Redbooks

ibm.com/redbooks

Evolve the Monolith to Microservices with Java and Node

Sandro De Santis

Luis Florez

Duy V Nguyen

Eduardo Rosa



Cloud

WebSphere

IBM

Redbooks

www.dbooks.org



International Technical Support Organization

Evolve the Monolith to Microservices with Java and Node

December 2016

SG24-8358-00

www.dbooks.org

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

First Edition (December 2016)

This edition applies to IBM WebSphere Application Server Liberty V8.5.5, IBM API Connect V 5.0, IBM Bluemix.

© Copyright International Business Machines Corporation 2016. All rights reserved.
Note to U.S. Government Users Restricted Rights -- Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

www.dbooks.org

Contents

Notices ix
Trademarks x

Preface xi
Authors xi
Now you can become a published author, too! xiii
Comments welcome xiii
Stay connected to IBM Redbooks xiii

Chapter 1. Overview of microservices 15
1.1 What is the microservices architecture 16
1.1.1 Small and focused around business domain 16
1.1.2 Technology neutral 16
1.1.3 Loosely coupled 17
1.1.4 Highly observable 17
1.1.5 Automation 18
1.1.6 Bounded context 18
1.2 Reasons for adopting the microservices architecture 19
1.2.1 Challenges with existing monolithic applications 19
1.2.2 Microservices for various stakeholders 21
1.2.3 Timing with availability of cloud era and other tooling 22
1.3 Transformation from a monolith to microservices 23
1.3.1 Fictional Company A business problem 23
1.3.2 Adopting microservices for an evolutionary architecture 24

Chapter 2. Evolution strategies 27
2.1 Microservices: An evolutionary architecture by nature 28
2.2 Evolutionary practices and patterns 30
2.2.1 Continuous integration, continuous delivery 30
2.2.2 Strangler application 30
2.2.3 Evolutionary database refactoring and deployment 31
2.3 Evolutionary architecture in practice 32
2.3.1 Gradually transforming to a catalog service 33
2.3.2 Strangling the customer and order components 33
2.3.3 Evolutionarily transforming the user interface to a UI microservice 34

Chapter 3. Identifying candidates within the monolith 35
3.1 Identifying candidates 36
3.2 Considerations for moving to microservices 36
3.3 Decomposing monolith application into microservices 37
3.3.1 Designing microservices 38
3.3.2 Choosing the implementation stack 40
3.3.3 Sizing the microservices 41
3.4 Refactoring 42
3.4.1 A reason to refactor to microservices 42
3.4.2 Strategies for moving to microservices 42
3.4.3 How to refactor Java EE to microservices 43
3.5 Identifying and creating a new architecture example 46
3.5.1 Architecture: As is 46

© Copyright IBM Corp. 2016. All rights reserved.

www.dbooks.org

3.5.2 Architecture: To be 47

Chapter 4. Enterprise data access patterns 51
4.1 Distributed data management 52
4.2 New challenges 52
4.2.1 Data consistency 52
4.2.2 Communication between microservices 58
4.3 Integration with the monolith application 61
4.3.1 Modifying the monolith application 61
4.4 Which database technology to use 61
4.4.1 Relational databases 61
4.4.2 NoSQL databases 62
4.5 Practical example: Creating the new microservices 63
4.5.1 New microservice catalog search 63
4.5.2 Evolution of the account data model 74

Chapter 5. Security and governance 81
5.1 Security in microservice architecture 82
5.1.1 Authentication and authorization 82
5.1.2 Service-to-service authentication and authorization 83
5.1.3 Data security 84
5.1.4 Defence in depth 85
5.2 Governance in microservice architecture 86
5.3 The case study 86
5.3.1 Integrating and securing the monolith system 87
5.4 Summary 92

Chapter 6. Performance 95
6.1 Common problems 96
6.2 Antipatterns to avoid 97
6.2.1 N+1 service call pattern 97
6.2.2 Excessive service calls 98
6.2.3 Garrulous services 99
6.2.4 Lack of caching 100
6.3 Lead practices to improve performance 100
6.3.1 Let the service own its operational data 100
6.3.2 Avoid chatty security enforcement 100
6.3.3 Leverage messaging and REST 100
6.3.4 Fail fast and gracefully 102

Chapter 7. DevOps and automation 103
7.1 Deployment automation 104
7.1.1 Development 104
7.1.2 Build 104
7.1.3 Test 108
7.1.4 Deploy 109
7.2 Monitoring 112
7.2.1 Collecting data 112
7.2.2 Dashboard 113
7.2.3 Error handling 114
7.3 Performance 115
7.3.1 Automatic scalability 115
7.3.2 Service discovery 117
7.3.3 Load balancing 117

Appendix A. Additional material 119
Locating the Web material 119
Cloning the GitHub repository 119
Social networks JSON data model example 119

Related publications 125
Other publications 125
Online resources 125
Help from IBM 125

www.dbooks.org

Notices

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

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

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing, IBM Corporation, North Castle Drive, MD-NC119, Armonk, NY 10504-1785, US

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

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

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

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

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

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

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

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

COPYRIGHT LICENSE:

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

© Copyright IBM Corp. 2016. All rights reserved.

www.dbooks.org

Trademarks

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

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

Bluemix®
Cloudant®
Concert™
DataPower®
DataStage®
DB2®
developerWorks®
IBM®
IBM API Connect™
IBM Watson™
PureApplication®
Rational®
Redbooks®
Redbooks (logo) ®
Terraform®
WebSphere®

The following terms are trademarks of other companies:

Evolution, and Inc. device are trademarks or registered trademarks of Kenexa, an IBM Company.

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

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

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

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

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

Preface

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

This IBM® Redbooks® publication shows how to break out a traditional Java EE application into separate microservices and provides a set of code projects that illustrate the various steps along the way. These code projects use the IBM WebSphere® Application Server Liberty, IBM API Connect™, IBM Bluemix®, and other Open Source Frameworks in the microservices ecosystem. The sample projects highlight the evolution of monoliths to microservices with Java and Node.

Authors

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



Sandro De Santis is an IT Architect in Brazil. He joined IBM in 2011. Before joining IBM, Sandro worked in the financial, insurance, and telecom industries. He has experience working in Java Platform, Enterprise Edition (JEE) and mobile and cloud platforms. Sandro is a SUN Java Enterprise Architect and IBM SOA. He has experience working in the Java, C#, JavaScript, and NodeJS languages.



Luis Florez is an IBM Enterprise Social Solutions Technical Specialist in Colombia. He joined IBM in 2014 as an IBM Bluemix Technical Specialist. He has eight years of experience in software design and development working with programing languages such as Java, PHP, C#, and JavaScript. He moved to Enterprise Social Solution Technical Sales in 2016 and now works with clients in Colombia. He specializes in helping his clients get the most out of the IBM Collaboration Portfolio with a focus on hybrid cloud.

© Copyright IBM Corp. 2016. All rights reserved.

www.dbooks.org



Duy V Nguyen is an IBM Certified IT Architect with solid experience in IBM and open technologies. He is also an IBM Cloud Engineer serving in the Cloud Engineering and Services team in the IBM CIO Office, Transformation and Operations group in IBM CHQ/US. His daily job is helping IBM to transform using new technologies, specifically cloud, mobile, and other emerging technologies. He is focusing on creation of cloud and mobile solutions for IBM employees, and providing his expertise in assisting IBM clients with enterprise mobile and cloud engagements as needed. His core experiences are in web, security, cloud, and mobile technologies.



Eduardo Rosa is an IT Specialist working for IBM Brazil as a technical advisor for IBM Bluemix. He helps customers to understand, adopt, and have success using the Bluemix platform. Before joining the Bluemix team, he worked for several years for the WebSphere portfolio. With nearly 20 years of IT experience, Eduardo has experience as a computer programmer, team leader, solution/software architect, and in software technical pre-sales. He is a technology enthusiast with deep knowledge of JEE, SOA, BPM, CEP, BAM, BRMS, and cloud computing. Eduardo loves Linux, but has learned to live in peace with all human beings, no matter what their OS choice is.

This project was led by:

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

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

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

Kyle Brown
Distinguished Engineer, CTO Cloud Architecture, IBM Cloud

Iain Duncan
Software Engineer, IBM Cloud

Erin Schnabel
WebSphere Liberty developer/engineer/guru/evangelist, and Microservices Architect, IBM Cloud

Adam Pilkington
Master Inventor, Software Developer, WebSphere Liberty Profile, IBM Cloud

Darrell Reimer
Distinguished Engineer, IBM Research

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

- Diane Sherman, IBM Redbooks Content Specialist
- Susan Marks, IBM Redbooks Content Specialist

- Ann Lund, IBM Redbooks Residency Administrator
- Thomas Edison, IBM Redbooks Graphics Editor

Now you can become a published author, too!

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

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

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

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

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

  ibm.com/redbooks

- Send your comments in an email to:

  redbooks@us.ibm.com

- Mail your comments to:

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

Stay connected to IBM Redbooks

- Find us on Facebook:

  http://www.facebook.com/IBMRedbooks

- Follow us on Twitter:

  http://twitter.com/ibmredbooks

- Look for us on LinkedIn:

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

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

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

www.dbooks.org

- Stay current on recent Redbooks publications with RSS Feeds:
  http://www.redbooks.ibm.com/rss.html

1

Overview of microservices

This chapter provides an overview of the key concepts of microservices architecture. This overview will help you understand what microservices architecture is and why it is so appealing for modern software application development in today's environments. The chapter also describes characteristics and principles of microservices architecture and introduces a scenario that is used throughout this book.

This chapter includes the following sections:

- What is the microservices architecture
- Reasons for adopting the microservices architecture
- Transformation from a monolith to microservices

© Copyright IBM Corp. 2016. All rights reserved.

www.dbooks.org

1.1 What is the microservices architecture

Microservices are relatively small, autonomous services that work collaboratively together. Each of the services implements a single business requirement.

Microservices architecture is an architectural style defined by Martin Fowler and James Lewis. They describe the style as, “an architectural approach that consists of building systems from small services, each in their own process, communicating over lightweight protocols.” For more information, see the following website:

http://martinfowler.com/articles/microservices.html?cm_mc_uid=34013674245414737751113&cm_mc_sid_50200000=147499071

The services are developed and deployed independently of one another. Each of these microservices focuses on performing a relatively small task that it does well.

The following sections highlight several characteristics of a typical microservices architecture for a conceptual look at the architectural style.

1.1.1 Small and focused around business domain

Although *small* is not a sufficient measure to describe microservices, its use is an attempt to imply the size of a microservice in comparison to a monolithic application. The definition of small in this context might vary among systems, and no rule exists that defines how small a service must be.

A microservice is typically responsible for a granular unit of work, so it is relatively small in size. One well-known guideline is the “Two-pizza team sizing” formula, which states that, *If you cannot feed your team with two pizzas, the microservice they are working on might be too big*. The microservice must be small enough so that everyone in the team can understand the whole design and implementation of the service. Also, it must be small enough that the team can maintain or rewrite the service easily, if necessary.

Another important characteristic of microservices is that each service focuses on fulfilling a granular business responsibility. Vaughn Vernon defined the term *business domain* in his book *Implementing Domain-Driven Design*. He defined business domain as, “what an organization does and the world it does it in.” He also specified that “each kind of organization has its own unique realm of know-how and way of doing things. That realm of understanding and its methods for carrying out its operations is its domain.” The unit of decomposition into microservices is effectively the business entity within the domain, meaning each microservice handles a completed business task. For example: Mortgage Services is a business domain. Loan Origination is a potential microservice in that domain. Loan Refinancing might be another microservice within the same domain. Calls and changes across service boundaries are usually expensive and must be avoided. The teams that own those services become expert in the corresponding business domain or sub-domains rather than becoming expert in some arbitrary technical areas.

1.1.2 Technology neutral

Teams that develop microservices must use technology that they are comfortable with. Do not dictate to a development team which programming language to use. Give developers the freedom to use whichever technologies make the most sense for the task and the people performing the task. This way of working takes full advantage of the most optimal technologies and skills that team members have. Microservices architecture still requires

technology decisions; for example, is access better with the Representational State Transfer (REST) application programming interface (API) or with some type of queuing? However, in general, you can choose any technology for your microservices architecture within broad limits.

1.1.3 Loosely coupled

Loose coupling is critical to a microservices-based system. Each microservice must have an interface designed in such a way that it has low coupling to other services. This allows you to make changes to one service and deploy it without having to change and redeploy other parts of the system.

To avoid coupling between services, you must understand what causes tight coupling. One cause of tight coupling is exposing the internal implementation of a service through its API. This exposure binds consumers of the service to its internal implementation, which leads to increased coupling. In this case, if you change the internal architecture of the microservice, the change might require the service's consumers to also change or be broken. That might increase the cost of change and potentially create concern about implementing change, which can lead to increased technical debt within the service. Any method that results in exposing internal implementation of a service must be avoided to ensure loose coupling between microservices.

Another mistake is making the API of a service too fine grained. If the API is too fine grained, calls among services can become too *chatty,* that is, more calls are being made through the network. Beyond potential performance problems, chatty communication can create tight coupling. Hence, the interface of the service must be designed in such a way that minimizes the back and forth calls placed in the network.

You must avoid low-cohesion implementation in one service by placing related properties, behaviors that represent a business entity, as close together as possible. Place related properties in one microservice; if you make a change to a property or behavior, you can change it in one place and deploy that change quickly. Otherwise, you must make changes in different parts and then deploy those scattered parts together at the same time; this results in tight coupling among those parts.

Each microservice must have its own source-code management repository and delivery pipeline for building and deployment. This allows each service to be deployed as necessary without the need for coordination with owners of other services. If you have two services and you always release those two services together in one deployment, that might indicate that the two services are better as one service and further work must be done against the current services decomposition. Loose coupling also enables more frequent and rapid deployments that eventually improve the responsiveness of the application to the needs of its users.

1.1.4 Highly observable

A microservices architecture requires a means for you to *visualize* the health status of all services in the system and the connections between them. This allows you to quickly locate and respond to problems that might occur. Tools that enable visualization include a comprehensive logging mechanism to record, store, and make the logs searchable for better analysis.

The more new services provisioned and added to the system, the more difficult it is to make those services observable. Because microservice architecture adds more complexity when more dynamic parts are added, the observation design must be clear so the logging and monitoring data being visualized provide helpful information for analysis.

www.dbooks.org

1.1.5 Automation

Automation is an important requirement for the effective design and implementation of any software application. For microservices, automation is a critical, but also challenging, task.

Microservices architecture introduces more complexities into the implementation of a system, in addition to operating the system in production. With a small number of machines and components to work with, it might be acceptable to manually provision a machine, install middleware, deploy the components, or manually log in to a server and collect the logs, and other manual tasks. When the number of components are increased, however, at some point you might not be able to use the manual approach.

Automation helps you to spin up a server and install necessary runtime environments. You can then put your microservices on those runtime environments quickly using lines of code. This automation allows you the ability to code the microstructure and to access the exact tool chain that is used for deployment of production services, so problems can be caught early. Automation is the core enabler of continuous integration and continuos delivery methods.

Embracing a culture of automation is key if you want to keep the complexities of microservices architecture in check. To do that, you need a comprehensive end-to-end approach to embrace automation through the whole software development lifecycle. This lifecycle includes test driven development through operations, such as IBM Bluemix® Garage Method. For more information, see the following website:

https://www.ibm.com/devops/method/

1.1.6 Bounded context

As you develop your model, remember to identify its bounded context, that is, where the model is valid. Bounded context are models with explicit boundaries where the model is unambiguous within the boundary. If you do not put a boundary around your model, eventually context is used that does not fit in your application. Context that fits in one part of your application might not fit in another, even if they have same name and refer to the same thing. For example, if you build an appointment scheduling system, you must know basic information about the client. Put in the context of appointment scheduling, this is a simple model with a little behavior beyond the name of the client. However, if you have a billing system in a billing context, you want to include contact and payment information for the client, but you do not need that information in the appointment scheduling context. If you try to reuse the same exact client model in multiple places, it is likely to cause inconsistent behavior in your system.

For example, you might decide to include some form of validation on the client model to ensure you have enough information to bill them. If you are not careful, that validation might inadvertently prevent you from being able to use the client model to schedule appointments; this is not a desirable behavior. The billing system might require clients to have a valid credit card to save changes. It does not make sense, however, for a lack of credit card information to prevent you from saving an appointment for the client in the appointment scheduling system. In this example, you have two contexts, but the boundary between them is blurred and overlapping. Eric Evans notes in his book *Domain-Driven Design* that “models are only valid within specific contexts. Therefore, it is best to explicitly define the contexts within which a model is applied. You can avoid compromising the model within the context, keeping it strictly consistent within these bounds, and avoiding distraction or confusion from outside issues.”

When you explicitly define your bounded contexts, you can usually see whether you have elements of a model that are trying to expand multiple contexts. In this example, you want to

keep a simple view of the client in the appointment scheduling system, and a fuller version of the client with contact and billing information in a billing context. You define these two views of a client in two separated classes, and they then reside in separated applications. Eric Evans recommends that bounded context maintains the separation by giving each context its own team, code base, database schema, and delivery pipeline.

The principals of bounded context is critical in a microservices architecture. The principals can be used as a guidance to identify and decompose the system into microservices correctly. Having bounded contexts well-defined, meaning business domains are separated by explicit boundaries, helps to reason the resulting microservices in the system. Having bounded contexts also helps to formalize the interactions between different services and build the interfaces between them effectively and efficiently.

1.2 Reasons for adopting the microservices architecture

This section describes several main reasons for having a microservices architecture in place. A microservices architecture is one of the drivers for a product or service owner to keep up with or stay ahead of the fast evolving pace of the IT industry.

1.2.1 Challenges with existing monolithic applications

In a *monolithic application*, the majority of logic is deployed and runs in a centralized, single runtime environment or server. Monolithic applications are typically large and built by either a big team or multiple teams. This approach requires more effort by and orchestration between the teams for any change or deployment.

Over time, better architectural patterns have been introduced to the monolithic model that help to add significant flexibility into the architecture. For example, one of the well-known patterns is model view controller (MVC),which decomposes the application into layers and reasonable components. Those patterns have several advantages, such as faster initial development, simpler application governing, and other advantages. The monolithic approach, though, has disadvantages, especially in the context of fast-paced technology changes in today's environment.

A monolithic approach can introduce several challenges:

- Large application code base

  Having a large code base can create difficulties for a developer who wants to become familiar with the code, especially a new member to the team. A large application code base might also overload the development environment tools and the runtime container used in developing the application. This, consequently, can cause developers to be less productive and might discourage attempts to make changes.

- Less frequently updated

  In a typical monolithic application, most if not all of the logic components are deployed and run on a single runtime container. This means that to update one small change to a component, the entire application must be redeployed. Also, assuming you have to promote small but critical changes into production, a large effort is required to run the regression test against the parts that are not changed. These challenges mean difficulties with continuous delivery of a monolithic application, resulting in decreased deployment times and a slow response to changes needed.

www.dbooks.org

- Dependency on one technology stack

  With a monolithic model, because of the challenges of applying changes, incrementally adopting new technologies or new versions of the technology-stack development framework becomes difficult. As a result, a monolithic architecture application usually has to continue to use existing technology, which consequently is an obstacle for the application to catch up with new trends.

