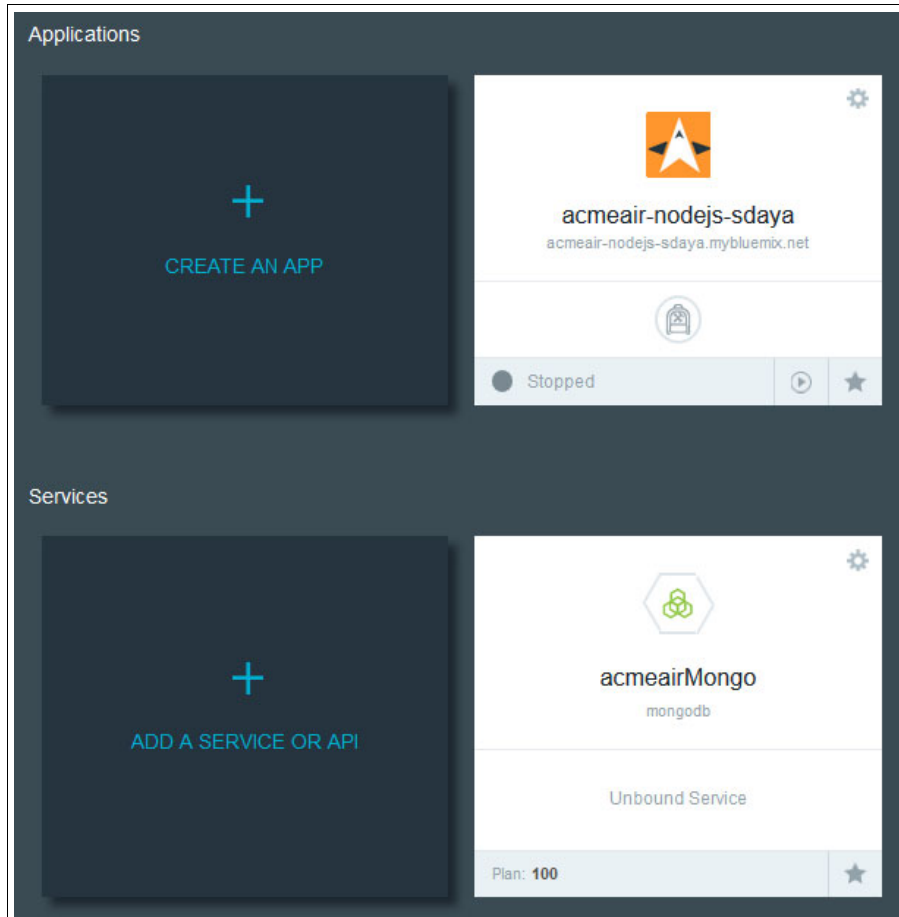


Figure 8-6 MongoDB service in Bluemix

Cloudant

For Cloudant, the following CF command creates the database:

```
cf create-service cloudantNoSQLDB Shared acmeairCL
```

Figure 8-7 shows the execution of the **create-service** command for Cloudant.

```
C:\Users\IBM_ADMIN\Documents\0.WIP\0.Code\acmeair-nodejs-public-v2.git>cf create-service cloudantNoSQLDB Shared acmeairCL
Creating service acmeairCL in org sdaya@ca.ibm.com / space AcmeAir as sdaya@ca.ibm.com...
OK
C:\Users\IBM_ADMIN\Documents\0.WIP\0.Code\acmeair-nodejs-public-v2.git>
```

Figure 8-7 Creating a Cloudant service

In your Bluemix dashboard, you should see the Cloudbant service that you just created, as shown in Figure 8-8.

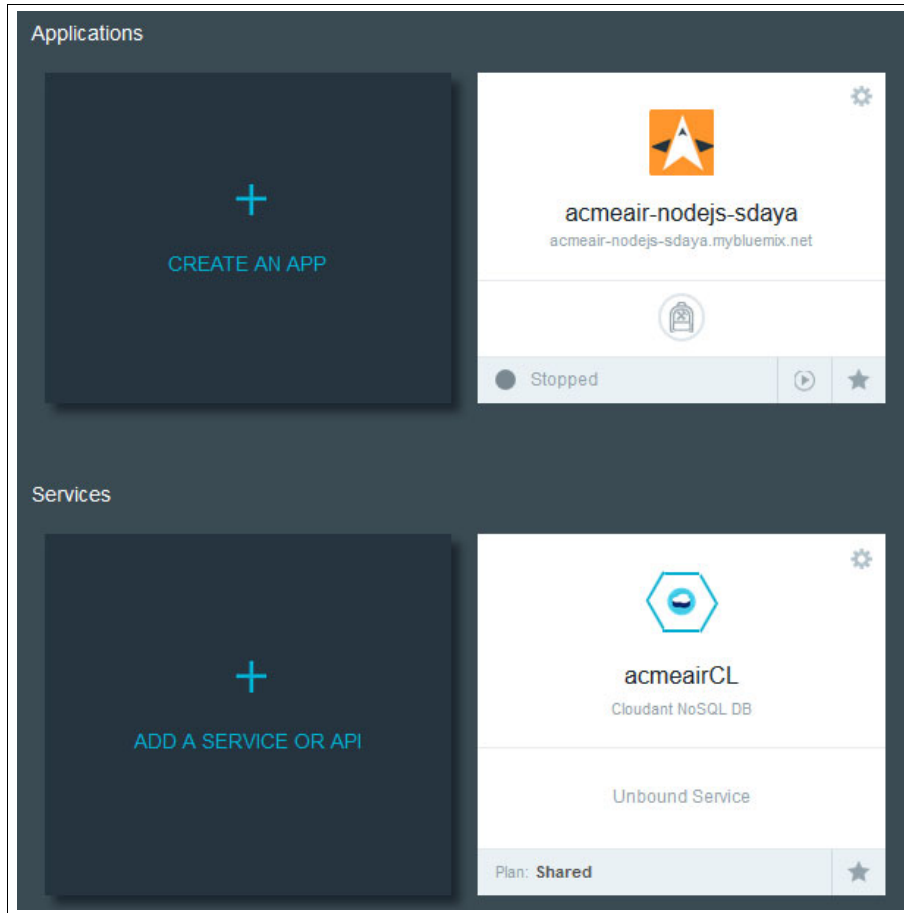


Figure 8-8 Cloudbant service in Bluemix

For Cloudant, we need to create the database and the search index. In the source code, you have the following file:

```
/document/DDL/cloudant.ddl
```

Use this file to create the database and search index.

By this point, we have the Acme Air web app deployed and a database created. Next, we need to bind the web app to the database service.

Bind the Acme Air web app to the database service

The following CF command binds the MongoDB service to the Acme Air web app:

```
cf bind-service acmeair-nodejs-<uid> acmeairMongo
```

For the Cloudant service the following command is similar, with the only difference being the name of the service being bound:

```
cf bind-service acmeair-nodejs-<uid> acmeairCL
```

Do not forget to replace `<uid>` with the unique ID you used to name your app. The screen capture in Figure 8-9 shows the execution of a `bind`.

```
C:\Users\IBM_ADMIN\Documents\0.WIP\0.Code\acmeair-nodejs-public-v2.git>cf bind-service acmeair-nodejs-sdaya acmeairMongo
Binding service acmeairMongo to app acmeair-nodejs-sdaya in org sdaya@ca.ibm.com / space AcmeAir as sdaya@ca.ibm.com...
OK
TIP: Use 'cf restage' to ensure your env variable changes take effect
C:\Users\IBM_ADMIN\Documents\0.WIP\0.Code\acmeair-nodejs-public-v2.git>
```

Figure 8-9 Binding a database to the Acme Air web app

Now we are ready to start the application.

Start the Acme Air application

Now that the Acme Air web app has been deployed and bound to a database, we can start the application. The following CF command starts the application:

```
cf start acmeair-nodejs-<uid>
```

Do not forget to replace `<uid>` with the unique ID that you used to name your app when you did the `cf push`. The screen capture in Figure 8-10 shows the result of the `cf start` command.

```
C:\Users\IBM_ADMIN\Documents\0.WIP\0.Code\acmeair-nodejs-public-v2.git>cf start acmeair-nodejs-sdaya
Starting app acmeair-nodejs-sdaya in org sdaya@ca.ibm.com / space AcmeAir as sdaya@ca.ibm.com...
OK
-----> Downloaded app package (892K)
-----> Downloaded app buildpack cache (3.1M)
-----> Node.js Buildpack Version: v1.16-20150409-1839
-----> Resetting git environment
-----> Requested node range: 0.10.x
-----> Resolved node version: 0.10.38
-----> Installing IBM SDK for Node.js from cache
-----> Checking and configuring service extensions
-----> Restoring node_modules directory from cache
-----> Pruning cached dependencies not specified in package.json
-----> Installing dependencies
-----> Caching node_modules directory for future builds
-----> Cleaning up node-gyp and npm artifacts
-----> No Procfile found; Adding npm start to new Procfile
-----> Building runtime environment
-----> Checking and configuring service extensions
-----> Found bunyan, log4js or ibmbluemix dependency. Installing Bluemix management client to your application
-----> Install local dependency: bluemix-management-client
bluemix-management-client@1.0.1 node_modules/bluemix-management-client
-----> Found boot js file: app.js
-----> Add header to app.js: require("bluemix-management-client");
-----> Installing App Management
-----> Uploading droplet (13M)

0 of 1 instances running, 1 starting
1 of 1 instances running

App started

Showing health and status for app acmeair-nodejs-sdaya in org sdaya@ca.ibm.com / space AcmeAir as sdaya@ca.ibm.com...
OK
requested state: started
instances: 1/1
usage: 1G x 1 instances
urls: acmeair-nodejs-sdaya.mybluemix.net

#0 state since cpu memory disk
   running 2015-04-30 03:31:27 PM 0.0% 40.9M of 1G 53.4M of 1G
C:\Users\IBM_ADMIN\Documents\0.WIP\0.Code\acmeair-nodejs-public-v2.git>
```