- Scalability

  Scalability is one of the biggest challenges of a monolithic architecture. Martin Abbot and Michael Fisher in the book *The Art Of Scalability* introduce a useful way to look at the scalability of a system; they use a three dimension scalability model or the scale cube. In this model, scaling an application by running clones behind a load balancer is known as X-axis scaling, or horizontal duplication. The other two kinds of scaling are Y-axis scaling, or functional decomposition, which is scaling by splitting different things; and Z-axis scaling, or data partitioning, which is scaling by splitting similar things. Because of the coherency as a whole, a typical monolithic application is usually able to be scaled only in one dimension in the scalability cube. As the production environment receives more requests, the application usually is vertically scaled by adding more computing resources for it to run or cloning to multiple copies in response. This way of scaling is inefficient and expensive.

  When the application reaches a certain size, the development team must be divided into smaller teams with each focusing on a particular functional area, working independently of each other, and often in different geographies. However, because of the natural coherent constraints among parts of the application, working together on changes and redeployments requires a coordinated effort by the teams. Figure 1-1 shows a scalability cube.

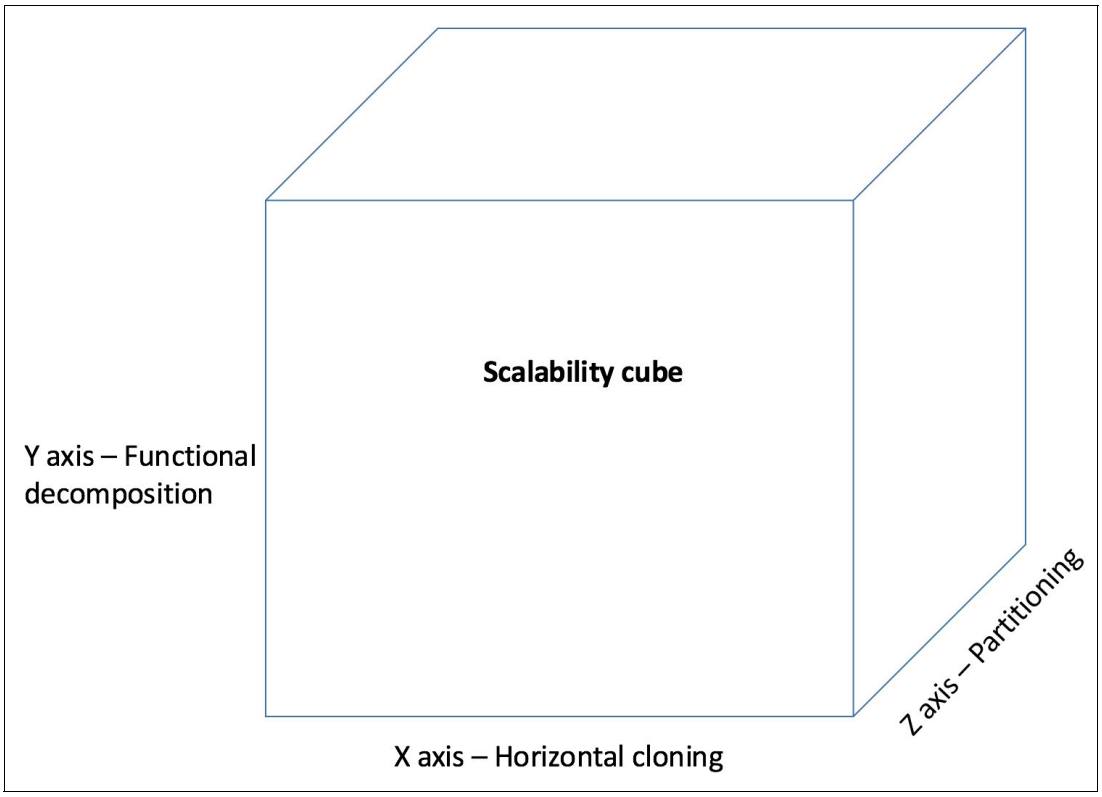


Figure 1-1 Scalability cube

1.2.2 Microservices for various stakeholders

The most important reason for using microservices architecture is to address the challenges that a monolithic model faces. This section discusses how a microservices approach can help solve problems of a monolithic system from the perspective of various stakeholders of the application.

For the business owner

As a business owner, you want your service to be adaptive to new customer and business needs. However, with a monolithic model, making and promoting changes in response to the business needs is slow because of the large code base. It is slow also because of the heavy internal constraints and dependencies among the components and layers.

Microservices architecture principles are built around high flexibility and resiliency. These two characteristics allow changes to be quickly promoted. This helps the business owner receive feedback and adjust the business and investment strategy more quickly to gain customer satisfaction and be more competitive in the market.

From a resources allocation perspective, because teams are smaller and focused, efficiency can be measured and visualized more easily. The business owner can then make investment decisions more easily and can move the resources from low business impact areas to higher business impact areas.

For service management

As a service management team member, you want to have less overhead operation workload to coordinate among teams so that you can improve the productivity of your services. A monolithic model requires much overhead workload. More coordination of activities is needed because a monolithic application usually has a big business scope and many infrastructure and operational touch points. As a result, each change to the application might require many reviews and approvals from different stakeholders. Microservices architecture leverages automation in every stage of the service delivery pipeline in a way that promotes self-service. This helps to minimize coordination of overhead from the service management team.

One important principle in microservices architecture is high observability. High observability provides the service management team the necessary tools to better oversee the status of each microservice in the system or across products. This helps improve service management productivity.

For developers

As a new developer joining a team, you want to become familiar with the source code quickly so that you can start gaining momentum and providing output. A large code base in a typical monolithic application can be an obstacle for you and potentially increase your learning curve. For all developers, a large code base can generate overhead for loading and running in the development environment, which causes lower productivity.

A large code base can also make code reviews and other collaborative development activities more stressful. Furthermore, when working on changes, the risk of breaking something else might cause the developer to hesitate with innovation and enhancing the application.

Microservices, however, are smaller and more lightweight, which can shorten the learning curve for new developers. Microservices can also help eliminate heavy loading and running

www.dbooks.org

issues, encourage the introduction of break-through changes, and, as a result, help increase productivity and innovation.

1.2.3 Timing with availability of cloud era and other tooling

This section discusses why now is the time for microservices architecture to be adopted.

Explosion of cloud environment and offerings

Microservices architecture embodies many advantages to use continuous integration and continuous deployment. The architecture also introduces a new level of complexity and requires a modern approach that embraces automation in every step of building your application. For example, from an infrastructure and governance perspective, the first requirement is a resilient infrastructure for dynamically and quickly spinning up a runtime box for the services. The box might be a virtual machine, a container, and so on. The next requirement is an efficient way to orchestrate and monitor the services. That is what a cloud platform in today's environment like IBM Bluemix can offer with its natural dynamism and resiliency.

With the variety of cloud offerings available for various service models, either infrastructure as a service (IaaS) or platform as a service (PaaS), developers have more options to shift to a microservices strategy. With an IaaS option, you can quickly spin up a machine in a matter of minutes and can wrap the infrastructure provisioning into a set of scripts to automate the process, as needed. If the you do not want to touch the complexity at the infrastructure level, a platform-level option is also available with different languages and as a service to quickly package, and then containand launch the microservice at will.

IBM Bluemix is an example of such platforms. IBM Bluemix has many services that are suitable for building microservices using cloud technologies, such as IBM container, Message hub, logging, monitoring, and other technologies. Bluemix enables developers to rapidly create, deploy, and manage their cloud applications by providing a critical foundation to simplify operations, determine security policies, and manage the health of the microservices in the system.

Availability and maturity of tooling

Besides the dynamism and resiliency that cloud infrastructure can offer to the shift to a microservices strategy, having comprehensive tooling is also a key requirement to the adoption of a microservices strategy. The tooling for microservices has been evolving and advancing through time. In today's environment, developers have many options to use an appropriate set of tooling for their microservices strategy, such as a logging stack, monitoring stack, service registry, or containerization technologies. This advanced tooling can help to address challenges that are introduced by microservices architecture to better deliver, manage, and orchestrate the services.

Figure 1-2 on page 23 shows a full stack example of IBM Watson™ cloud services built on microservices architecture. This evolutionary architecture is powered by cloud technologies, a set of comprehensive tools, and also agile processes.

The architecture contains several major stacks:

- DevOps

  Each Watson cloud service is developed and then containerized inside an immutable virtual machine and then deployed onto IBM SoftLayer cloud infrastructure automatically through a bold DevOps process. Typical patterns of microservices architecture are used, such as service discovery, API gateway, and others, by using both IBM-specific and open source technologies. The services are then made available on IBM Bluemix platform.

- Elasticsearch, Logstash, Kibana (ELK) stack, or Elastic Stack

  Elk stack is the logging stack of the system and contains a set of tools to capture logs and store them into a powerful, centralized, and searchable logs database. For more information, see the following website:

  https://www.elastic.co/webinars/introduction-elk-stack

- Monitoring stack

  Figure 1-2 shows a set of tools that allows monitoring the whole system from a centralized dashboard, including a notification mechanism to send alerts based on a particular event.

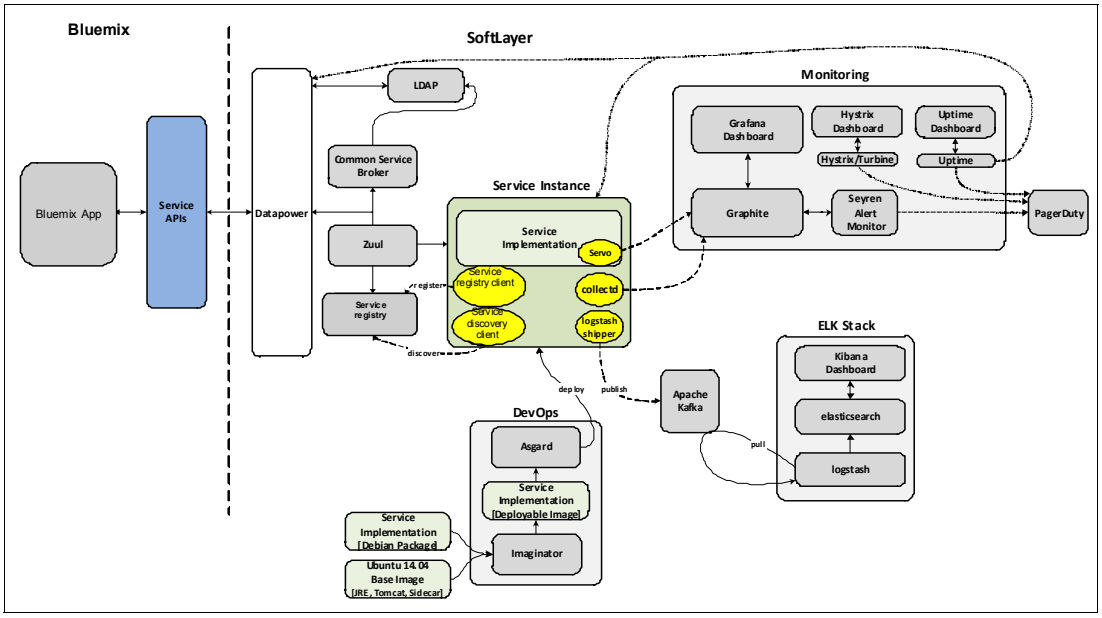


Figure 1-2 An example of tooling for enabling microservices architecture

1.3 Transformation from a monolith to microservices

This section introduces a practical scenario that transforms a monolithic application into microservices.

1.3.1 Fictional Company A business problem

Fictional Company A, an e-commerce company, provides online purchasing services and operates its business using a traditional Java EE-based web application called Customer Order Service. Although the application has been serving the business well, Company A started struggling with responding to new business requirements. These requirements included the following:

- Reaching out to customers who use mobile devices
- improving customer experiences based on insights about their personal behavior on the internet
- Expanding infrastructure to handle more requests from new and existing customers
- Keeping IT costs low
- And other requirements

www.dbooks.org

The current customer order service application is not designed to enable changes in business domain and is not open for applying new technologies for accelerating innovation. Figure 1-3 on page 24 describes the logical architecture overview of the current monolithic application.

Company A wants to transform the customer order service application to embrace and better handle changes in both business and technical perspectives and has a list of major business requirements:

- The new system must be evolutionary, meaning it must be flexible for changes.
- No down time is allowed in moving traffic from the current system to the newly built system.
- The new application must be able to scale on demand, or automatically, based on the payload sent to the system, so that it can react to dynamic shopping behavior patterns.
- The new system must be open for leveraging emerging technologies to embrace innovation.

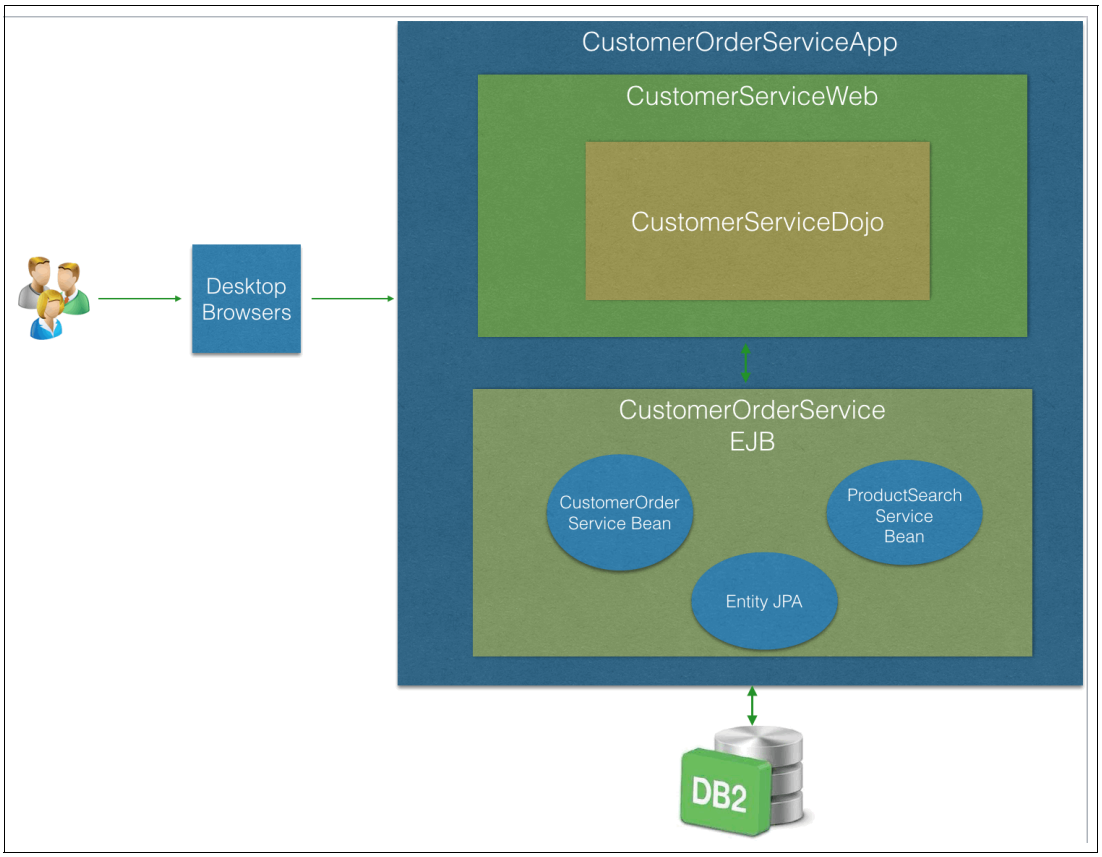


Figure 1-3 Current monolithic application

1.3.2 Adopting microservices for an evolutionary architecture

The major motivation of microservices architecture is to solve problems of traditional monolithic architecture, which is difficult to change. The microservices approach embraces changes to every composing part of the architecture.

Referring to the business requirements, microservices architecture is a good fit for Company A to adopt in building the new system. Company A applies best practices and patterns to

transform the existing monolithic application to a more revolutionary architecture with the intention to eventually migrate the application to microservices architecture over time.

The following major steps and activities are conducted, as discussed in the remaining chapters of this book:

- Evolution strategies

  To have a suitable strategy for transforming the case study monolithic application, different patterns and suggested practices must be discovered and considered. Chapter 2, “Evolution strategies” on page 27 describes possible strategies that can be considered and applied in practice.

- Identifying candidates to be transformed into microservices

  In this step, relatively small components or pieces of functionality of the application are selected. From these pieces, new microservices can be provisioned to make the pieces more conducive to constant or evolutionary changes. Chapter 3, “Identifying candidates within the monolith” on page 35 describes this aspect with practical steps.

- Data access patterns

  Because data is the most important asset in an IT system, a crucial step is to have the correct approach for data access in the transformation to microservices architecture. Chapter 4, “Enterprise data access patterns” on page 51 describes possible data access patterns and how to apply them in this case study example.

- Security and governance

  Chapter 5, “Security and governance” on page 81 describes how components of the application are governed in the more distributed new model and how security challenges are handled. It includes techniques and practical implementation with the case study application.

- Performance and scalability

  When addressing scalability issues of a monolithic application, microservices architecture, with more distributed characteristics, introduces performance challenges. Chapter 6, “Performance” on page 95 describes the two aspects, demonstrated by the case study example.

- DevOps and automation

  Automation is the enabler that makes the microservices approach possible. Chapter 7, “DevOps and automation” on page 103 discusses DevOps together with automation in the IBM Bluemix platform.

www.dbooks.org

2

Evolution strategies

This chapter discusses microservices as an evolutionary architecture and introduces popular patterns to be considered when building a microservices strategy. It also briefly describes how the strategies are implemented in the case study.

This chapter includes the following sections:

- Microservices: An evolutionary architecture by nature
- Evolutionary practices and patterns
- Evolutionary architecture in practice

© Copyright IBM Corp. 2016. All rights reserved.

www.dbooks.org

2.1 Microservices: An evolutionary architecture by nature

Architectures are usually created by following the predefined principles that help to reflect the strategic business goals of an organization. Business models tend to changed little, but in today's environment, organizations want to enable a highly scalable business to serve a greater number of transactions and customers. They need to have flexible operational processes to reach new markets quickly and to be more innovative in existing markets. The common steady set of goals leads to a set of architectural principles that drive the way IT systems are evolutionarily architected, designed, implemented, and operated. One of the most important principles of an evolutionary architecture is the ability to support continual and incremental changes along multiple dimensions in both the technical and business domain spheres.

Traditional architectural elements of software tend to be difficult to change. This means that after decisions are made and components are designed and implemented, making changes to the architecture of the system is not easy. Some evolution has occurred in the way systems are architected with the introduction of more layered architectures and patterns to put those architectures in practice. Figure 2-1 shows an example of a typical layered architecture that enables system evolution.