Figure 8-10 Starting the Acme Air web app

You can also start the web app from the Bluemix dashboard. When the application is started, go to the route in a web browser. The following route is an example:

<http://acmeair-nodejs-<uid>.mybluemix.net>

You should see the Acme Air home page shown in Figure 8-11.

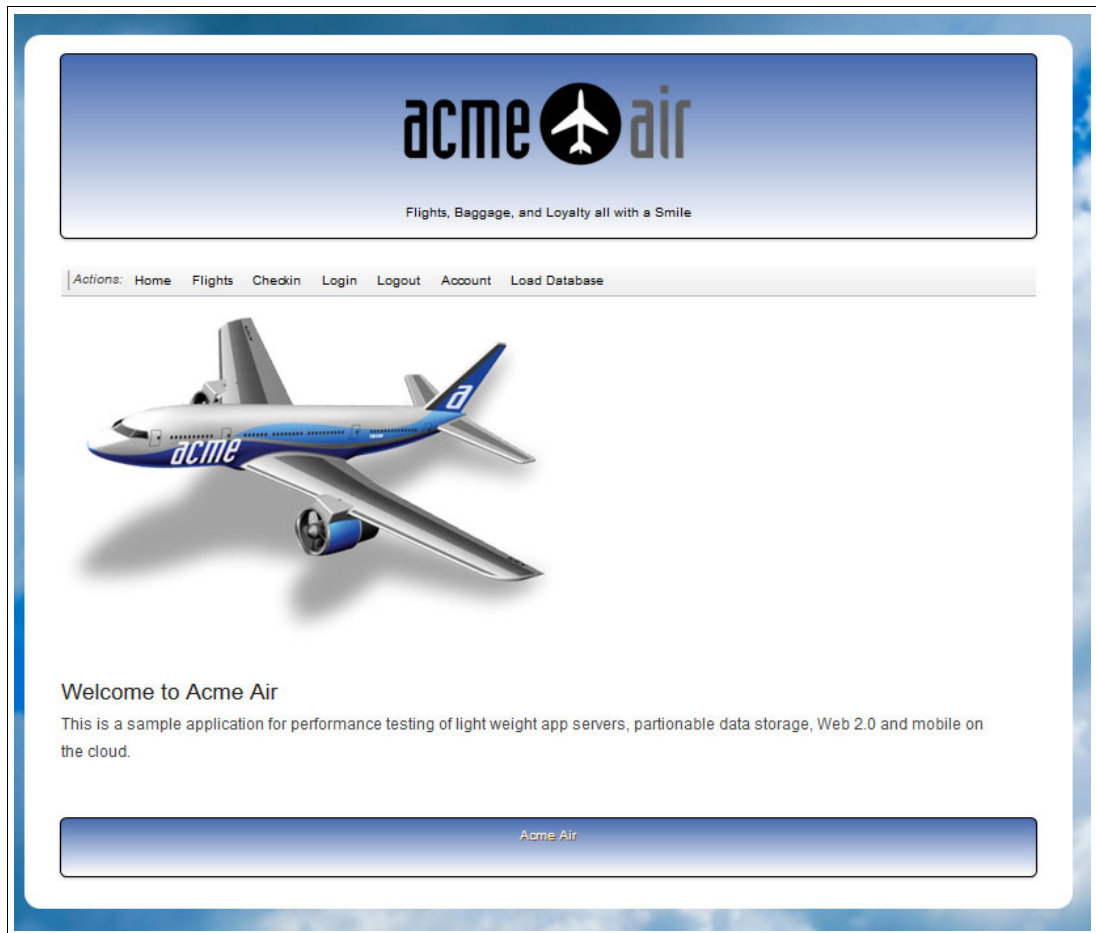


Figure 8-11 Acme Air home page

The first thing you must do is click the **Load Database** action in the top menu bar. You can now log in, search for flights, and so forth.

8.2 Redesigning application to use microservices

The original Acme Air monolithic application was changed to efficiently use the capabilities of a microservices architecture. We use an implementation of the Acme Air sample application for Node.js, which uses server-side JavaScript. This implementation can support multiple data stores, can run in several application modes, and can support running on various runtime platforms, including stand-alone bare metal system, virtual machines, Docker containers, IBM Bluemix, and IBM Bluemix Container service.

The Acme Air application has the following features:

- ▶ Query flights
- ▶ Book flights
- ▶ Account profile

The Acme Air project was transformed from its original monolithic design to efficiently use microservices architecture by splitting the authentication service off from the core application to its own separate application (service).

For the auth-service, the customersession methods from customerservice were moved to a separate Representational State Transfer (REST) service. The web app was changed to use REST calls rather than local calls. The authentication service is called on every request, so the new design supports better scalability of subsystems. A central database is used by both the web front end and the authservice.

Figure 8-12 shows the Acme Air application architecture for the microservices version.

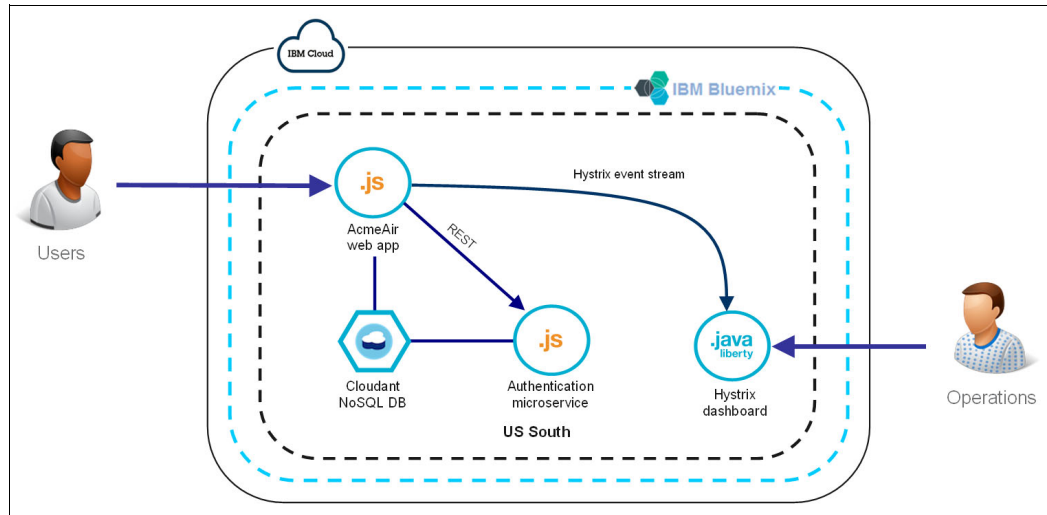


Figure 8-12 Acme Air application architecture for microservices version

To implement a microservices architecture for this application, the main Node.js application delegates to the authorization service Node.js application hosted on host:port, defined in **AUTH_SERVICE**. End-to-end requests from a client include REST endpoints, business logic, and data tier interactions.

The Acme Air application enables users to choose a data store of either Cloudant, MongoDB or Cassandra. Figure 8-18 on page 145 shows the application deployed in Bluemix. A MongoDB (acmeairMongo) is used by both the web front end and the authentication microservice (uniquely named acmeair-nodejs-sdaya and authservice-sdaya in our deployment).

8.2.1 Deploying the Acme Air application in microservices mode

To deploy the Acme Air application in microservices mode, you need to first follow the steps to deploy it in monolithic mode. After you have done that, follow the steps described in the following sections.

Log in to Bluemix

Log in to Bluemix as described earlier using the `cf api` and `cf login` commands.

Push the Acme Air authentication service

The following CF command pushes the authentication service to Bluemix. This is similar to pushing the web app described in an earlier section, the difference being the .js file that is run by the node. In the case of the web app, it was app.js. In this case, it is authservice-app.js:

```
cf push authservice-<uid> --no-start -c "node authservice_app.js"
```

Bind the authentication service to the database service

Just like the Acme Air web app, the authentication service needs to be bound to the database. Just as before, depending on the database that you created, you use one of the following two CF commands:

```
cf bind-service authservice-<uid> acmeairMongo
```

or

```
cf bind-service authservice-<uid> acmeairCL
```

Start the authentication service

Start the authentication service using the following CF command:

```
cf start authservice-<uid>
```

At this point, you have an authentication service up and running. You now need to configure the web app to use it. The section that follows provides details about what needs to be done.