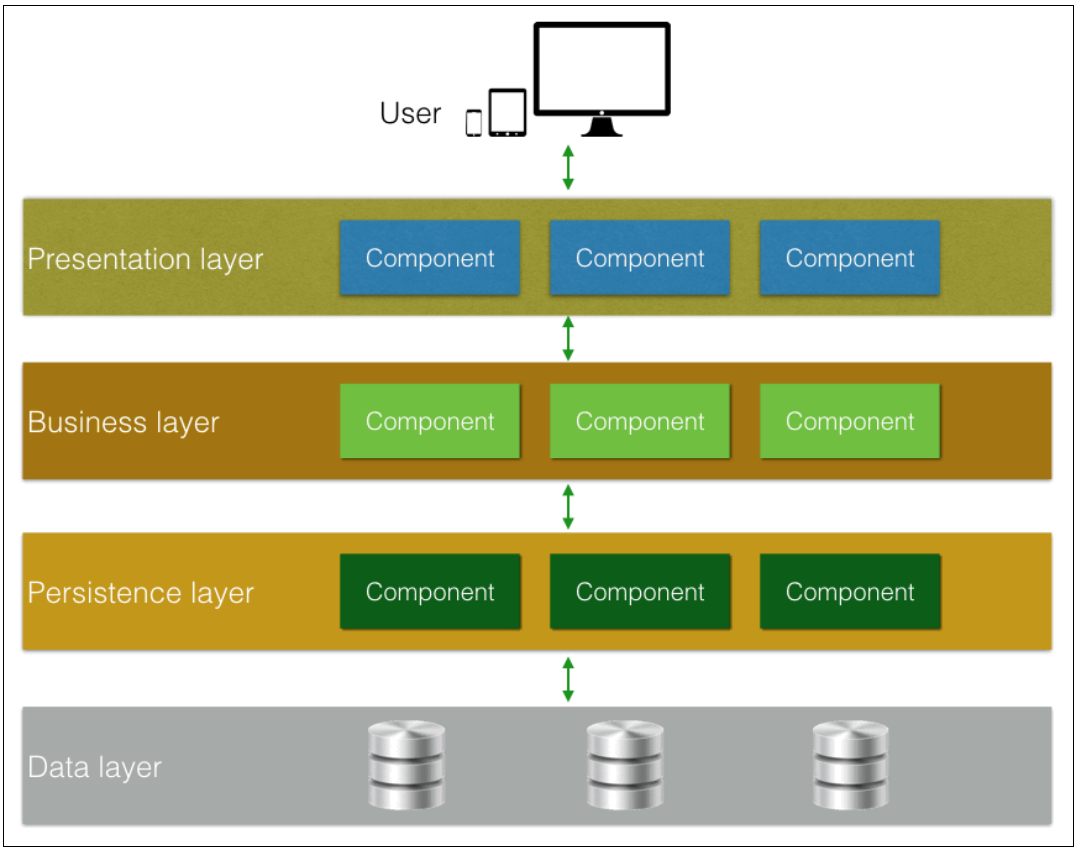


Figure 2-1 An example of a layered architecture

With layered architectures, implementing changes at each layer without impacts on other parts of the system is easy. For example, if you want to change a component in the persistence layer, this change theoretically has little effect on the business and database layers and no effect on the presentation layer. This represents some level of evolution where the system has four opportunities to be evolved over time; each of the four separated layers

can evolve. The evolution, however, is only enabled on one dimension: the technical aspect of the architecture.

When the business domain perspective is added into the architecture, there is no dimension for evolution for that perspective. Implementing changes in business domain can be difficult depending on the change requirement; changes in the business domain usually require touching on different modules or layers of the system. For example, making a change to the customer of the service that the system is serving, including related information and flow, is usually difficult. The difficulty arises because some parts of the customer are in the business layer, and other parts are scattered across the data layer, presentation layer, and other layers.

Microservices architecture is an evolutionary architecture by nature. One of its defining characteristics is that each service forms its own bounded context and is operationally distinct from all other services in the system. The technical aspect of the architecture is, for the most part, encapsulated entirely inside each service in the system. You can easily take one of the services out of the system and put another one into its place without impacting the other services. That is because of the highly decoupling principle that is part of the architectural style. Hence, there is more flexibility with microservices architecture. This flexibility supports an ability to evolve every service in the system.

Figure 2-2 shows an example architecture overview diagram of a typical microservices architecture. Note that API Gateway is not a mandatory requirement for a microservices architecture. It is beneficial, however, to have API Gateway to handle different API requests by routing them to appropriate microservices. If you do use API Gateway, design and implement it carefully to avoid coupling between services. Each microservice can have multiple micro components that might be separated into different layers. Figure 2-1 shows this type of microservice.

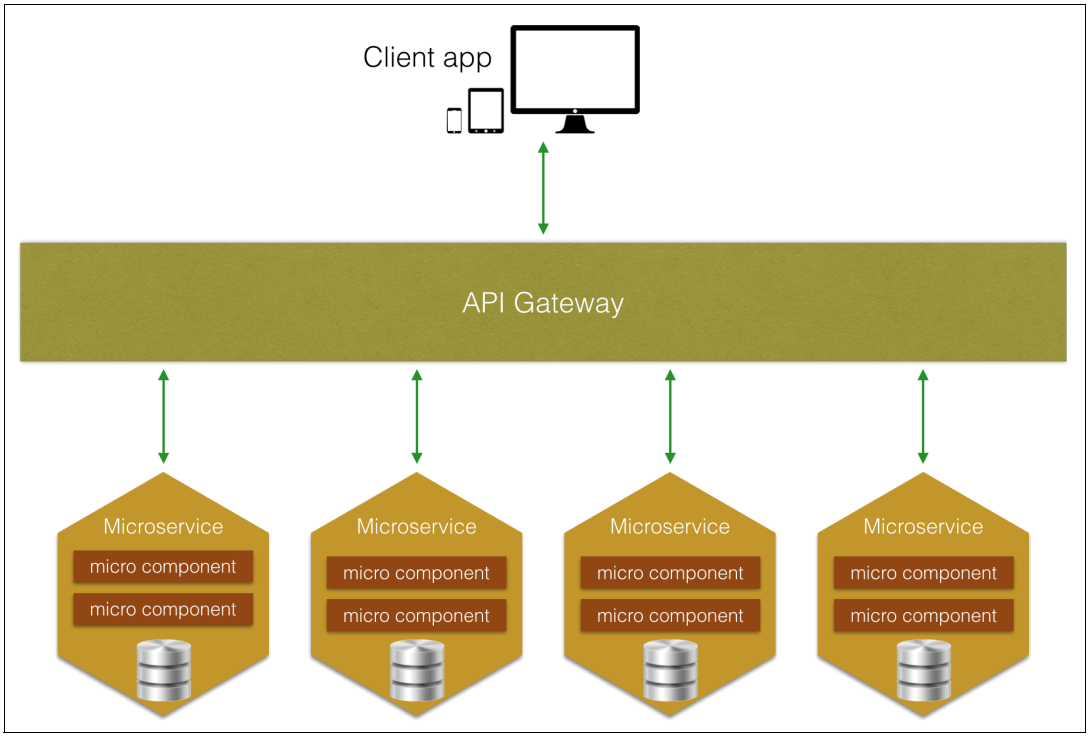


Figure 2-2 More evolvability dimensions in microservices architecture

www.dbooks.org

2.2 Evolutionary practices and patterns

Having an evolutionary architecture is important in building a more flexible business model. This section discusses practices and patterns that can be used as enablers of evolutionary architecture.

2.2.1 Continuous integration, continuous delivery

Continuous integration (CI) is a software development practice in which developers of the software frequently integrate their work, usually daily, resulting in multiple integration times per day. The idea behind this preferred practice is to insulate potential integration problems as early as possible to solve problems that might occur. CI is backed by an automated process that includes building and testing activities to verify every build and detect integration errors quickly. Because potential issues can be isolated and caught earlier, CI helps to significantly reduce integration problems during development cycles and eventually helps to deliver products quickly.

CI is usually accompanied by continuous delivery (CD), which refers to continuously releasing every good build that passes tests into production. By adopting CI and CD, developers can reduce the time spent to fix bugs and increase the effort spent on developing new features or enhancing the existing ones. The adoption of CI and CD creates a higher business value impact.

CI and CD are powered by leveraging automation in every step of the development cycle, including creation of the infrastructure. Automation has evolved due to the availability and maturity of cloud technologies and CI and CD tools. Both CI and CD play a crucial role in the evolution of an architecture.

2.2.2 Strangler application

Martin Fowler introduced the term *strangler application* in his seminal article *Strangler Application* in 2004. In this pattern, a new system captures and intercepts calls to mature applications, routes them to other handlers, and gradually replaces them altogether. Rewriting the whole existing system from the first step is risky; the strangler application method reduces risks more than the cut-over rewrite approach. In moving from monolithic to microservices architecture, you can incrementally refactor your monolithic application. You gradually build a new application that consists of microservices; you then run this application together with the monolithic application. Over time, the amount of functionality handled by the monolithic application is incrementally decreased until either it disappears entirely or becomes part of the new microservices. Figure 2-3 on page 31 describes the transformation using the strangler application pattern.

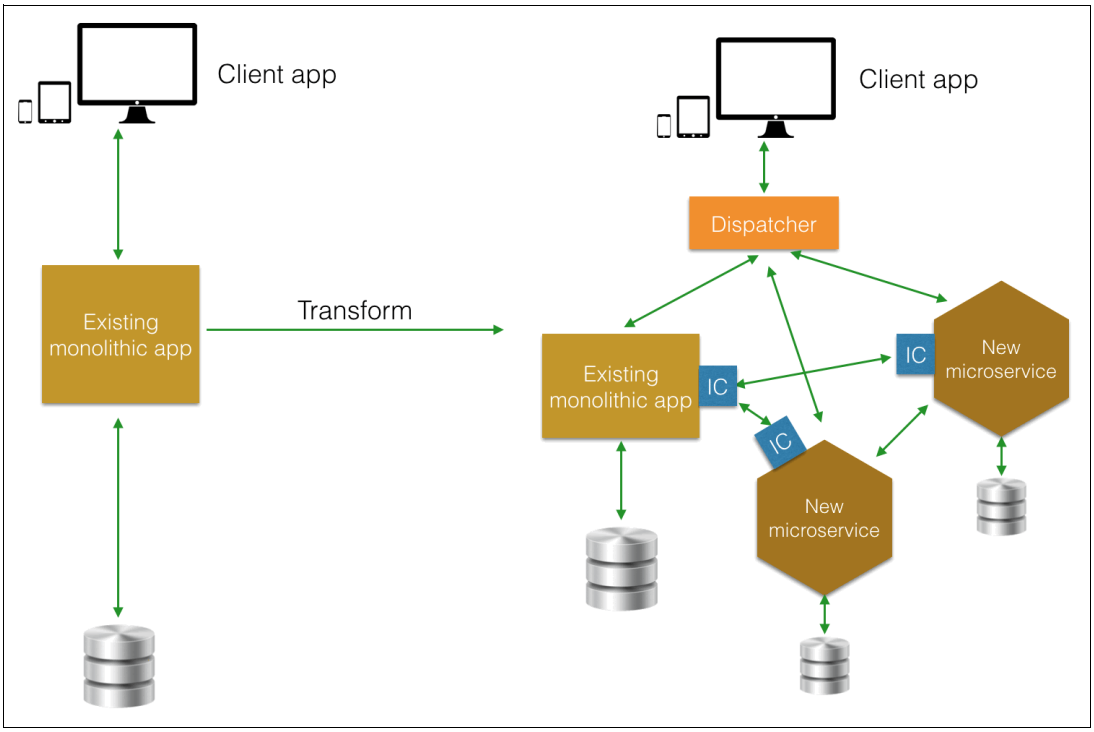


Figure 2-3 Incrementally strangling an application

Using the strangler application pattern, the resulting architecture has both the earlier monolithic application and the new services. The architecture also has two other major components:

- Dispatcher: This component works as a requests router that handles incoming requests by routing the requests corresponding to new functionality to new services. It routs previous requests to the existing monolithic application.
- IC: These components reside in either the new services, the monolithic application, or both. They are responsible for integrating the monolithic application with the newly created services. You might not totally replace portions of the monolithic application; you might wrap them as a system of records that resides as a set of application programming interfaces (APIs).

One principle in an evolutionary architecture is to be well prepared for the unknown, that is, to be proactive rather than predictive about the movement in the future. Because of the natural evolution, what you build today becomes old and strangled in the future. Therefore, design the new application in such a way that a graceful fading away of the mature application occurs.

2.2.3 Evolutionary database refactoring and deployment

Often, a database (more specifically, data in the database) is the most important asset in an IT system. When your application is evolving, that asset must also be changed to be aligned with the application code changes. The database changes here include both database structural changes and data migration from the old structure to the new one. You might even have to consider moving between different types of databases. For example, moving some parts or the whole database from a relational database management system (RDBMS) to a NoSQL style. The refactoring and migration of data in that situation is even more challenging. Unfortunately, the traditional tools and processes for updating the database are not innovated

www.dbooks.org

and mature enough to keep up with the fast evolving pace in application development methodologies. This situation can be an obstacle for releasing the application rapidly.

A new approach to database change management and deployment that is powered by automation is required to make continual and incremental changes in an evolutionary architecture like microservices architecture. This section discusses several techniques as suggested practices to make database changes more evolved and aligned with application code changes. In their book *Refactoring databases: Evolutionary database design,* Scott Ambler and Promod Sadalage describe 13 lessons that might be used as guidance when building your database refactoring strategy. Those lessons provide the foundation for three steps of data refactoring: Decomposing big change into small changes, version controlling the changes, and automating database deployment.

Decompose big change into small changes

If you have a big database change to be applied, divide it into a smaller changes; make them small, individually testable changes. Each change is usually a combination of database structural change, data migration, and associated access code. In this way, you can quickly isolate any possible failure that the changes might produce in those small chunks and subsequently react and correct them. The purpose of this decomposition is also to map the database changes into application features that use the changes. This allows the modifications to be incrementally and continuously deployed altogether in the same agile process.

Version control the changes

Version control starts with organizing your database change scripts as necessary. Ensure that the change scripts are stored in the same repository as the application source code. Also ensure that you have the same directory structure and naming convention with the code. In this way, you can align the database changes with a particular application modification in a particular release that requires the changes. After the alignment is structurally in place, apply version control on the database change scripts. This process helps to facilitate the deployment of database changes and provides common ground for the database team as they coordinate and collaborate with other teams.

Automate database deployment

Automation is the key to managing and deploying evolutionary database changes. Ensure the alignment between the application code changes and the database changes are in place. Do this by applying the decomposition at the appropriate granularity, reorganizing the database changes script, and applying version control. When this is done, you are ready for a more evolutionary-enabled database deployment approach that is powered by automation. You then can adopt database deployment automation in a similar way with application code deployment by using agile CI and CD practices.

2.3 Evolutionary architecture in practice

This section describes how to apply an evolutionary approach to building a more flexible architecture that enables changes over time. The approach is demonstrated with a case study example application for Fictional Company A. Technical architectures are usually driven by strategic business needs. Chapter 1, “Overview of microservices” on page 15 describes the business requirements to help you start building an evolution strategy for the monolithic application.

The following basic steps serve as a guideline for the transformation:

1. Start the evolution strategy with product data: Customers are having trouble finding product data on their site. The goal is to make the data available and present it in a more interactive way. Moving data into an Elasticsearch database might provide customers the ability to do more powerful searches. This can be the starting point to build a new microservice on the cloud, so consider moving and migrating data to a NoSQL model, establishing new indexing, and deploying to a cloud environment.
2. Continue with an account service by getting more information about the customer to personalize the content that is provided for them. This personalization can improve the customer experience. You can achieve this requirement by combining the existing customer data with social data, resulting in a potential new JSON-based data model.
3. Move part of the customer data to the cloud. Consider a hybrid cloud integration pattern to address the data synchronization issue between the earlier data source and the newly created data source.
4. Optimize the user interface (UI) layer to better serve users. Start with the supporting mobile platforms, then incrementally update the UI by using the microservices approach based on the new data model.
5. Ordering is the last part to transform and move. You can start the transformation to microservices by putting a set of APIs around the current ordering component to add more flexibility and decoupling. This allows you to be ready for future steps.

2.3.1 Gradually transforming to a catalog service

Information about products that Company A provides is the core data set of the company's business. Making the data reachable in the easiest and most interactive way possible to the customers is a critical factor in architecture success. Product-related components are good candidates to start the evolution strategy. Resulting artifacts might be a new microservice running in a cloud environment for resiliency and scalability. In this example, Catalog service is the name of the new service. Because the products logics are moved completely to the new microservice, data must also be gradually migrated to the same cloud platform. This eliminates any potential latency when the new service needs to make calls to the database for populating data.

Technologies for implementing the new catalog are numerous. However, in considering an evolutionary approach, also consider keeping the same technology stack that is used in the monolithic application, which is Java based, to avoid carrying big changes at the same time. The technology stack can be changed later in further steps based on business needs and the availability and maturity of a particular technology available in the market at the time that the architectural decisions are made. A cloud platform such as IBM Bluemix offers various options to quickly deploy the service code into a runtime instance. Data for the newly created Catalog service can also be kept in a relational database management model at first to avoid any big changes.

The next step is to migrate the data to a more modern lightweight NoSQL and JSON model to improve the search experience. Chapter 4, "Enterprise data access patterns" on page 51 describes data migration strategy and implementation.

2.3.2 Strangling the customer and order components

Business continuity is important to Company A, therefore, the transformation must be implemented carefully so that no interruption of services occurs during the transform process. To mitigate the risks of system down time, use a strangler application technique.

www.dbooks.org

Customer-related business components and data are decomposed and transformed into a more evolutionary new model, but the old components continue to be operated in parallel at the same time. The traffic to old customer and order components are incrementally moved to newly created services.

2.3.3 Evolutionarily transforming the user interface to a UI microservice

One of the business requirements of Company A is to expand the system to reach out to customers who use mobile devices. The customers in this context include existing and potential new customers. To fulfill the requirement, the following major steps are conducted as a guideline to transform the user interface to the microservices architecture:

- Change the CustomerServiceDojo module to support mobile users and set a step toward a responsive design approach.
- Use responsive design-enabled client-side libraries (for example, Bootstrap and Angular) to improve the user experience and make the UI more modular and easier to change.
- Start moving the UI layer into the cloud environment for more rapid development, deployment, and adaptability to change.

3

Identifying candidates within the monolith

This chapter introduces how to identify candidates within the monolith applications for microservice architecture and practices. It also describes how to design and refactor the new components.

This chapter includes the following sections:

- Identifying candidates
- Considerations for moving to microservices
- Decomposing monolith application into microservices
- Refactoring
- Identifying and creating a new architecture example

© Copyright IBM Corp. 2016. All rights reserved.

www.dbooks.org

3.1 Identifying candidates

Candidates for the evolution to microservices architecture are monolith applications with components that cause any of these situations:

- You are unable to deploy your application fast enough to meet requirements because of difficulty maintaining, modifying, and becoming productive quickly, which results in long cycles of time-to-market to roll out new services. Perhaps the business requires a new function to take advantage of a new market opportunity, or you want to link into a new social media service. In either case, you are unable to build, test, and deploy your application in a timely manner.
- You are unable to apply a single deployment. Usually it is necessary to involve other modules or components to build, test, and deploy, even for a small change or enhancement, because there is not separation of modules and components inside the same package of `.ear` or `.war` files.
- Only a single choice of technology exists and you cannot take advantage of new technologies, libraries, or techniques that are being adopted by other enterprises. This can manifest itself in several ways. Perhaps your current technology stack does not support the functionality you need. For example, you want to generate documentation automatically from your code, or you want to link to a new service, but the current technology used in your system or platform does not offer these features.
- Your large systems have the following characteristics:
  - High amount of data in memory
  - High-use CPU operations
  - Unable to scale a portion of the application; usually the entire application must be scaled
  - Cannot be easily updated and maintained
  - Code dependencies are difficult to manage
- Onboarding new developers is difficult because of the complexity and large size of the code.

3.2 Considerations for moving to microservices

Consider the following important aspects before you move to microservices.

- Flexibility for the development platform

  Microservices offer the flexibility to choose the correct tool for the job; the idea is that each microservice can be backed by a different technology (language and data store). Different parts of a single system might be better served with differing data storage technologies, depending on the needs of that particular service. Also, similarly, a microservice can use different languages; this enables you to choose the preferred technology for a particular problem rather than following the standard.