Configure the Acme Air web app to use the authentication service

For the Acme Air web app to use the authentication service rather than the authentication mechanism built into it, we need to set an environment variable called `AUTH_SERVICE` and give it the Uniform Resource Locator (URL) of the authentication service. The following CF command creates this environment variable:

```
cf set-env acmeair-nodejs-<uid> AUTH_SERVICE authservice-<uid>.mybluemix.net:80
```

Figure 8-13 shows the result of this command.

```
C:\Users\IBM_ADMIN\Documents\0_WIP\0_Code\acmeair-nodejs-public-v2.git>cf set-env acmeair-nodejs-sdaya AUTH_SERVICE authservice-sdaya.mybluemix.net:80
Setting env variable 'AUTH_SERVICE' to 'authservice-sdaya.mybluemix.net:80' for app acmeair-nodejs-sdaya in org sdaya@ca.ibm.com / space AcmeAir as sdaya@ca.ibm.com...
OK
TIP: Use 'cf restage' to ensure your env variable changes take effect
C:\Users\IBM_ADMIN\Documents\0_WIP\0_Code\acmeair-nodejs-public-v2.git>
```

Figure 8-13 Setting the `AUTH_SERVICE` environment variable

In the Bluemix dashboard under Environment Variables for the Acme Air web app, you should see the environment variable that you just created. It is in the USER-DEFINED tab, as shown in Figure 8-14.

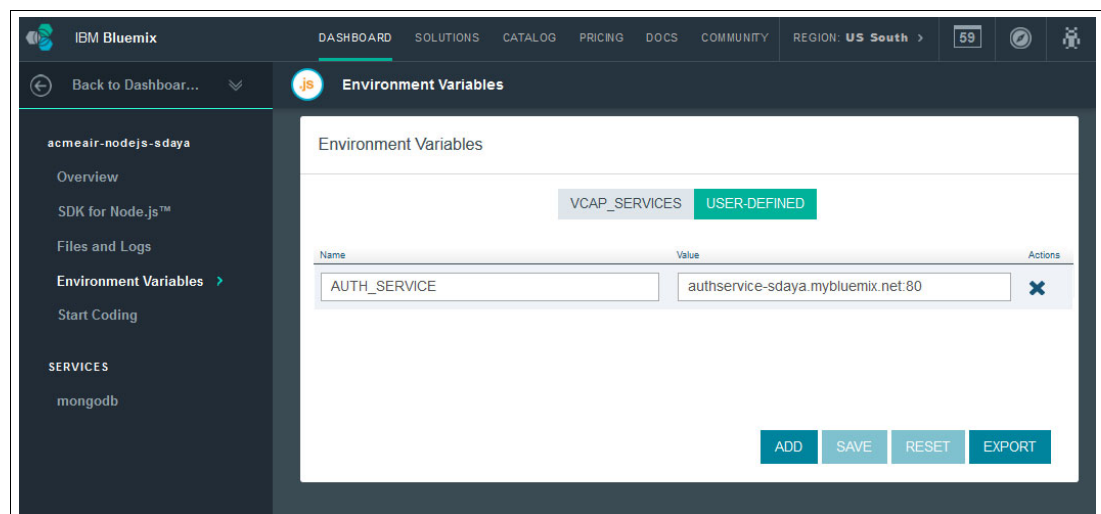


Figure 8-14 The `AUTH_SERVICE` environment variable in the Bluemix dashboard

You need to restart the Acme Air web app for this environment variable to be picked up. You might want to enable Hystrix first, before restarting the app.

Enable Hystrix

Enabling the use of Hystrix is also done using an environment variable. An environment variable named **enableHystrix** needs to be set to true using the following CF command:

```
cf set-env acmeair-nodejs-<uid> enableHystrix true
```

Again, if you go to the Bluemix dashboard, you can see the **enableHystrix** environment variable set to true, as shown in Figure 8-15.

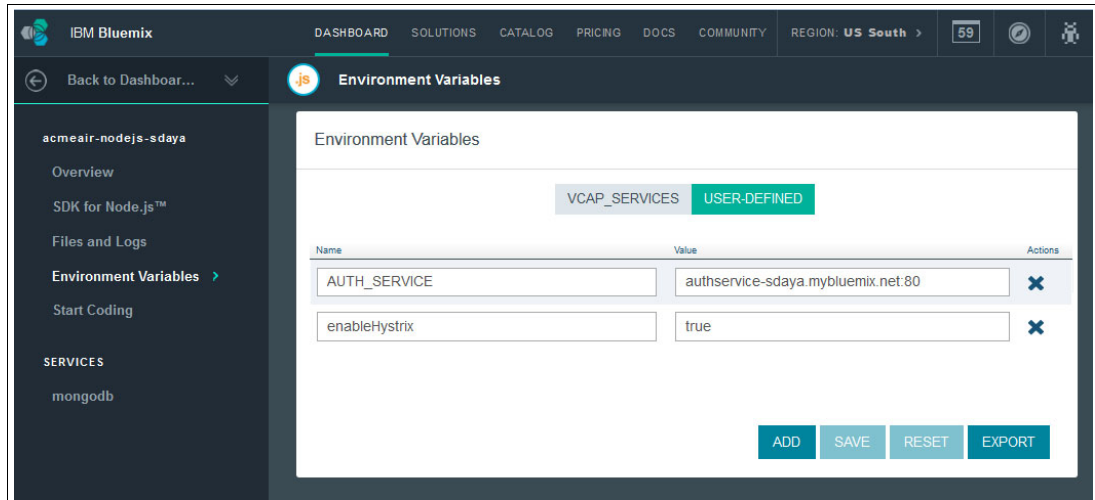


Figure 8-15 The enableHystrix environment variable in the Bluemix dashboard.

Restart the Acme Air web app and try to perform a login. This now uses the authentication service.

8.3 Adding Hystrix to monitor services

Creating a highly agile and highly available service on the cloud is possible if you prepare and plan for the occurrence of broken or failed components, and take measures to keep the other services functioning, not causing a total application failure.

To redesign the Acme Air application, the team developed application logic to separate the authorization services from the webapp service. To enable resilience and fault-tolerance, circuit breaker and metrics were introduced through a command pattern to fail fast and rolling window statistics collection. Hystrix monitoring was also enabled for these services by generating Hystrix Stream with JavaScript Object Notation (JSON), which is used by Hystrix Dashboard.

By incorporating Netflix OSS Hystrix-like functions (command pattern, circuit breaker, and metrics), we have protection from and control over latency, and over failure from dependencies. We are able to stop cascading failures in our microservices distributed system. If a failure occurs, it is rapidly recovered. The near real-time monitoring, alerting, and operations control enables us to detect if a failure occurs, and rapidly recover. We can fallback and gracefully degrade if necessary.

Hystrix Dashboard enables us to be alerted, make decisions, affect change, and see results in seconds. Adding Hystrix-like functionality (command pattern, circuit breaker, and metrics) to your microservices architecture helps defend your application by providing a greater tolerance of latency and failure. Overall, this helps you gradually degrade rather than fail completely.

Applications in a complex distributed architecture have many dependencies, each of which is susceptible to failure. If the host application is not isolated from these external failures, the host application is at risk of failure whenever a dependency fails. Threads and semaphores can be isolated with circuit breakers.

By implementing a *command pattern* + *circuit breaker* design, we can achieve fault-tolerance and resilience through fail-fast. Real-time monitoring and configuration changes enable you to see service and property changes take effect immediately.

8.3.1 Deploying the Hystrix Dashboard to Bluemix

To deploy the Hystrix Dashboard, you need to download the web archive (WAR) file for the dashboard. You can find a link to the download on the following website:

<https://github.com/Netflix/Hystrix/wiki/Dashboard#installing-the-dashboard>

The following CF CLI command deploys the Hystrix Dashboard to Bluemix:

```
cf push hystrix-<uid> -p hystrix-dashboard-1.4.5.war
```

The version of the Hystrix Dashboard at the time we wrote this was 1.4.5. The WAR file gets deployed to the IBM WebSphere Application Server Liberty for Java run time in Bluemix. When the Hystrix Dashboard app is running, you can access the dashboard using the following route:

<http://hystrix-<uid>.mybluemix.net>

You should see the Hystrix Dashboard, as shown in Figure 8-16.