- Design by functionality and responsibilities

  Microservices offer separation of components and responsibilities: each microservice is responsible for a determinate topic. In this way, you can apply new features or improvements in specific parts of the system, avoiding mistakes and avoiding damaging a part of the system that is working. This approach also can be used for isolating older functionality; new functionality can be added to the monolith system by adding a microservice.

- Easily rewrite complete services

  Usually microservices are small services that are easy to rewrite, maintain, and change, providing a strong encapsulation model and a safe way to refactor and rewrite.

- Flexibility for changes

  With microservices, you can defer decisions and have flexibility in the growth of your architecture as your understanding of the system and domain increases. You can defer difficult decisions, such as decisions about data storage technology, until they are absolutely required. For example, the only function that a service provides is a thin domain-aware wrapping over a data store, but the most important function to get correct is the interface with which your other services interact.

- Driving business value

  Business owners see the benefit of microservices because they want their teams to be able to respond rapidly to new customer and market needs. With the monolithic application development approach, IT response is slow. Microservices are more aligned to business because they allow for frequent and faster delivery times than monolithic services. Microservices allow business owners to get feedback quickly and adjust their investments accordingly.

  Others benefits are as follows:

  - Smaller focused teams enable business owners to easily manage resources more effectively, for example, moving resources from low-impact business areas to higher-impact areas.
  - Scaling an individual microservice for removing bottlenecks enables a smoother user experience.
  - Identifying and eliminating duplicate services reduces development costs.

- Flexibility regarding scalability

  With microservices, different parts of the system can be scaled; each microservice is responsible for specific functionality, which results in more flexible scalability.

- Security zoning

  Security architects insist on a layered approach to building a system to avoid the risk of having important code running on web-facing servers. Microservices can provide zoning that is analogous to the traditional layered approach. Business logic and critical data storage can be separated from the services that provide HTML rendering. Communication between the individual microservices can be firewalled, encrypted, and secured in other ways.

- Team skills

  With microservices, a team can be grouped by skills or location without the associated risks or difficulties involved with having separate teams working on the same code base.

3.3 Decomposing monolith application into microservices

This section describes techniques for decomposing a monolith application into a microservice.

www.dbooks.org

3.3.1 Designing microservices

One of the great advantages of microservices architecture is the freedom to make decisions about technology stack and microservice size on a per-service basis. This freedom exists because microservices architecture starts by having a clear understanding of the user's experience.

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, microservices instead focus on the user experience (that is, what the users are thinking, doing, and feeling as they have the experience). Design thinking is helpful for scoping work to usable and releasable units of function. Designs help to functionally decompose and identify microservices more easily. Design thinking includes the following concepts:

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

Hills

Hills are statements that provide a business goal for your release timeline. A Hill defines who, what, and how you want to accomplish the goal. 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 smooth user experience. The Hills definition must be targeted at a user, and it must be measurable. Avoid using vague or non-quantifiable adjectives when using Hills. Example 3-1 shows a sample Hill.

Example 3-1 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 intends to target. Hills Playback sets the scope for a specific release time period. Playback Zero 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 3-1 on page 39 shows a Hills Playback timeline.

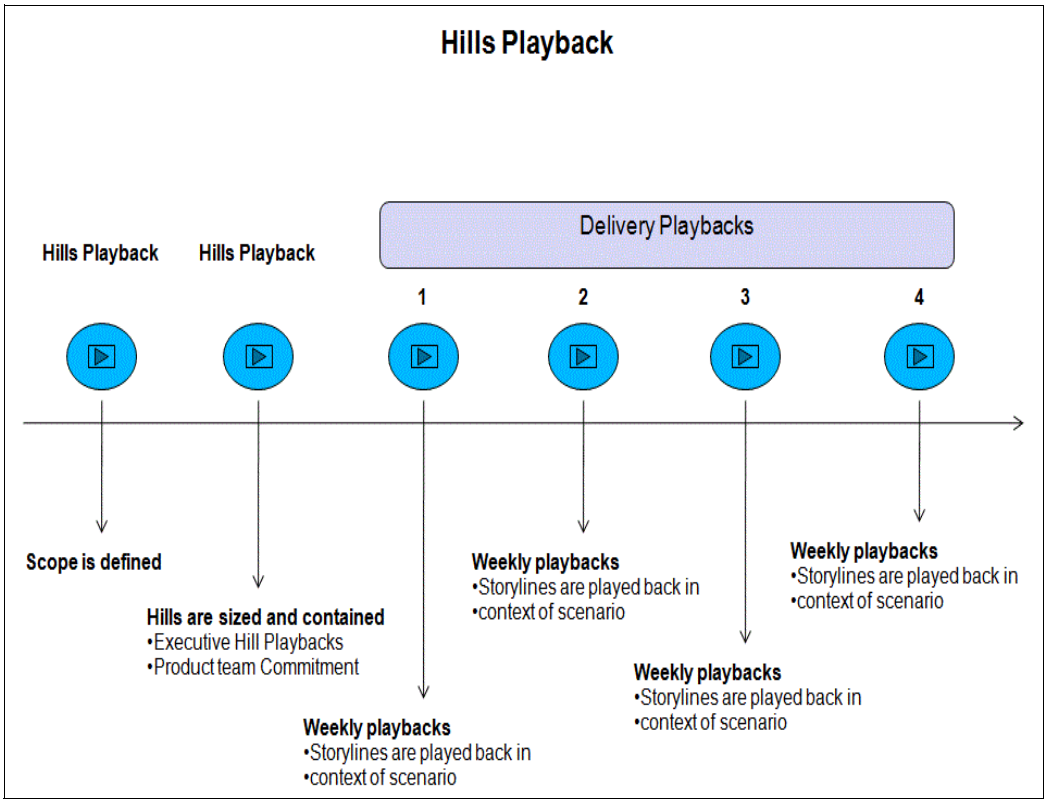


Figure 3-1 Hills Playback timeline

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. The user story is expressed in terms of the user experience, for example, “As a developer, I want to find samples and quickly deploy the samples 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 are not repeated.

Sponsor users

Sponsor users are users who are engaged throughout the project to represent target personas for the 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 the opportunities for potential reuse of a service across other designs. The Hill definition described in “Hills” on page 38 is targeted at an iOS solution with Bluemix.

www.dbooks.org

Based on the Hill's example, you have the following requirements:

- Ability to deploy to Bluemix and other websites
- Ability to deploy to Bluemix as a separate microservice
- Ability to deploy to a Bluemix microservice team

The Deploy to Bluemix microservice team is responsible for implementing the service. The team is also responsible for performing functional and system integration testing of the stories in which that microservice is used. The service includes logging data to collect usage information. This helps to better quantify the microservices effect on the business. It provides visibility into the most popular applications being deployed and into user acquisition after users deploy a sample.

3.3.2 Choosing the implementation stack

Because microservices systems consist of individual services running as separate processes, the expectation is that any competent technology that is capable of supporting communication or messaging protocols works. The communications protocols might be, for example, HTTP and REST; the messaging protocols might be MQ Telemetry Transport (MQTT) or Advanced Message Queuing Protocol (AMQP). You must consider several aspects, though, 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, such Java Message Server (Java EE JMS) handle requests using an event loop that is often single-threaded, yet can process many more requests when handling them requires downstream input and output (I/O) operations.

- I/O versus processor (CPU) bound

  Solutions such as Node.js work well 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, doing one of the following actions is preferred:

  - Offload long-running operations to a set of workers that are written in a stack that is best suited for CPU-intensive work (for example, Java, Go, or C).
  - Implement the entire service in a stack capable of multithreading.

- Memory and CPU requirements

  Microservices are always expressed in plural because you run several of them, not only 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. However, Java EE stacks, such as IBM WebSphere Liberty, mitigate this problem. Again, stacks such as Node.js and Go are a go-to technology because they are more lightweight and require less memory and CPU power per instance.

In most systems, developers use Node.js microservices for serving web pages because of the affinity of Node.js with the client-side JavaScript running in the browser. Developers use

a CPU-friendly platform (Java or Go) to run back-end services and reuse existing system libraries and toolkits. However, there is always the possibility of trying a new stack in a new microservice without dragging the remainder of the system through costly rework.

3.3.3 Sizing the microservices

One of the most frustrating and imprecise tasks when designing a microservices system is deciding on the number and size of individual microservices. There is no strict rule regarding the optimal size, but some practices have been proven in real-world systems.

The following 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 might want to break up a microservice that is physically too large. Large services are difficult to work with, difficult to deploy, and take longer to start and stop. However, be careful not to make them too small. When microservices are too small, frequently referred to 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 (that is, do one thing and do it correctly), it is best to start with larger services. You can always split one service into two later.

- Too many responsibilities

  A service that is responsible simultaneously for different subjects might need to be split up because usually it is difficult to test, maintain, and deploy. Even if all of the responsibilities are of the same type (for example, REST endpoints), there might be too many responsibilities for a single service to handle.

- Service type

  A good rule is that a microservice does only one thing, for example, one of the following tasks:

  – Handle authentication
  – Serve several REST endpoints
  – Serve several web pages

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

- Bounded context separation

  This technique 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 there are few links to a sever that can be turned into microservices. If a microservice needs to talk to 10 other microservices to complete its task, that can be an indication that the division was made in an incorrect place in the monolith.

- Team organization

  Many microservices systems are organized around teams that are responsible for writing the code. Therefore, microservice partition follows team lines to maximize team independence.

www.dbooks.org

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. The expectation, therefore, is that microservice numbers and size are dictated by organizational and technical principles.

A well-designed microservices system uses a combination of these techniques. These techniques require a degree of good judgment that is acquired with time and experience with the system. Until that experience is acquired, start in small steps with microservices that might be on the larger size (more like mini-services), until more “fault lines” have been observed for subsequent subdivisions.

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

3.4 Refactoring

Refactoring is a practice that modernizes the application and gains resources provided by new structures and platforms. The migration of a monolithic application to microservices follows the same path. Refactoring adds microservices to an application without changing the purpose of the application.

This section describes several techniques for refactoring a monolith application to microservices; it is largely based on the following IBM developerWorks® article by Kyle Brown:

https://www.ibm.com/developerworks/cloud/library/cl-refactor-microservices-bluemix-trs-1/index.html

3.4.1 A reason to refactor to microservices

Starting over with new runtimes and programming languages is costly, especially when much of the code is developed in Java and still works. Refactoring using microservices is a more cautious approach because you can keep the old system running and move the monolithic application in parts to a more sustainable and current platform.

3.4.2 Strategies for moving to microservices

Moving a monolith application to microservices can involve these strategies:

- Convert the whole monolith application to microservices

  The construction of a new application based on microservices from scratch can be the best option. However, because this approach can involve too much change at one time, it is risky and often ends in failure.

- Refactor gradually

  This strategy is based on refactoring the monolith application gradually by building parts of the system as microservices running together with the monolith application. Over time the amount of functionality provided by the monolith application shrinks until it is completely migrated to microservices. This is considered a careful strategy. Martin Fowler refers to this application modernization strategy as the *strangler application*.

http://www.martinfowler.com/bliki/StranglerApplication.html

Others aspects of this approach are as follows:

- Do not add new features in the monolithic application; this prevents it from getting larger. For all new features, use independent microservices.
- Create microservices organized around business capability, where each microservice is responsible for one topic.
- A monolithic application usually consists of layers, such as presentation, business rules, and data access. You can create a microservice for presentation and create another microservice for business access data. The focus is always on functionality and business.
- Use domain-driven design (DDD) Bounded Context, dividing a complex domain into multiple bounded contexts and mapping out the relationships between them, where a natural correlation between service and context boundaries exists.

3.4.3 How to refactor Java EE to microservices

Consider these important aspects when moving Java Platform and Java EE applications to microservices:

- Repackaging the application (see Figure 3-2 on page 44)

  Use the following steps:

  a. Split up the enterprise archives (EARs).

     Instead of packaging all of your related web archives (WARs) in one EAR, split them into independent WARs. This might involve some minor changes to code, or more likely to static content, if you change application context roots to be separate.

  b. Apply the container-per-service pattern.

     Next apply the container-per-service pattern and deploy each WAR in its own application server, preferably in its own container (such as a Docker container or a Bluemix instant runtime). You can then scale containers independently.

  c. Build, deploy, and manage each WAR independently.

     After they are split, you can manage each WAR independently through an automated DevOps pipeline (such as the IBM DevOps Pipeline Service). This is a step toward gaining the advantages of continuous delivery.

www.dbooks.org

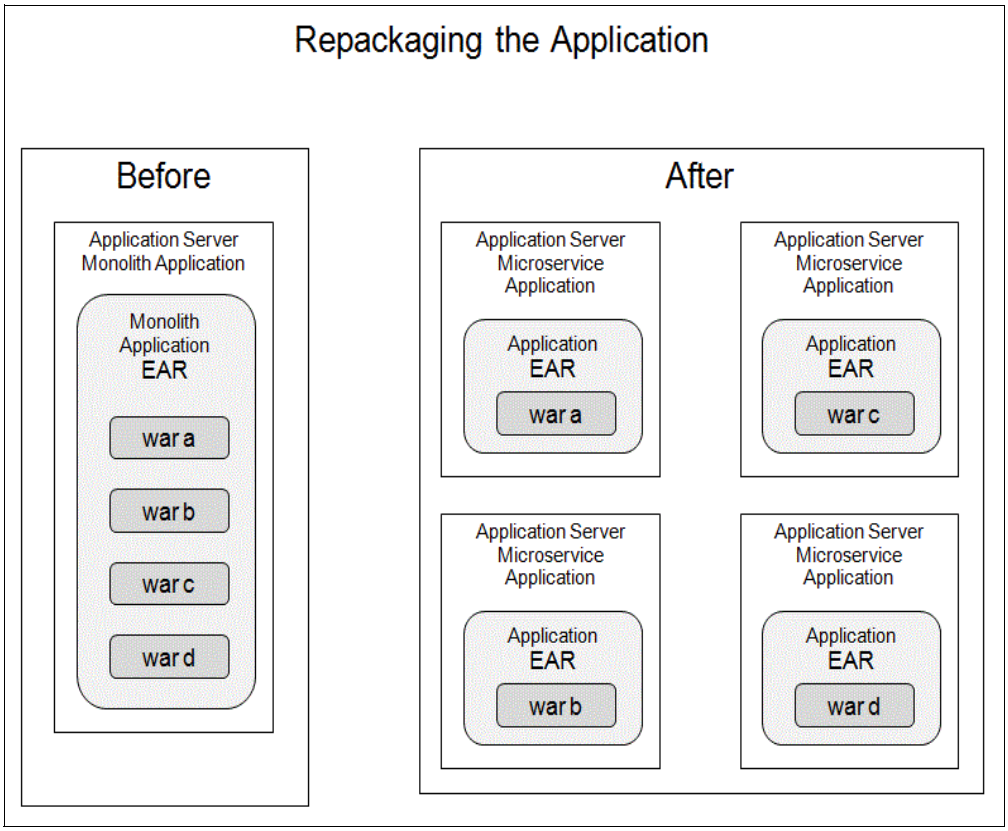


Figure 3-2 Repackaging the monolith application

Note: For more information about repackaging the monolith applications, see the following website:

https://www.ibm.com/devops/method/content/code/practice_refactor_microservices/

- Refactoring the monolith code (see Figure 3-3 on page 45):

  After repackaging, where your deployment strategy is down to the level of independent WARs, you can start looking for opportunities to refactor:

  – Front-end

    For simple program servlets/JSPs that are usually front ends to database tables, create a domain layer that you can represent as a RESTful service. Identifying your domain objects by applying domain-driven design can help you identify your missing domain services layer. After building, in the next phase you can refactor your existing servlet/JSP application to use the new service or build a new interface using JavaScript, HTML5, and CSS, or as a native mobile application.

    HTTP session state: In this case, a good approach is moving the HTTP session state to a database.

  – SOAP or EJB services

    Create the mapping to a RESTful interface and re-implement the EJB session bean interface or JAX-WS interface as a JAX-RS interface. To do this, you might need to convert object representations to JSON.

  – REST or JMS services

    You might have existing services that are compatible, or can be made compatible, with a microservices architecture. Start by untangling each REST or simple JMS service

from the remainder of the WAR, and then deploy each service as its own WAR. At this level, duplication of supporting JAR files is fine; this is still a matter of packaging.

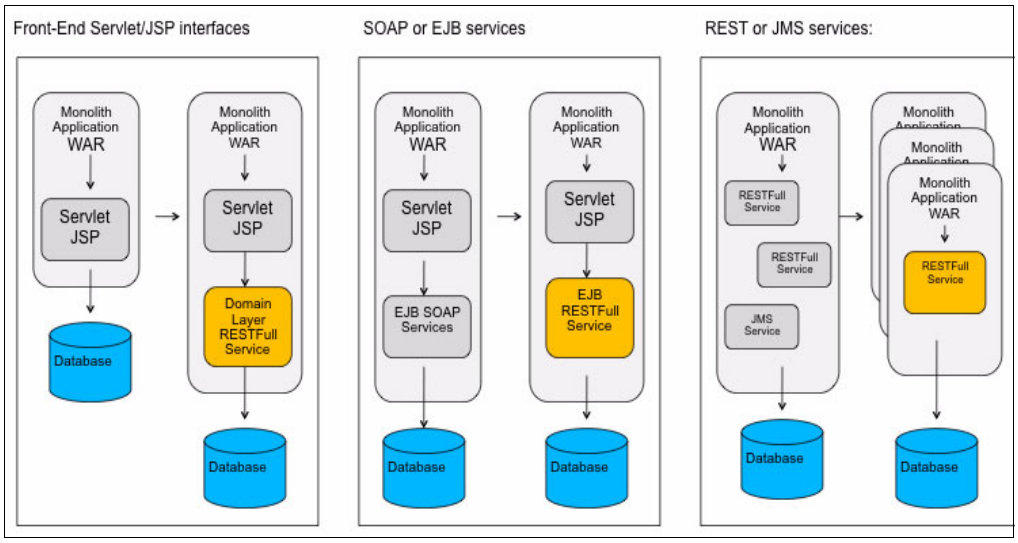


Figure 3-3 Refactoring monolith code overview

- Refactoring monolith data

  After you build and repackage the small services defined in the previous steps, the next step might be the most difficult problem in adopting microservices. That step is to create a new data model for each microservice based on the current data model.

  Here are rules to follow:

  – Isolated islands of data (see Figure 3-4 on page 46)

    Begin by looking at the database tables that your code uses. If the tables used are either independent of all other tables or come in a small, isolated *island* of a few tables joined by relationships, you can split those out from the rest of your data design.

    For defining which database to use, verify the types of queries that you want to use. If most of the queries you use are simple queries on primary keys, a key-value database or a document database might be the best option. However, if you do have complex joins that vary widely (for example, the queries are unpredictable), staying with SQL might be your best option.

  – Batch data updates

    If you have only a few relationships and you decide to move your data into a NoSQL database anyway, consider whether you only need to do a batch update into your existing database. Often, when you consider the relationships between tables, the relationships do not consider a time factor; they might not always need to be up to date. A data dump and load approach that runs every few hours might be sufficient for many cases.

  – Table denormalization

    If you have more than a few relationships to other tables, you might be able to refactor (or in database administrator terms, *denormalize*) your tables.