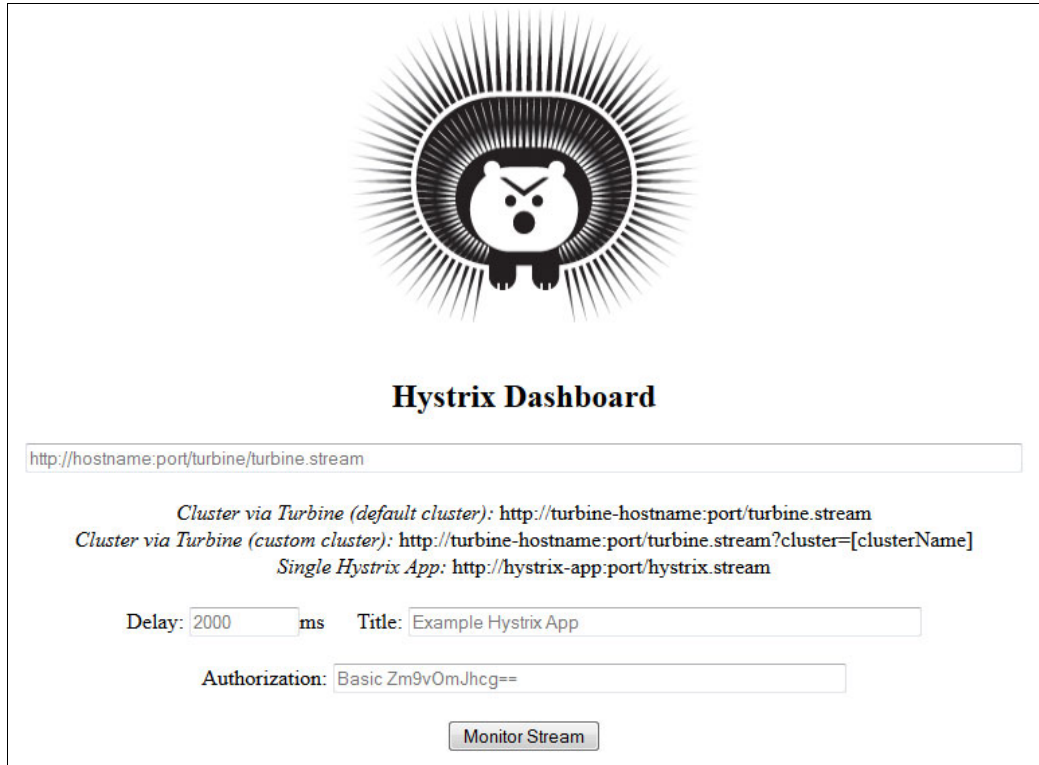


Figure 8-16 Hystrix Dashboard home page

To monitor the Acme Air authentication service, you need to monitor the following Hystrix event stream:

<http://acmeair-nodejs-<uid>.mybluemix.net/rest/api/hystrix.stream>

Specify that stream on the Hystrix Dashboard home page and click the **Monitor Stream** button. You can now see the details, as shown in Figure 8-17.

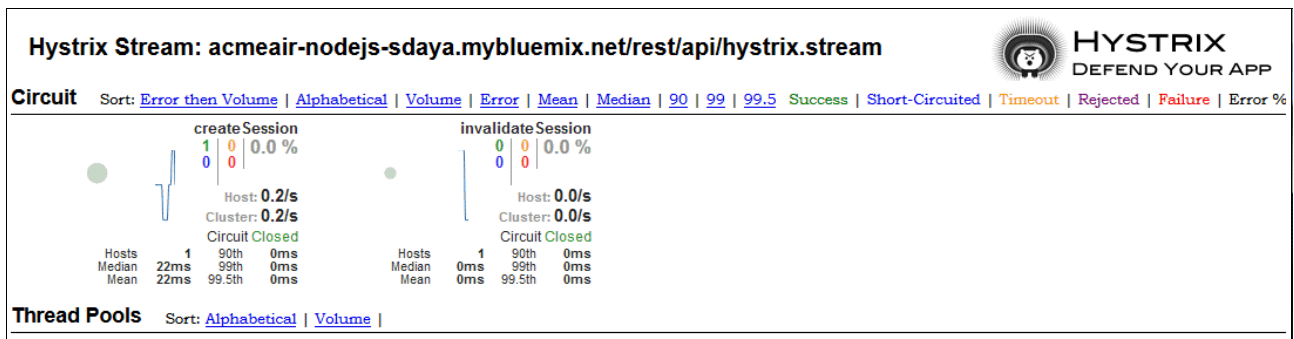


Figure 8-17 Hystrix Dashboard

The screen capture in Figure 8-18 shows all of the Acme Air components for the microservices mode in the Bluemix dashboard.