    Often, the reason for using highly normalized schemas was to reduce duplication, which saved space, because disk space was expensive. However, disk space is now inexpensive. Instead, query time is now what you must optimize; denormalization is a straightforward way to achieve that optimization.

www.dbooks.org

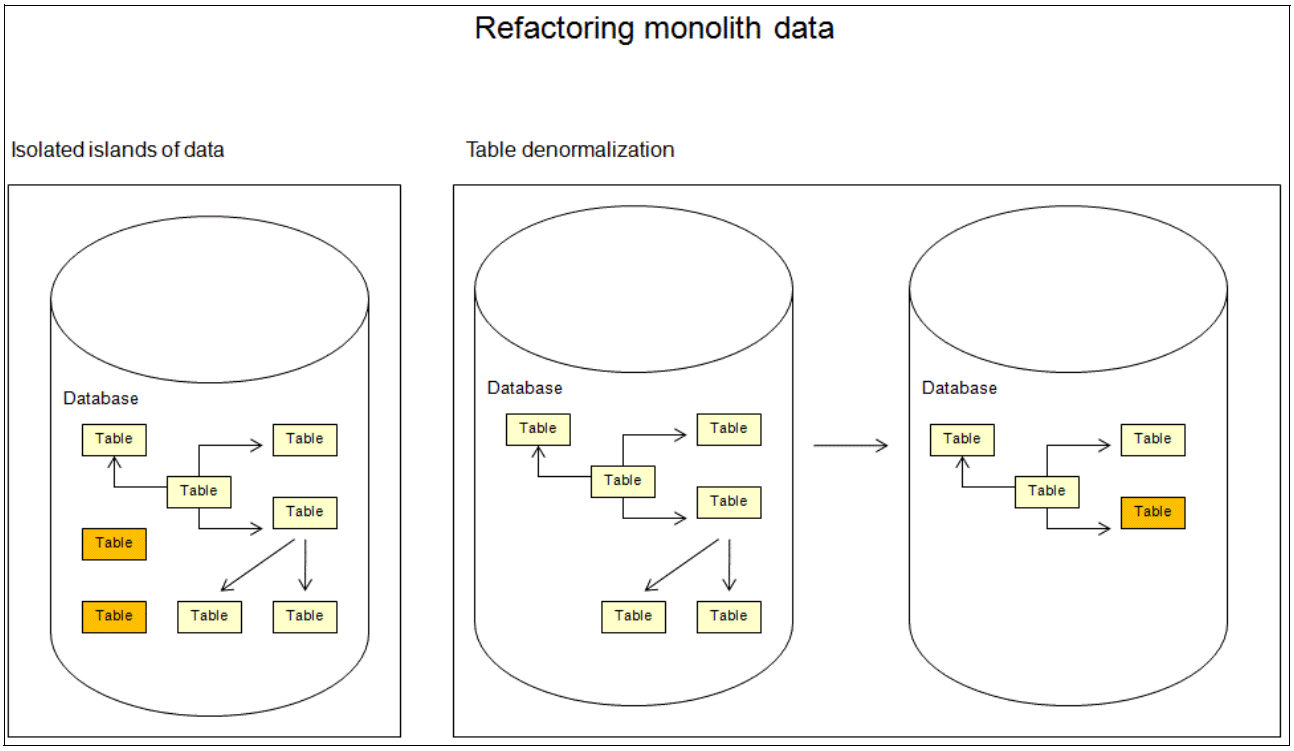


Figure 3-4 Refactoring monolith data overview

For more information, see Chapter 4, “Enterprise data access patterns” on page 51.

3.5 Identifying and creating a new architecture example

This section defines a new architecture based on the business case and monolith application described in 1.3.1, “Fictional Company A business problem” on page 23. This section follows the theory in 1.3.1 and provides the reasons and benefits for each new service.

3.5.1 Architecture: As is

The current application is a classic Java monolith application, defined in one EAR package and running inside an application server. The EAR package contains specific components for each part of the application, usually using the layers pattern. The current application uses JSF and Dojo for the front end, EJB for the business component, and JPA for persistent component and access (DB2® and SQL), as shown in Figure 3-5 on page 47.

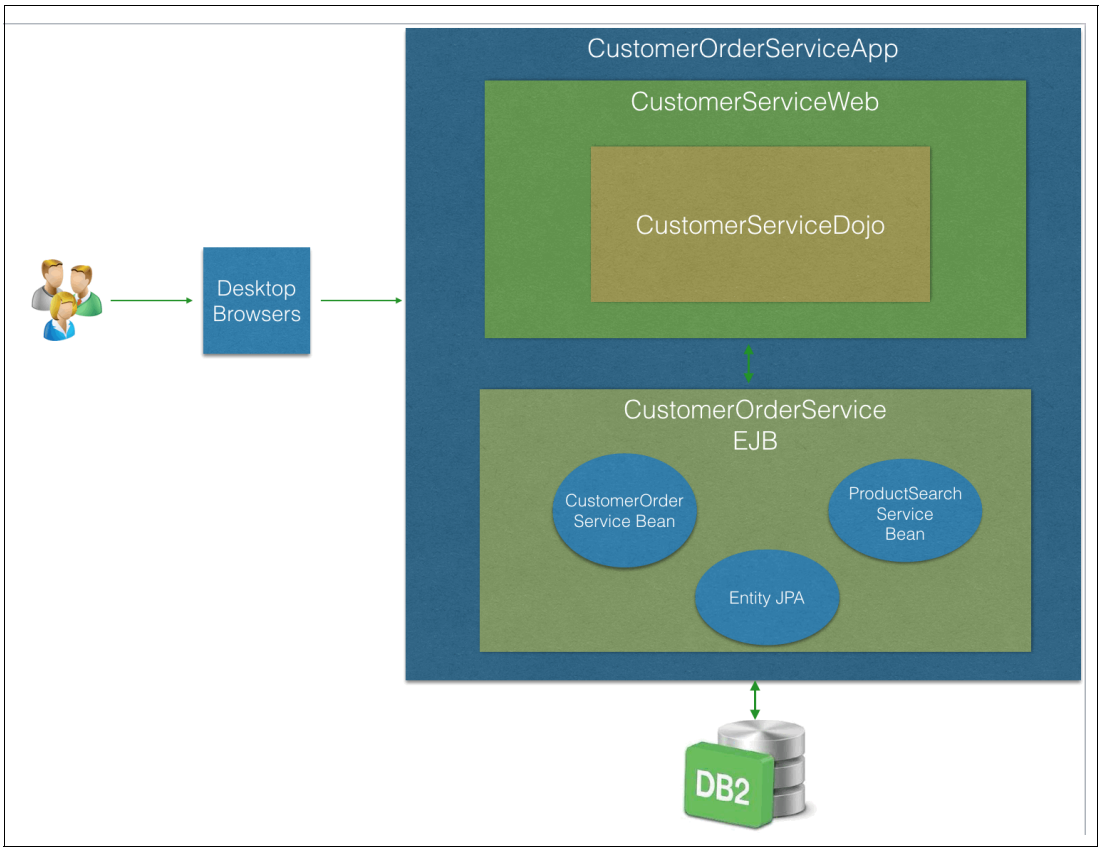


Figure 3-5 Monolith application architecture overview

3.5.2 Architecture: To be

The new architecture is based on the platform as a service (PaaS) cloud environment. The architecture uses services to improve control in the development cycle (DevOps). It also uses other availabilities such as database, runtime, security, application server, and others to improve new features for the new application. It also offers more isolating scalability for each service.

Using gradual migration, the Catalog and Account components are moved to microservices, and the Order component is kept in the monolith application (see Figure 3-6 on page 48).

www.dbooks.org

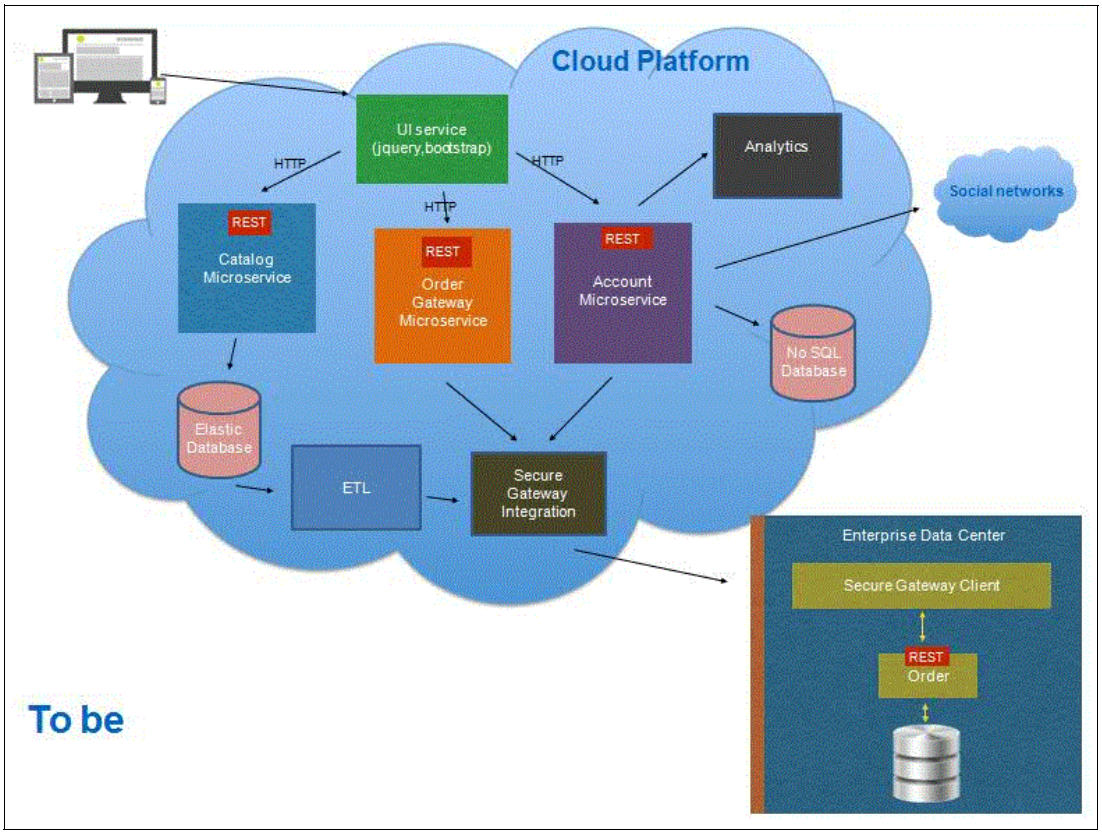


Figure 3-6 Microservice architecture overview

Details of each component are as follows:

- PaaS platform

  The PaaS provides a cloud-based environment with everything required to support the complete lifecycle of building and delivering web-based (cloud) applications, without the cost and complexity of buying and managing the underlying hardware, software, provisioning, and hosting.

  PaaS offers the following benefits:

  - You can develop applications and get them to market faster.
  - You can deploy new web applications to the cloud in minutes.
  - Complexity is reduced with middleware as a service.

- Security gateway

  A secure web gateway is a type of security solution that prevents unsecured traffic from entering an internal network of an organization. It is used by enterprises to protect their employees and users from accessing and being infected by malicious web traffic, websites, viruses, and malware. It also ensures the implementation and compliance of an organization's regulatory policy. In the new architecture, this component is used for integration with database SQL from a monolith application. In the new architecture, this service also is responsible to ensure integration with integrity and security with the monolith application.

  For more information, see to Chapter 5, "Security and governance" on page 81.

- Five distinct applications

  These applications are as follows:

  - UI application using jQuery, Bootstrap, and AngularJS

    A new front-end application using technologies and frameworks that enable running in different gadgets (mobile, tablet, desktop) is created. The necessary business information (Account, Catalog, and Order) is retrieved by using a REST API from the microservices.

  - Microservice application for Catalog

    The component responsible for the Catalog component is moved to a microservice, becoming an isolated application for enabling better scalability. Also, a new search service is added using elastic technology to the microservice to improve search inside each element.

    The catalog is updated using extract, transform, and load (ETL), which gets data from the SQL monolith application using the gateway service for integration.

    For more information, see Chapter 4, “Enterprise data access patterns” on page 51.

  - Microservice application for Order

    The Order component is kept in the monolith application, but a microservice is created to get Order information from the monolith application and uses a gateway service for integration.

  - Microservice application for Account

    The component responsible for the Account component is moved to a microservice, becoming an isolated application for enabling better scalability. Also, a new NoSQL database is added for the Analytics service and integration with social networks. These new features improve potential for collecting information to use for offering new products.

    For more information, see Chapter 4, “Enterprise data access patterns” on page 51.

  - Enterprise Data Center

    The migration of a monolith application is partial. In this phase, the Order feature is kept in the monolith application. A new API interface is added to the monolith application to offer access to the Order information.

    For more information, see Chapter 4, “Enterprise data access patterns” on page 51.

See Appendix A, “Additional material” on page 119 for information about accessing the web material for this publication.

www.dbooks.org

4

Enterprise data access patterns

This chapter describes the key database topics to consider when you evolve a monolith application to a microservices architecture. It describes the challenges you might encounter and the patterns you can use to meet those challenges. It shows an example of the necessary tasks to move from monolith to microservices on a Java application.

This chapter includes the following sections:

- Distributed data management
- New challenges
- Integration with the monolith application
- Which database technology to use
- Practical example: Creating the new microservices

© Copyright IBM Corp. 2016. All rights reserved.

www.dbooks.org

4.1 Distributed data management

A common monolith application usually has one or two relational databases that include all the information needed for the system to work. This monolith application is usually managed by a specific team. The application's architecture makes adding new functionalities and scaling and changing the application a slow and high-risk process.

In a microservices architecture, each microservice must have its own database that allows new functionalities (that is, new microservices) to be added quickly and with low risk of affecting other functionalities of the application. This approach also allows the use of the appropriate tool for each functionality. For example, you might use the Data Cache for Bluemix service or a Redis database to store key value information about data that is common for the application. This data might be a list of cities, stores, or airports. The appropriate tool might also be a relational database to store transaction (order) information that must have consistency among the various entities that are related to the process.

This new approach offers advantages (see Chapter 1, "Overview of microservices" on page 15), but moves from a point of centralized data management to distributed data management. This approach brings new challenges that must be considered.

4.2 New challenges

When moving from centralized data management to having multiple tools and databases, challenges that were not present in a monolith application exist:

- Data consistency
- Communication between microservices

4.2.1 Data consistency

By having a single database for a complete application, you can get, modify, and create various records in only one transaction, giving you the security of having the same view of the information all the time. In a monolith application, the inconsistency problem occurs when the application grows large and you need to gain more speed in the queries. As a result, you create cache services or send common information one time to the client device (for example, mobile device, HTML, desktop application, and so forth). This reduces the server calls, which then produces the problem of needing to ensure that all stages (that is, database, cache, and client) have the same information.

In a microservices architecture with distributed data management (multiple databases), the inconsistency problem occurs from the beginning of development. Whether the application is big or small does not matter. For example, to create a new order of a ticket in a rewards system where a person orders air tickets with reward points, you must first get the current points total of the person. You must also get the cost in points of the air ticket, and then, after comparing both values, create the order.

This process in a microservices architecture has the following stages:

1. Get the price of the ticket in points from the Catalog microservice.
2. Get the total amount of points of the person ordering the ticket from the Rewards microservice.
3. Compare the points cost and the amount of points in the business logic level of the Orders microservice.

4. Create the ticket order from the Orders microservice.
5. Create the reservation of the flight from an external service for reservations.
6. Update the number of points of the person creating the order from the Reward microservice.

This process has three separate microservices, each with a different database. The challenge is to share the same information among all the microservices. This is to ensure none of the information changed as the ticket completes and to ensure that the complete information is updated in each microservice at the end of the transaction.

The consistency problem is common in distributed systems. The CAP theorem states that in a distributed system, consistency, availability, and partition tolerance (CAP) cannot occur at the same time; only two of these characteristics can be guaranteed simultaneously.

For a simple explanation of the CAP theorem, see the following website:

`http://ksat.me/a-plain-english-introduction-to-cap-theorem`

In today's environment, *availability* is the most important choice. Therefore, two patterns can be considered as solutions to the consistency problem:

- Pipeline process
- Event-driven architecture pattern

Pipeline process

This process might seem like the easy solution for the consistency problem, and it works for scenarios where you need to follow a set of steps to complete one process (transaction). Continuing with the example of a rewards system, with the pipeline pattern, a single request to the Orders microservice starts a flow between microservices, as shown in Figure 4-1 on page 54.

www.dbooks.org

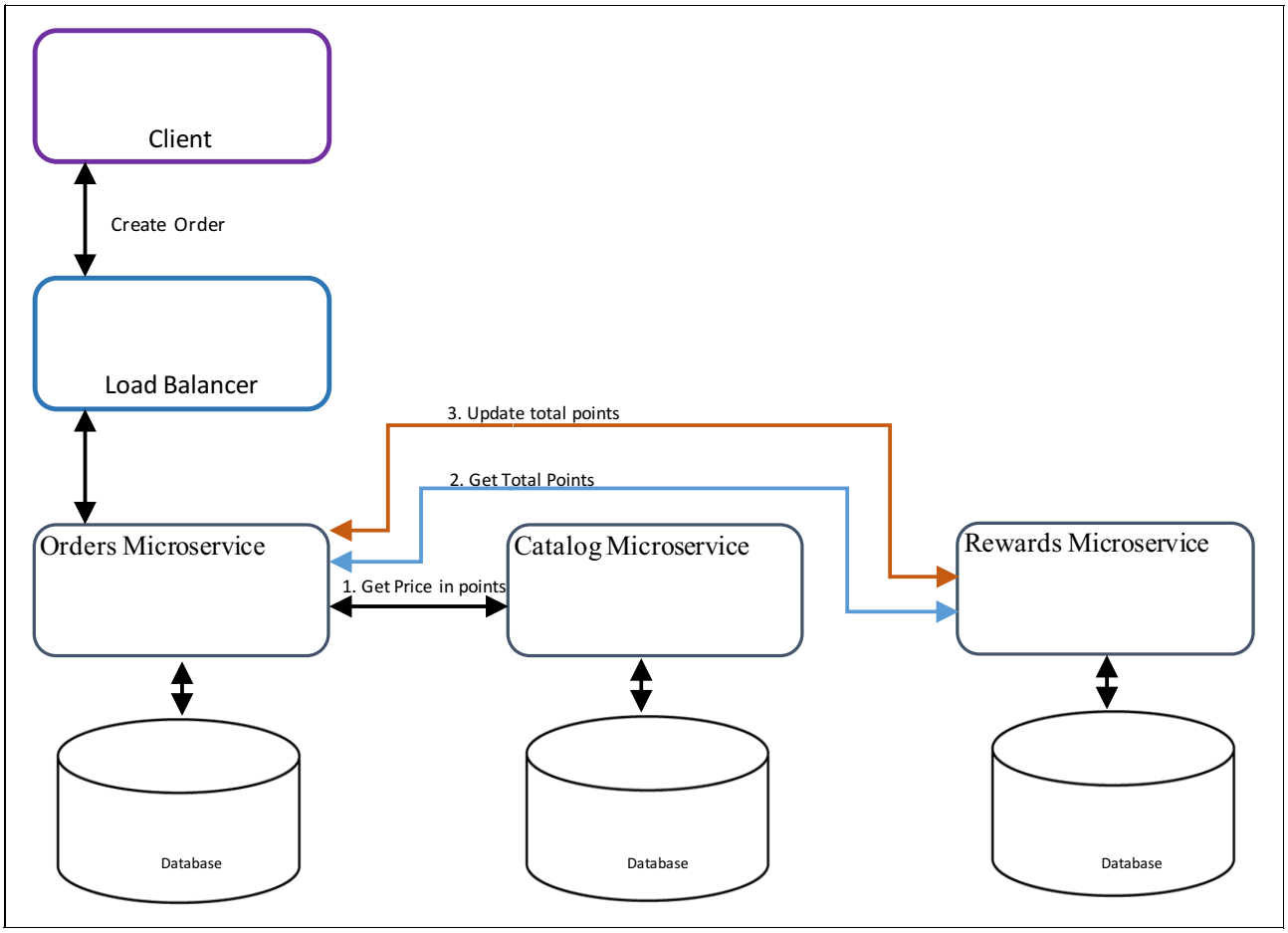


Figure 4-1 Pipeline pattern

Although the pipeline process shown in Figure 4-1 is common, this might not be the best solution because it creates a dependency between services. Therefore, if you make a change in the Catalog or Reward microservices, you must change the Orders microservice.

Event-driven architecture pattern

The other solution for the consistency problem is the event-driven architecture, where the microservices communicate with each other by publishing an event when something important happens in their databases. For example if a record is created or updated in a microservice and other microservices subscribe to that queue of events, when the subscribing microservices receive the message, they update or create their own records. This record update or creation might lead to a new event being published.

By using events, you can create transactions that are related to multiple microservices.

Referring to the previous example, the steps and events necessary to complete the transaction are as follows:

1. The client creates an order calling the Orders microservice. This microservice then creates an order record in its database with an *in-process* status and publishes an *order created* event (see Figure 4-2).

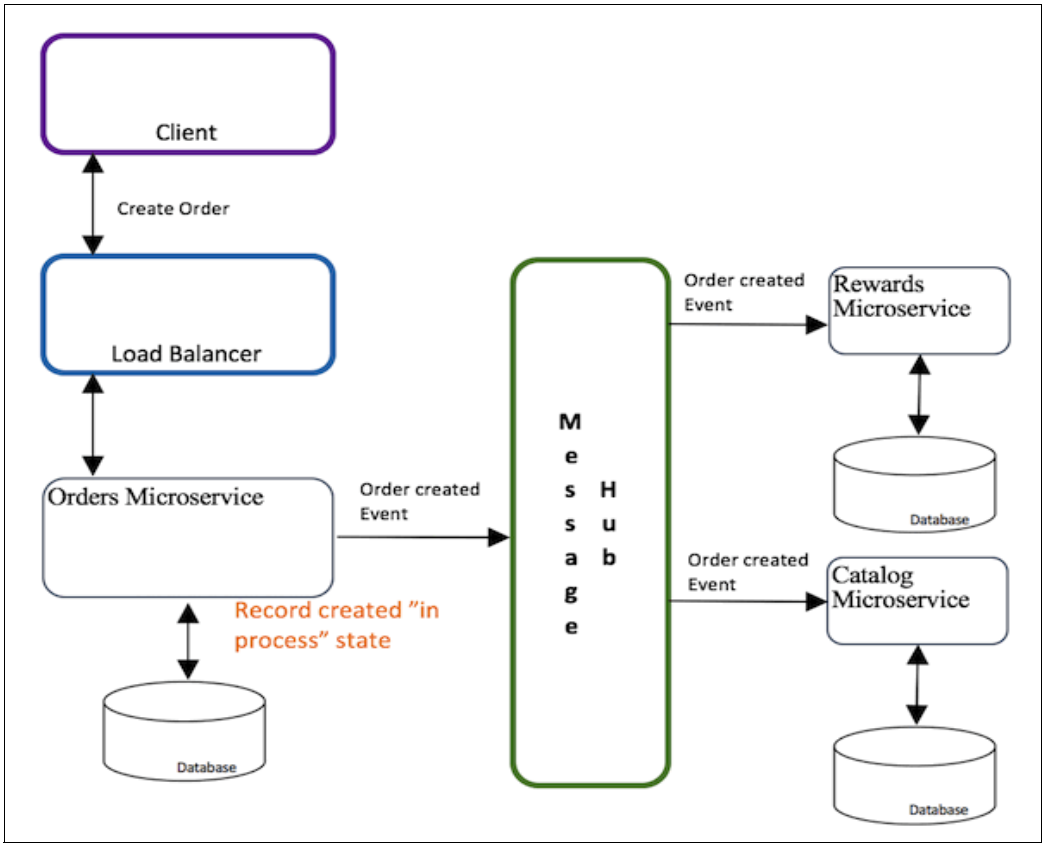


Figure 4-2 In-process status record created

www.dbooks.org

2. The Rewards and Catalog microservices receive the *order created* event message and update their respective records to reserve the points and to hold the ticket price. Each one then publishes a new event (see Figure 4-3).

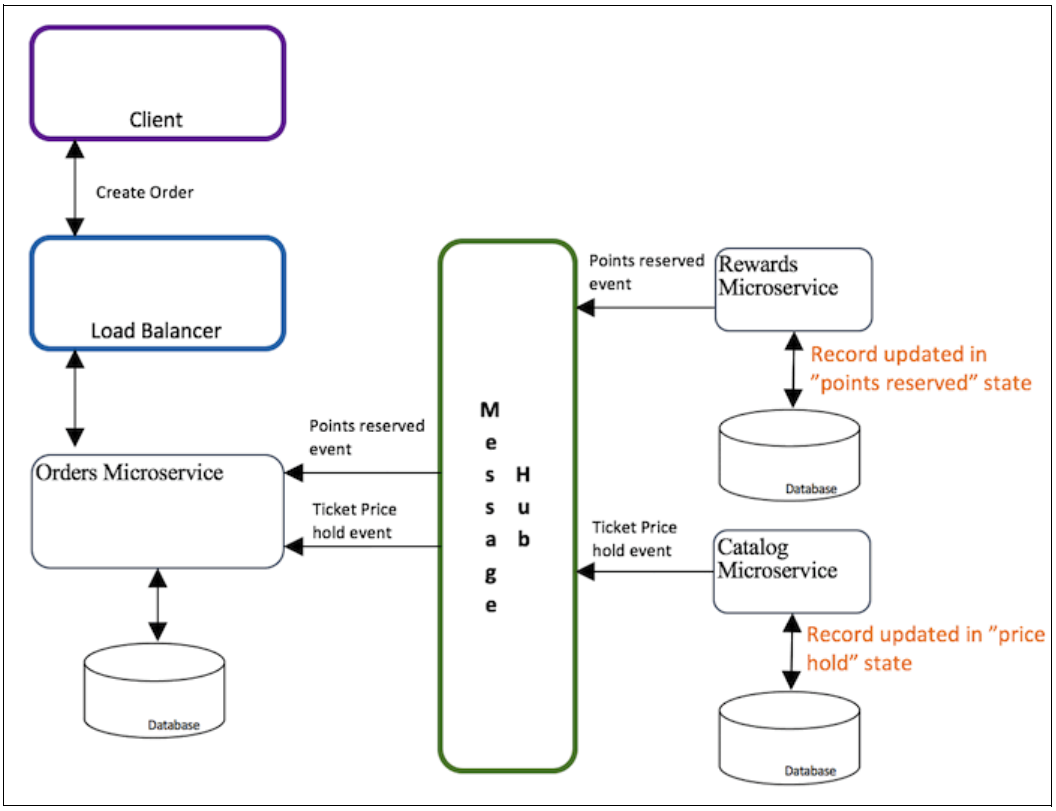


Figure 4-3 Confirmation form microservices

3. The Orders microservice receives both messages, validates the values, and updates the created order record state to *finished*. It then publishes the *order finished* event (see Figure 4-4 on page 57).

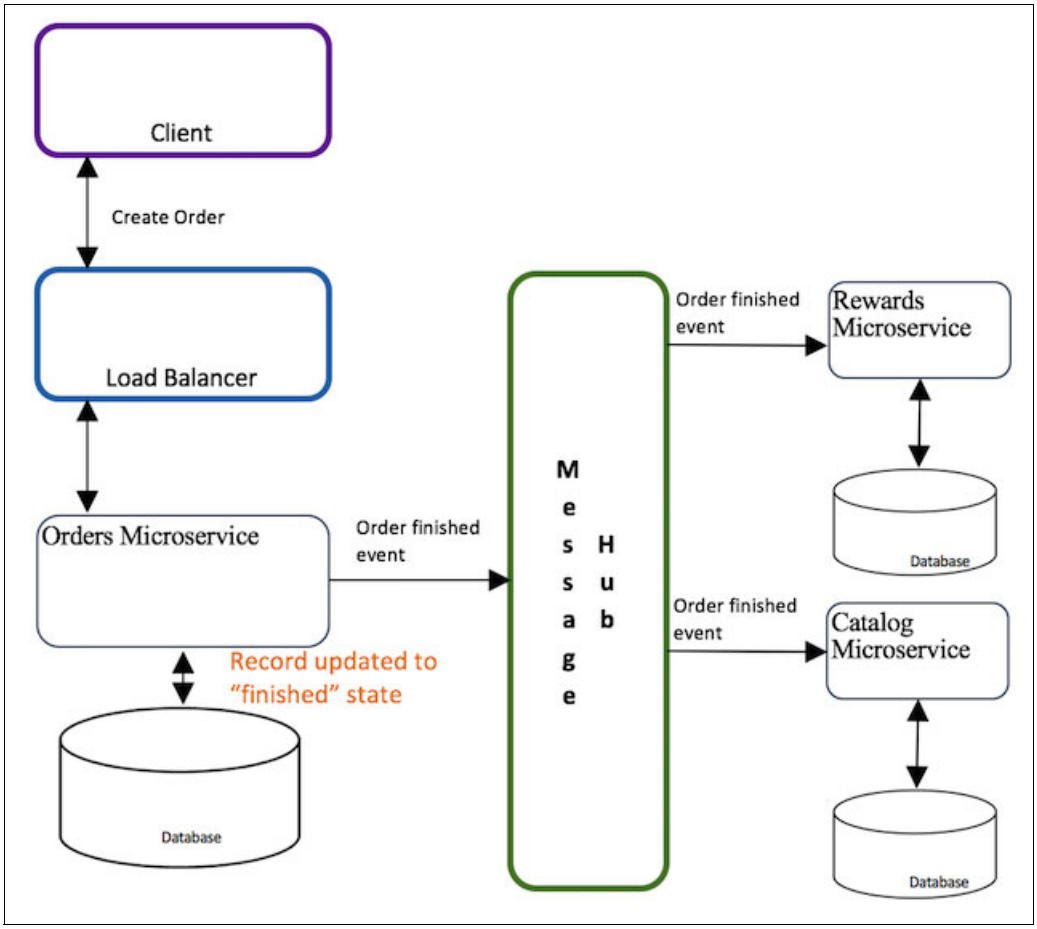


Figure 4-4 Record state updated to finish

www.dbooks.org

4. The Rewards and Catalog microservices receive the final event and update the respective records to an *active* state (see Figure 4-5).

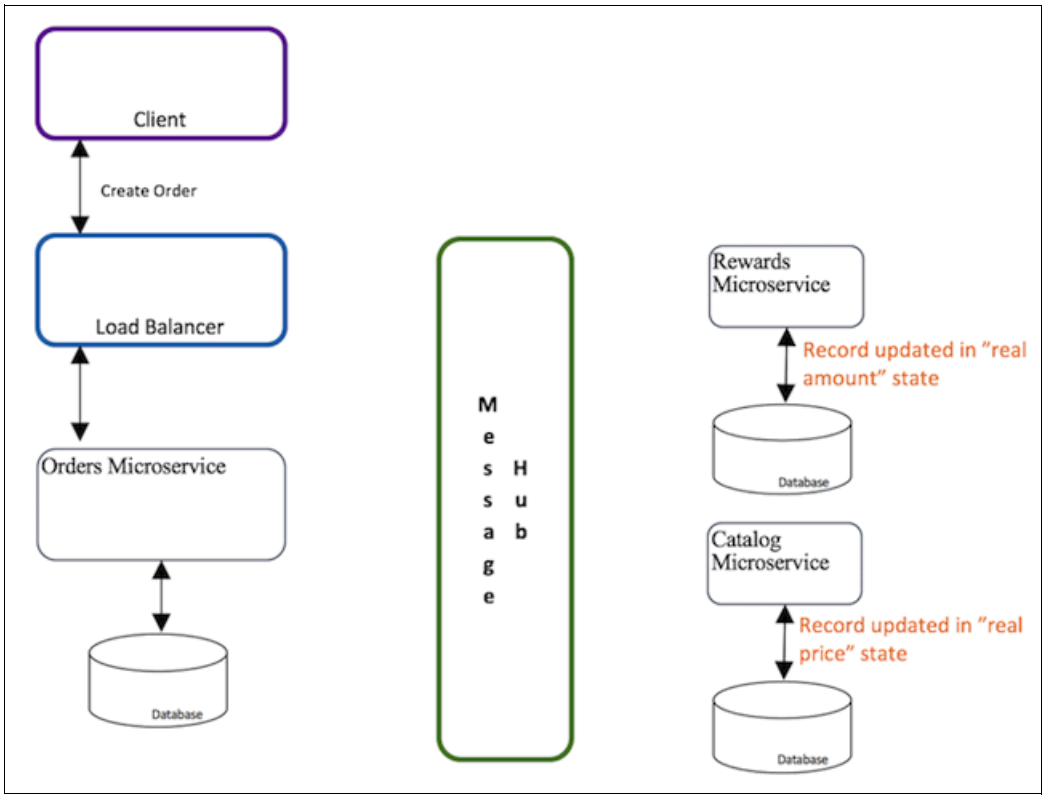


Figure 4-5 Final update of related records

With this pattern, each microservice updates its database and publishes an event without a dependency on other microservices; it is ensured that the message broker delivers the message at least one time to the related microservices. Some benefits are achieved thus far, however, consider that this is a complex programming model so now the *rollback* transactions must be programmed. For example, suppose the user does not have enough points to order the ticket. A transaction that cancels the created order must be created to resolve that situation. Section 4.2.2, “Communication between microservices” describes another pattern that might help with this consistency problem.

4.2.2 Communication between microservices

As 4.1, “Distributed data management” describes, new challenges must be met in addition to dealing with data consistency. The next challenge is communication between microservices. In a monolith application, having only one database allows information to be queried from different tables with no problems. At this point, however, defining how to consolidate information from separate microservices is necessary before delivering the information to the client.

Continuing with the airline rewards system example in 4.2.1, “Data consistency” on page 52, you create a list of tickets in the catalog containing all the reviews that users made about places. You must get that information from two databases:

- The list of places (from the Catalog microservice database).
- All the comments about those places listed in the catalog microservice (from the Reviews microservice database).

After you get the information from the databases, the consolidated information must be sent to the client. In a microservices architecture, one microservice must not query the database from another microservice because this creates a dependency between them. With the approach in the example, each database must be accessed only by the API of its microservice. For cases like this, use the Gateway pattern.

API Gateway pattern

The API Gateway pattern consists of the creation of a middle tier to provide additional interfaces that integrate the microservices. This middle tier is based on an API Gateway that sits between the clients and microservices; it provides APIs that are tailored for the clients. The API Gateway hides technological complexity (for example, connectivity to a mainframe) versus interface complexity.

The API Gateway provides a simplified interface to clients, making services easier to use, understand, and test. It does so by providing different levels of granularity to desktop and browser clients. The API Gateway can provide coarse-grained APIs to mobile clients and fine-grained APIs to desktop clients that can use a high-performance network.

In this scenario, the API Gateway reduces chattiness by enabling 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 one time, and it leverages the low-latency connectivity and more powerful hardware on the server side.

Continuing with the example, to get the list of places and reviews, create an API Gateway exposing a service to get the consolidated list of places with their reviews. This gateway makes a call to the Catalog and to the Rewards microservice, consolidates the information, and sends it to the client, as shown in Figure 4-6.

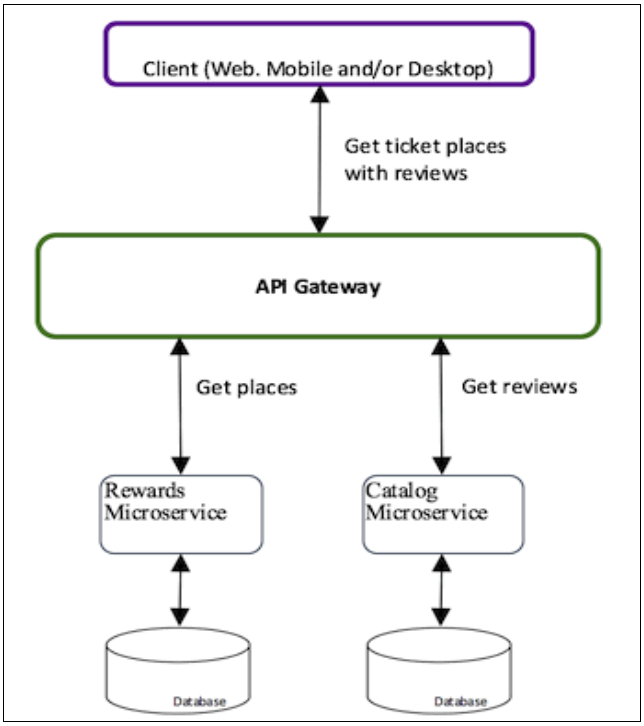


Figure 4-6 API Gateway pattern to aggregate information from separate microservices

This pattern can help solve inconsistency and data aggregation problems. However, it presents a new risk that is related to the system performance (latency) because now there is

www.dbooks.org

an additional network point, which is the entry point for clients. Therefore, it must be able to scale and have good performance.

Shared resources pattern

To gain more autonomy in a microservices architecture, each service must have its own database, and databases must not be shared or accessed from a different microservice. However, in a migration scenario, a difficult task might be to separate some entities that are already working. Instead of having problems with decoupling all the entities, an easier way to solve this special case exists.

For cases in which entities are highly coupled, a different solution that might be seen as more than an anti-pattern can help to solve this special condition. The solution is to have a shared database. In this form, instead of having two microservices and two databases, one for each microservice, you have only one database that is shared between the two microservices. The important key for the shared resources pattern to work is to keep the business domain close (that is, use it only when the entities are related to the same business domain). Do not use it as a general rule in the architecture but as an exception. The example in Figure 4-7 shows how this pattern works.

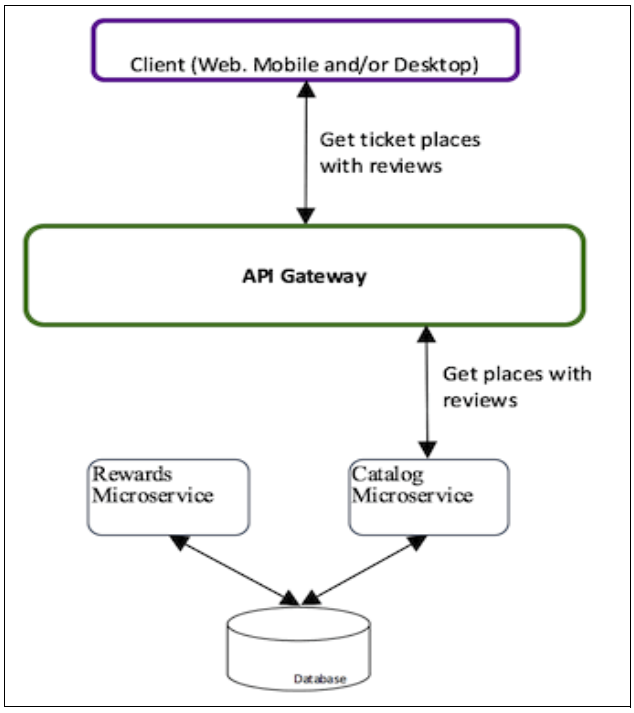


Figure 4-7 Shared resource pattern to share information between services

Because this pattern is used with a business domain approach, it might also eventually have only one microservice to manage two entities. So, for this example, if the application has reviews for the places that are in the Catalog database, you can consider these reviews as an attribute of the ticket and leave them in the same database and the same microservice. Ultimately, you have the Catalog microservice with all the information that is related to the tickets, which includes their reviews.