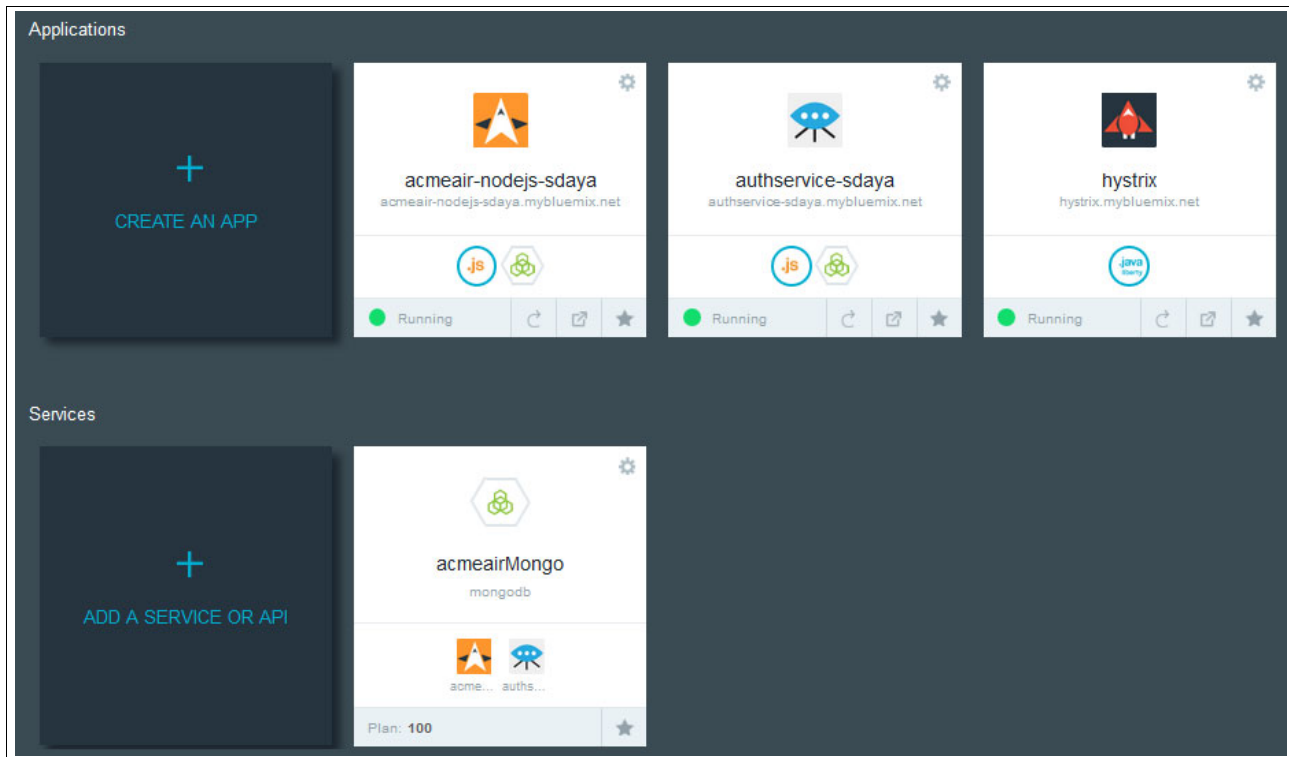


Figure 8-18 Bluemix dashboard shows the Acme Air components

New design

A benefit seen with microservices architecture is the ability to continuously update systems in real time, deploying code features in a short time and measuring instant responses. Creating a highly agile and highly available service on the cloud is possible if you prepare and plan for the occurrence of broken or failed components. Also consider how to keep the other services functioning, not causing a total application failure. Hystrix offers a fault tolerance solution to defend your application.

The application needed the ability to scale to billions of web API calls per day, and to support mobile user interactions. The original Acme Air monolithic application was changed to efficiently use the capabilities of cloud deployment, microservices architecture, and Netflix OSS Hystrix monitoring.

The end result is a fault-tolerant, high-performing application design that puts control in the hands of the development team. The microservices design provides better fault tolerance of the main web application with regards to dependent services.

Related publications

The publications listed in this section are considered particularly suitable for a more detailed description of the topics covered in this book.

IBM Redbooks

The following IBM Redbooks publications provide additional information about the topic in this document. Note that some publications referenced in this list might be available in softcopy only:

- ▶ *Smart SOA Connectivity Patterns: Unleash the Power of IBM WebSphere Connectivity Portfolio*, SG24-7944

You can search for, view, download or order these documents and other Redbooks, Redpapers, Web Docs, draft and additional materials, at the following website:

ibm.com/redbooks

Other publications

This publication is also relevant as a further information source:

- ▶ *Building Microservices*, 978-1491950357

Online resources

These websites are also relevant as further information sources:

- ▶ IBM Bluemix home page:
<https://bluemix.net>
- ▶ IBM Bluemix main documentation page:
<https://www.ng.bluemix.net/docs/>
- ▶ IBM SOA solutions website:
<http://www-01.ibm.com/software/solutions/soa/>
- ▶ IBM developerWorks article about autoscaling on Bluemix:
<http://www.ibm.com/developerworks/cloud/library/cl-bluemix-autoscale/>
- ▶ Information about creating Bluemix spaces:
<https://www.ng.bluemix.net/docs/#acctmgmt/index.html#acctmgmt>
- ▶ Information about Deploy To Bluemix button:
<https://www.ng.bluemix.net/docs/#manageapps/index-gentopic2.html#appdeploy>
- ▶ Information about manifest files:
<http://docs.cloudfoundry.org/devguide/deploy-apps/manifest.html>

- ▶ Information about deploying your application using the Cloud Foundry CLI:
https://www.ng.bluemix.net/docs/#starters/upload_app.html
- ▶ Information about IBM Cloud Integration service:
<https://www.ng.bluemix.net/docs/#services/CloudIntegration/index.html#gettingstartedwithcloudintegration>
- ▶ Information about IBM Secure Gateway:
<https://www.ng.bluemix.net/docs/#services/SecureGateway/index.html#gettingstartedsecuregateway>
- ▶ IBM SoftLayer website:
<http://www.SoftLayer.com>

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

Redbooks

Microservices from Theory to Practice: Creating Applications in IBM Bluemix Using the Microservices Approach

(0.2"spine)

0.17"->0.473"

90->249 pages



SG24-8275-00

ISBN 0738440817

Printed in U.S.A.

Get connected