4.3 Integration with the monolith application

In the evolution of a monolith application to use microservices, not rewriting or creating a new one, certain situations can arise where the complete application cannot evolve or the evolution of the application occurs in stages. Therefore, you must be sure that the evolved and new components of the application are able to communicate and work together with the unchanged parts of the monolith application (for example, transactional systems that are too complex to change).

In cases where you plan to keep one or more parts of the monolith application, an important factor is to consider this part as one more microservice. You can then integrate it the same way you integrate other microservices.

4.3.1 Modifying the monolith application

Various approaches to refactoring an application are described in 3.4, “Refactoring” on page 42. The result of refactoring a monolith application yields separate microservices, each with a separate database. In most cases, you keep alive some parts of the monolith application. The goal for the parts of the monolith application is to make them communicate in the same way that the other microservices are communicating. To do this involves the patterns and the technology stack in which the monolith application was developed.

If you use the event-driven pattern, make sure that the monolith application can publish and consume events, including a detailed modification of the source code to make these actions possible. This process can also be done by creating an event proxy that publishes and consumes events. The event proxy translates them to the monolith application so that the changes in the source code are minimal. Ultimately, the database remains the same.

If you use the API Gateway pattern, be sure that your gateway is able to communicate with the monolith application. To achieve this, one option is to modify the source code of the application to expose RESTful services that are easy to consume by the gateway. This can also be achieved by the creation of a separate microservice to expose the monolith application procedures as REST services. The creation of a separate microservice avoids big changes in the source code; however, it includes the maintenance and deployment of a new component.

4.4 Which database technology to use

The capability to choose the appropriate tool for each functionality is a big advantage, but it leaves the question of which database is best to use. The objective of this section is to help you understand the various technologies and use cases for each database. Then, when you decide to evolve your application from a monolith to a microservices architecture, you can also decide whether changing the technology you are using is a good approach. In simple terms, this decision can be summarized as SQL or NoSQL.

4.4.1 Relational databases

Relational databases (known as SQL databases) are well known; you might already use a relational database as the database of the monolith application you want to evolve. SQL databases are used in almost all applications that need to persist data, no matter what the use case or scenario is.

www.dbooks.org

Some of the characteristics provided by a relational database follow:

- Normalized data to remove duplication and reduce the database size
- Consistency and integrity in transactions with related data (for example, atomicity, consistency, isolation, and durability, or ACID)

A relational database is not the best option in these cases:

- Fast access to big amounts of data
- Hierarchical data
- Key-value data (that is, caches and static data)
- Rapidly changing schemas or multiple schemas for the same data

4.4.2 NoSQL databases

The term *NoSQL* comes from the fact that almost all systems were developed in an SQL database, even if the data was not relational; the term NoSQL clarifies that the database is not an SQL database. The term refers to databases that cover all types of data, for example key-value, documents, and time-series data.

Today there are various NoSQL database options, each one to achieve a specific outcome. In microservices, each service must have its own database. This allows you to evaluate the database both from its integration capabilities and from the specific case or problem that must be solved.

Some examples of NoSQL databases are as follows:

- Document databases to save documents and their data. The following example cases are are for saving sales information including the store, seller, client, and products in the same document:
  - Apache CouchDb: JSON documents
  - MongoDB: JSON documents
- Graph databases to store data with a natural relationship. An example includes social network data or customer data, including past purchases, to make real-time suggestions based on interests:
  - Apache Tinkerpop
- Key-value databases used for high availability, reference data, or caching memory. An example is a list of stores in each city common to different parts of the application:
  - Redis, in memory database
  - IBM Data Cache service on Bluemix, in memory database
- Columnar databases to store large amounts of isolated data for analytics, for example, all the user interactions in a website to make real-time analytics of the user's behavior:
  - Druid
  - Cassandra

In environments today, NoSQL databases are generally accepted. With so many solutions, the challenge is to find the correct one to solve your problem. In this case, the simplest solution might be the best solution, starting with the analysis of the technologies you already know and whether they are the correct solution for your problem. Learning a new technology, however, might be easier, rather than trying to adapt the one you already know. Also, with cloud computing, installing and managing every component is not necessary, so you can immediately start using the tools you need.

4.5 Practical example: Creating the new microservices

As described in 3.5, “Identifying and creating a new architecture example” on page 46, the evolution of the Java monolith application is divided into four microservices. These microservices are composed of a new user interface (UI), a new microservice to search products, an evolved customer microservice, and the Orders microservice.

The first microservice to create in this example is the new Search microservice, divided into the following stages:

1. Select the technology stack:
   - Cloud computing environment
   - Run time
   - New database technology
   - Secure integration
   - Data transformation
2. Migrate the data to a new database:
   - On-premises database secure connection
   - Extract, transform, and load (ETL) the data
3. Create the API to search the catalog.

4.5.1 New microservice catalog search

As described in Chapter 1, “Overview of microservices” on page 15, a better search system is needed for the application so users can find the products they need in an easier way. This new search must be quick and composed of dynamic rather than specific filters.

In the monolith application, you can filter the products list only by the category of the product, and these categories are created in the SQL database called ORDERDB. Here, one product belongs to one or more categories and one category can have one or more products. Figure 4-8 shows how the products and categories relate (that is, many-to-many).

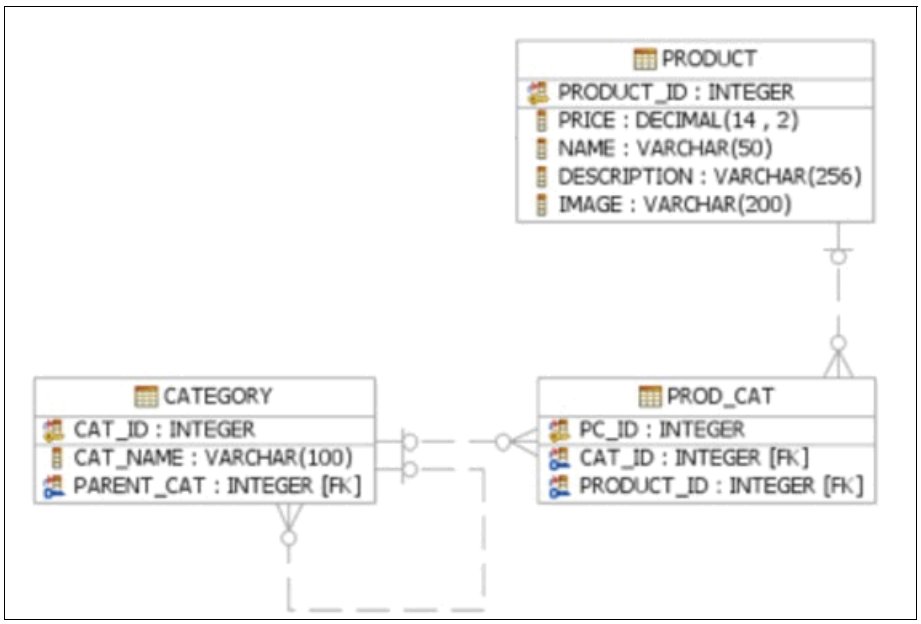


Figure 4-8 Product categories relational model

www.dbooks.org

Select the technology stack

Select the appropriate technologies for this kind of search. Also, select the appropriate environment to deploy, create, run, and manage this microservice.

Cloud computing environment

As described in 3.5.2, “Architecture: To be”, this example uses a platform as a service (PaaS) to gain more speed and flexibility. Start evaluating the options and decide which one is the best fit for this solution.

Although various PaaS providers are available, this example uses IBM Bluemix because it gives flexibility in the deployment options: Cloud Foundry, Docker containers, and virtual machines. These options are based in open technology. IBM Bluemix also has a rich catalog of services that has all the tools needed to create the complete solution.

Run time

In a microservices architecture, each microservice can be written in a different language. Although choosing a language might be difficult because so many are available, this example is not focused on why one is better than the other. Because this example creates a RESTful API only to search the products, Node.js is selected with the express framework for its callback-based asynchronous I/O that allows non-blocking interactions. It is selected also for its speed in creating APIs.

Database

As described in 1.3.1, “Fictional Company A business problem”, improving the search options is necessary so that the process is easier for customers. The search must be dynamic text and with a relevance or score of what the users want, not by filters that developers define.

Relational databases are *not* the best solution for this kind of job for the following reasons:

- You must create full text fields and then create indexes, both of which increase the database size.
- You must base the searches on the columns (fields) of the table, which means only adding new filters of the product attributes and not creating a dynamic search.
- Relational databases do not work quickly when millions of records exist.

Therefore, the NoSQL database option is the choice for this example.

In the IBM Bluemix catalog, several selections are available for managed NoSQL database services. Because the objective of this example is to improve the search options, the Elasticsearch-by-compose service is selected for the following reasons:

- It has full-text search.
- It is JSON document-oriented where all fields are indexed by default and all the indices can be used in a single query.
- It is without schema, which allows the addition of new attributes to products in the future.
- It works with a RESTful API; almost any action can be made using a simple HTTP call with JSON data.
- The Compose service deployment scales as the application needs, and it is deployed in a cluster by default.

Secure integration

Creating the new microservice involves migrating all the information from the product and category tables to the new database (Elasticsearch). The first step in the process is to create a secure integration between the on-premises database and Bluemix. This step involves

using the Secure Gateway service in the IBM Bluemix catalog, which provides secure connectivity from Bluemix to other applications and data sources running on-premises or in other clouds. Use of this service does not require modifying the data sources and does not expose them to security risks.

For more information, see the following website:

https://console.ng.bluemix.net/docs/services/SecureGateway/secure_gateway.html

Data transformation

As noted in “Secure integration” on page 64, all product information in the tables must be migrated to Elasticsearch in Bluemix. The data must first be transformed from the tables and relations to a JSON model. The advantage of using a JSON model is that you can show the information the way you want in the presentation layer by saving it with the schema you need. This is better than showing it based on how the data is saved and the relations it has with the other tables.

The main objective is to provide a better search service for the user: how does the user want to search or view? Based on that information, the document is created. In the monolith application, the search is based on the categories of the product because that is how the data is saved. However, now it can be saved based on the products and can use the category as one more attribute for the search. Figure 4-9 compares the relational model and the JSON model.

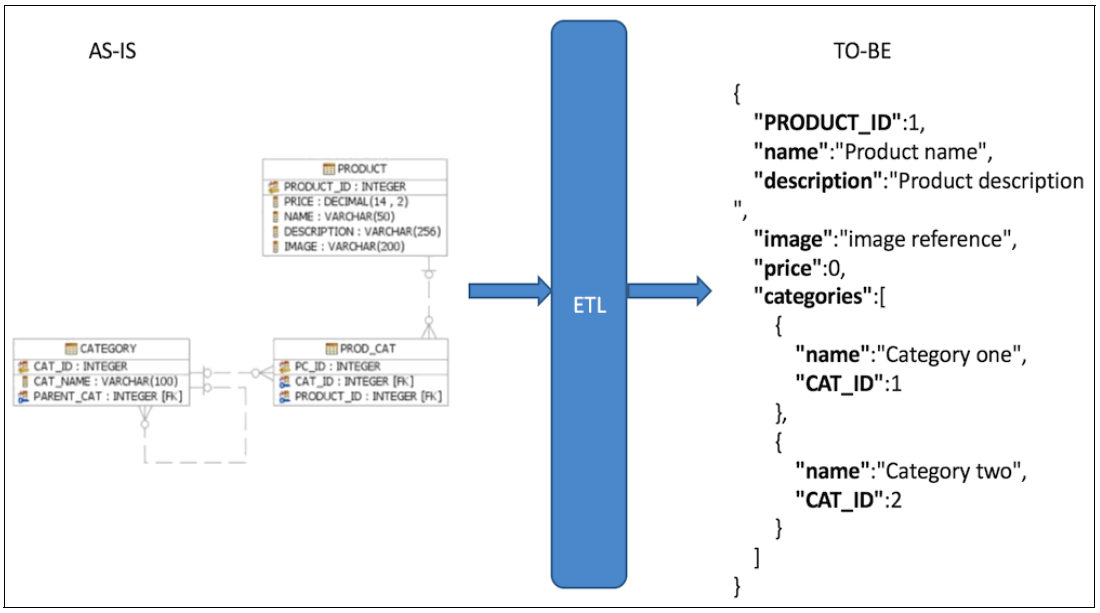


Figure 4-9 Product relational model compared to JSON

As Figure 4-9 shows, the same `PRODUCT_ID` and `CAT_ID` are kept to continue working with the monolith orders system.

After you have the new JSON model, you define how to extract, transform, and load the data into Elasticsearch. To help with these types of tasks, Bluemix offers services such as the Dataworks service. This service gets data from on-premises databases or in other cloud infrastructures and loads them into a Bluemix Data service, or the IBM DataStage® on Cloud service designed for complex data integration tasks. Although those are useful services, the current version of Dataworks does not support loading data into Elasticsearch, and the DataStage service is too advanced. Thus, for this example, an ETL tool is created as an API inside the Catalog search application in Bluemix.

www.dbooks.org

Figure 4-10 shows the architecture of the new Catalog search microservice.

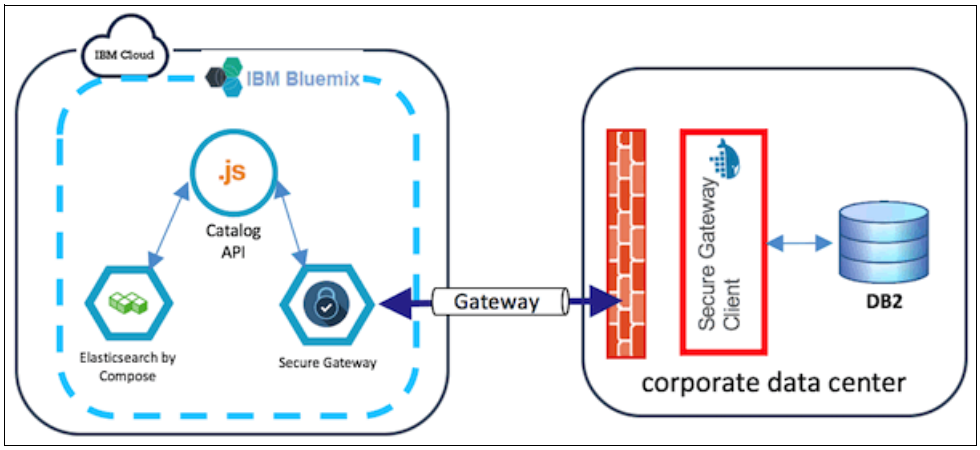


Figure 4-10 Catalog Search microservice architecture

Migrate the data to a new database

The steps for migrating all product information to the new Elasticsearch database involve securing a connection with the on-premises data center and then extracting, transforming, and loading the data.

On-premises database connection

Use the following steps to select the Secure Gateway service in Bluemix as the secure way to connect to the on-premises data center and then test the connection:

1. Open the IBM Bluemix website:

   `http://www.bluemix.net`

   Click either **LOG IN** (to log in with your IBMid) or click **SIGN UP** (to create an account).

2. Create the Secure Gateway service:
   a. Go to the Bluemix Catalog.
   b. Select the **Secure Gateway service** in the Integration category.
   c. Click **Create**.

3. Configure a gateway in the Secure Gateway dashboard. This option creates the connection point in Bluemix to connect to the on-premises or cloud services in a secure tunnel:
   a. Click the **Add Gateway** option from the Secure Gateway Dashboard.
   b. Insert the name of the gateway and select the security options and then click **ADD GATEWAY.** See Figure 4-11.

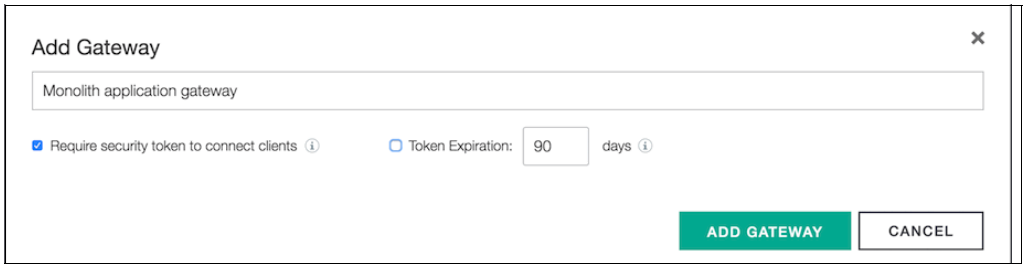


Figure 4-11 Add Gateway options

4. Add Destination for the Secure Gateway. Use this option to configure the on-premises resources that connect from Bluemix:
   a. Click the Gateway you just created.
   b. Click **Add Destination**.
   c. Select **On Premises** and click **Next**.
   d. Enter the host name and port (or IP address) of the resource you want to connect to, in this case the IP Address of the IBM DB2 server. Your window is now similar to what is shown in Figure 4-12. Click **Next**.

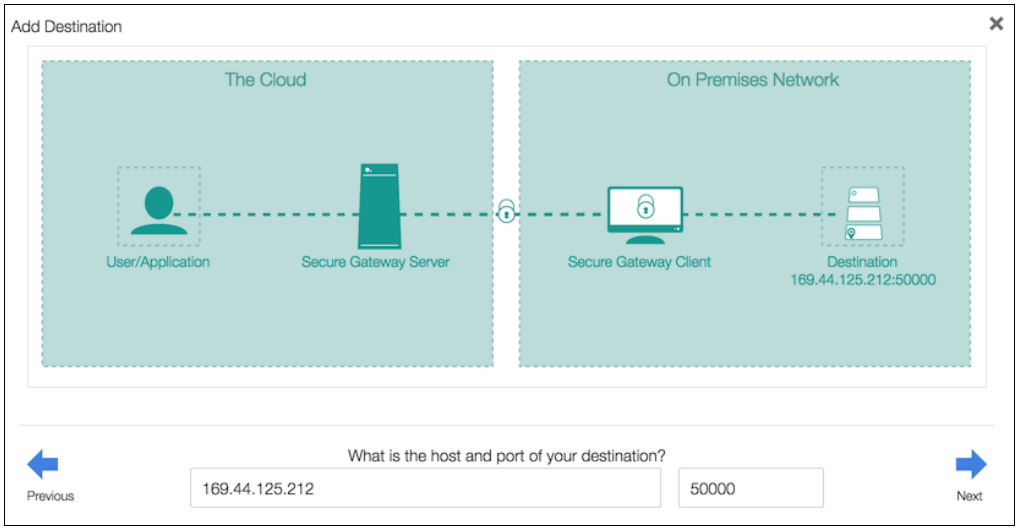


Figure 4-12 Secure Gateway Add Destination

   e. Select **TCP** as the protocol to connect to DB2 and click **Next**.
   f. Select **None** as the authentication method and click **Next**.
   g. Leave the IP Tables rules empty and click **Next**.
   h. Enter a name to use for the destination, in this case `Monolith DB2`, and click **Finish**.

   Your Monolith application gateway window is now similar to Figure 4-13.

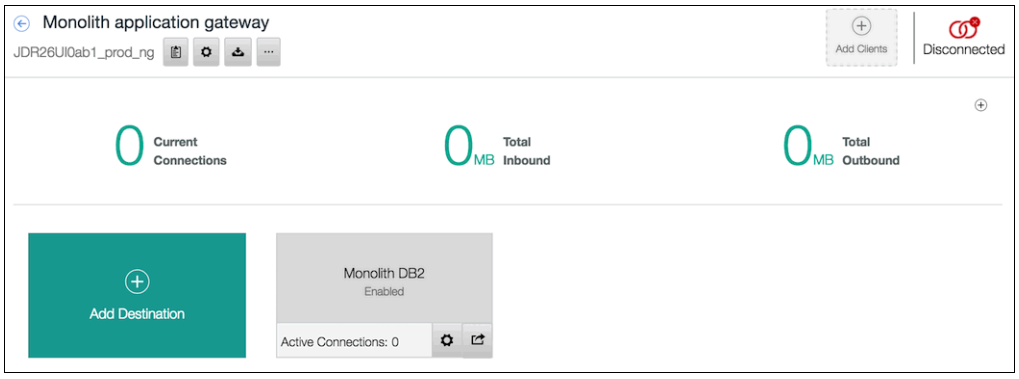


Figure 4-13 Monolith application gateway with DB2 destination created

www.dbooks.org

The Monolith DB2 destination shows details (see Figure 4-14) such as the cloud host and port to use to connect to the on-premises DB2.

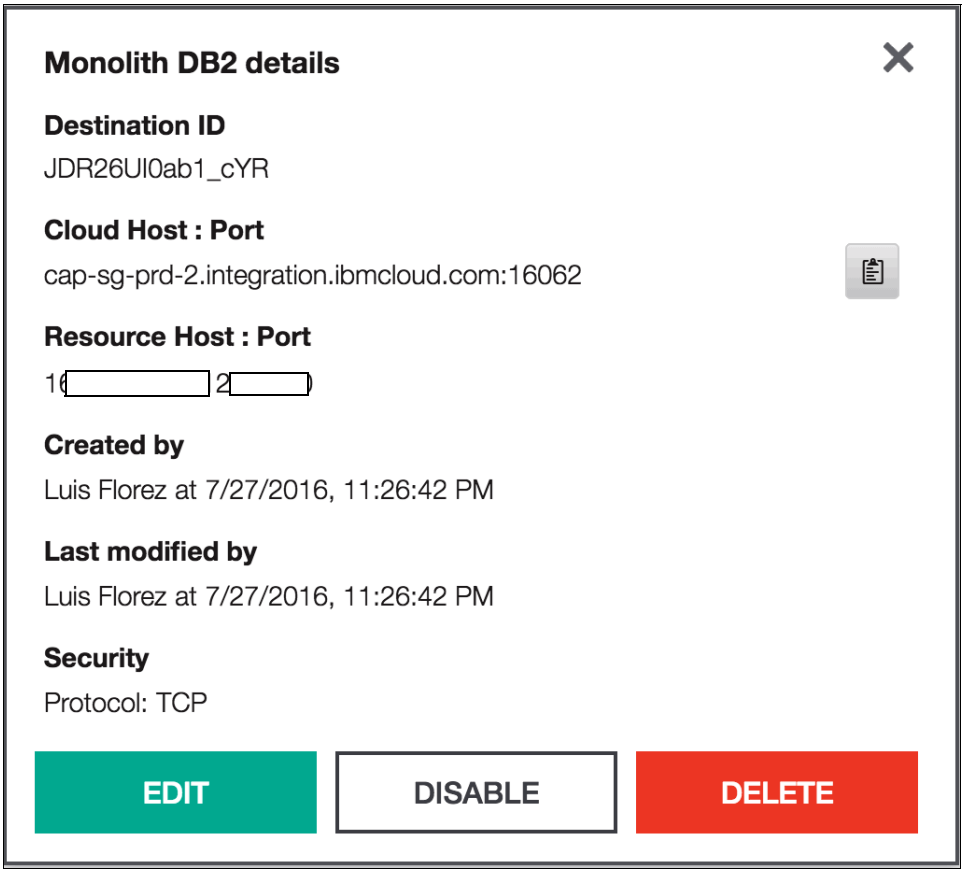


Figure 4-14 Monolith DB2 destination details

5. Now you can configure the gateway client. This client is the endpoint that runs on the on-premises data center to allow the secure connection. Use the following steps:
   a. On the Monolith application gateway dashboard, click **Add Clients**.
   b. Select a gateway client option to use (this example uses the Docker option):
      - IBM Installer: Native installer for different operating systems
      - Docker: A Docker image that is easy to deploy as a Docker Container
      - IBM DataPower®: A client to install in IBM DataPower
   c. Run the **docker** command to start the gateway client:

      ```
      docker run -it ibmcom/secure-gateway-client <gatewayId> --sectoken <secure token>
      ```

      Note: The `<secure token>` is the token provided in the gateway client selections window.

      Several seconds after running the gateway client, a confirmation is displayed (Figure 4-15 on page 69).

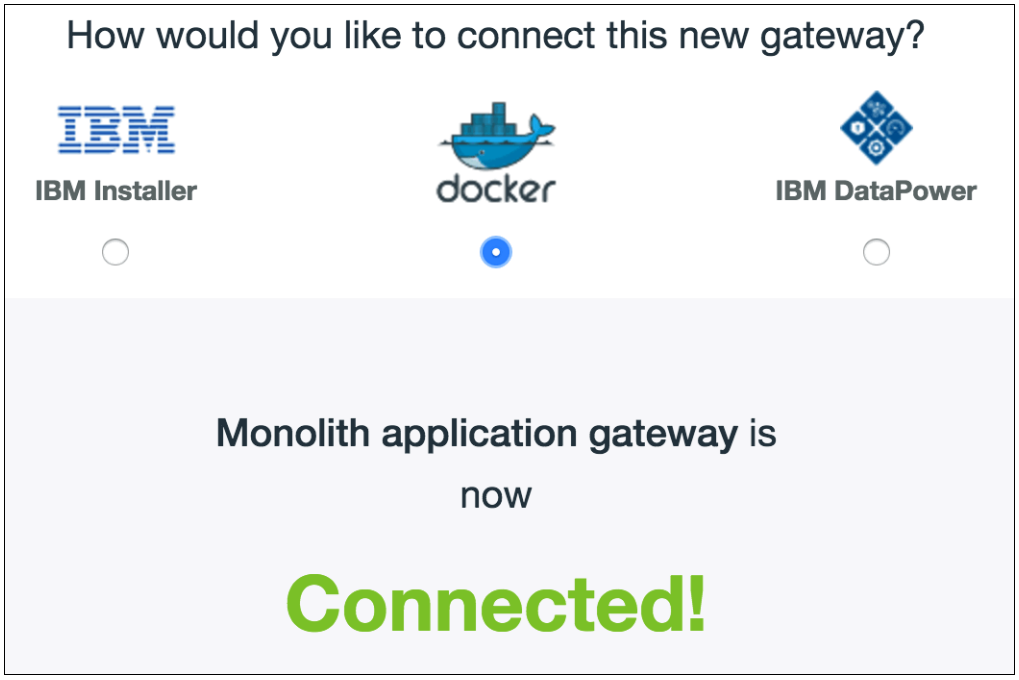


Figure 4-15 Gateway client connected confirmation

d. After installing and executing the Gateway client, configure the Access Control List (ACL) to allow the traffic to the on-premises database by entering the following command in the gateway client console:

```
acl allow <db2 ip address>:<port>
```

For this example, the command is as follows:

```
acl allow 1xx.xx.xx.x2:xxxxx
```

e. Test the connection to the DB2 database from your preferred DB2 client by using the cloud host and port given in the destination that is configured in step h on page 67.

Figure 4-16 on page 70 shows an example connection test.

www.dbooks.org

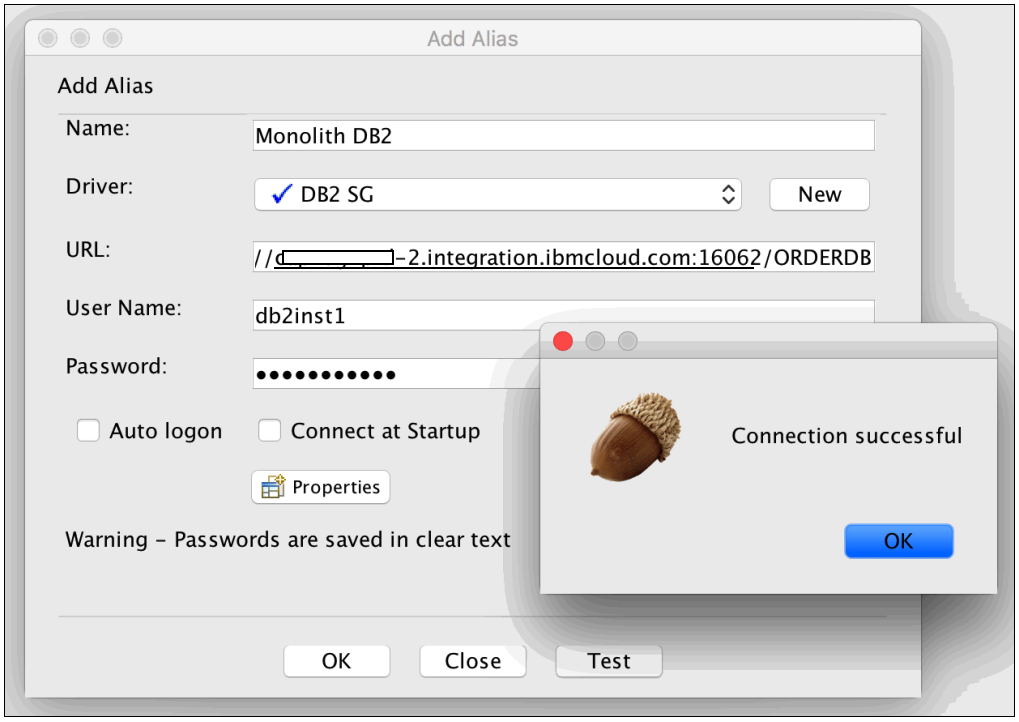


Figure 4-16 DB2 connection test using the information provided by the secure gateway

Extract, transform, and load the data

Now that the secure connection to the on-premises database is established, create the script that extracts all the products and their related categories, transforms each record into JSON, and posts it to the Elasticsearch service in IBM Bluemix:

1. Create the Elasticsearch service:
   a. Create a Compose account in:

      `https://www.compose.com/`

   b. Create the Elasticsearch deployment:
      i. Enter the deployment name.
      ii. Select the location you want to use.
      iii. Select the initial deployment resources.
      iv. Click **Create Deployment**.

      Now you can see all the information of your deployment, status, connection information and usage.

   c. Add a user to the deployment:
      i. Select **Users** → **Add User**.
      ii. Enter the user name and password.

   d. Configure the deployment information in Bluemix:
      i. Log in to IBM Bluemix:

         `http://www.bluemix.net`

      ii. Select the **Elasticsearch by Compose** service in the Data & Analytics category of the Bluemix Catalog.

iii. Enter the user name and password of the user you created on the Elasticsearch deployment.

iv. Enter the host name and password of the composed deployment. This information is in the Elasticsearch deployment Overview, under the Connection information section.

v. Click **Create**.

The connection information of the Elasticsearch deployment is displayed and is ready to use as a Bluemix service (Figure 4-17).

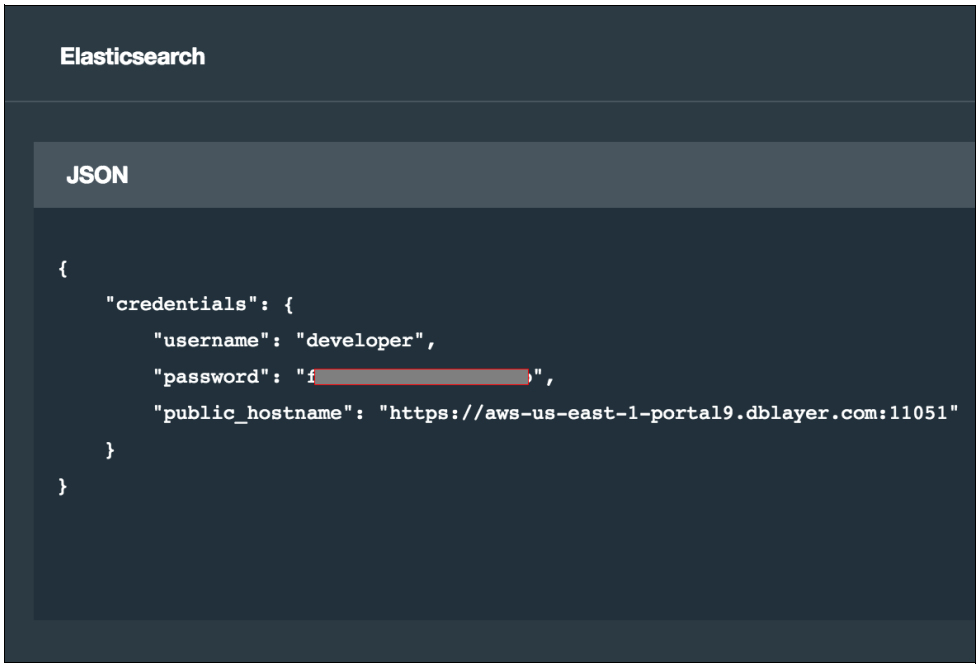


Figure 4-17 Elasticsearch connection information as a Bluemix service

2. Create the ETL application. To create the ETL application for this example, a small Node.js script was developed. This script connects to the DB2 database using the secure gateway cloud host information. It also uses a join to query the products and catalog tables, creates a JSON array with the format defined in the “Data transformation” on page 65, and inserts it into Elasticsearch using its *bulk* API.

   The repository for the code can be accessed on GitHub. See Appendix A, “Additional material” on page 119 for more information.

   The following is important information about the script:

   – This script is included as an API inside the Search application described in “Create the API to search the catalog” on page 72.

   – The connection for DB2 and Elasticsearch variables are defined in the `catalogsearch\config\config.js` file.

   – The script that extracts, transforms, and loads the data is located on the `catalogsearch\controllers\etl.js` file.

   – Line 15 of the `catalogsearch\controllers\etl.js` file is the query that is run to extract the data.

   – Lines 26 - 39 create the JSON array of the products.

   – Line 40 calls the Elasticsearch *bulk* API with the JSON array created.

www.dbooks.org

- From a web browser, you can run the script by entering the path defined in the `catalogsearch\routes\api\v1\etl.js` file.
- You can deploy this application on Bluemix, by modifying the `name` and `host` variables in the `catalogsearch\manifest.yml` file and running the **`cf push`** command from the Cloud Foundry command-line interface (CLI.)

3. Keep data in sync. For this example, all product information is now in the Elasticsearch deployment. Because this example is a one-time job, creating synchronization tasks between the databases is not part of this discussion. However, if you want to keep your data synchronized, you can find strategies at the following website:

   https://www.elastic.co/blog/found-keeping-elasticsearch-in-sync

Create the API to search the catalog

Now that the product information is in the Elasticsearch deployment, you can create a Node.js application to expose an API to search the products from the various clients (web pages and mobile applications).

Searching for a product in this monolith application involves selecting a category and then looking product by product, as shown in Figure 4-18.

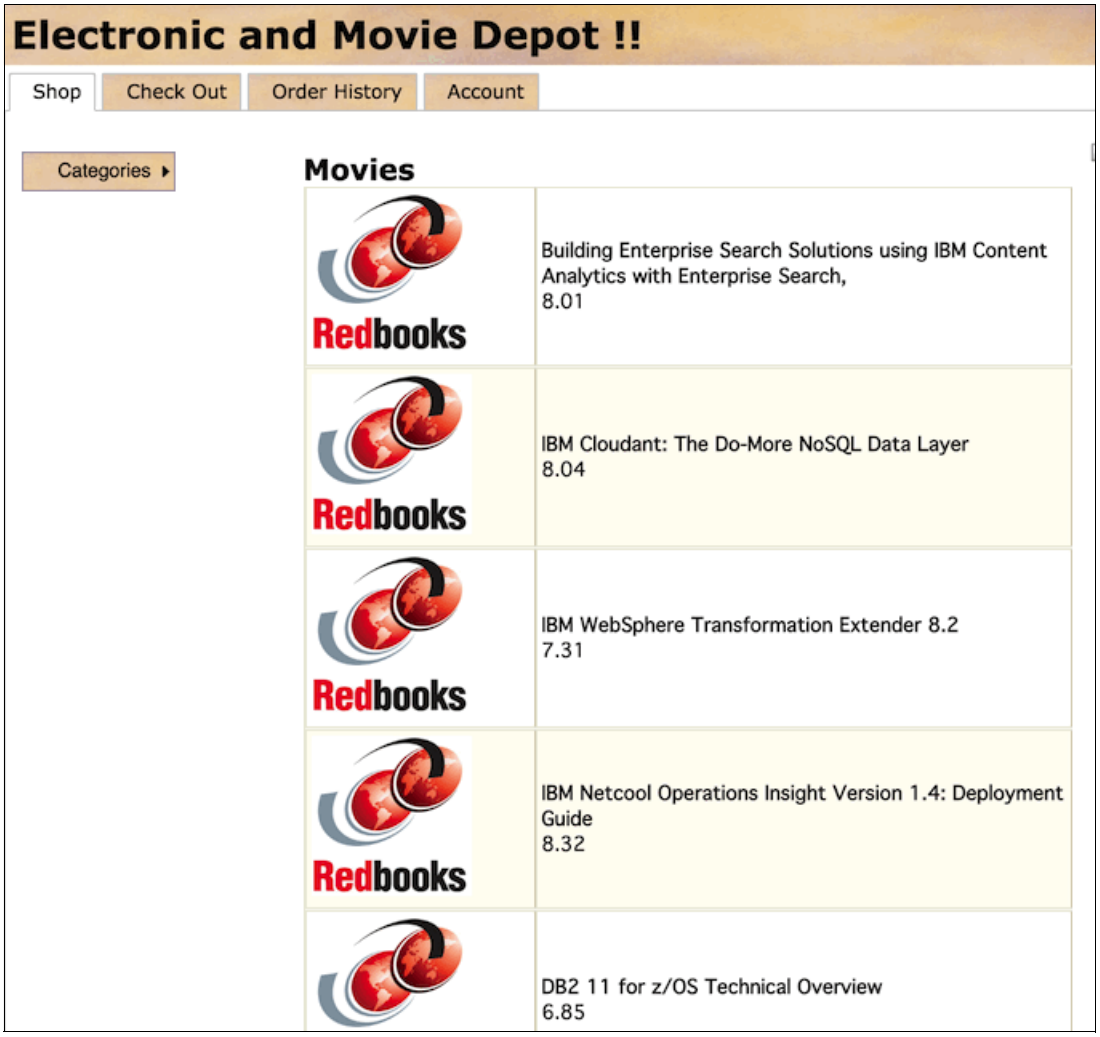


Figure 4-18 Monolith application product search

To make the user experience easier, provide a full text search by entering only one text field and calling a new search API. Figure 4-19 shows an example of the search of a category by sending only its name to the API.

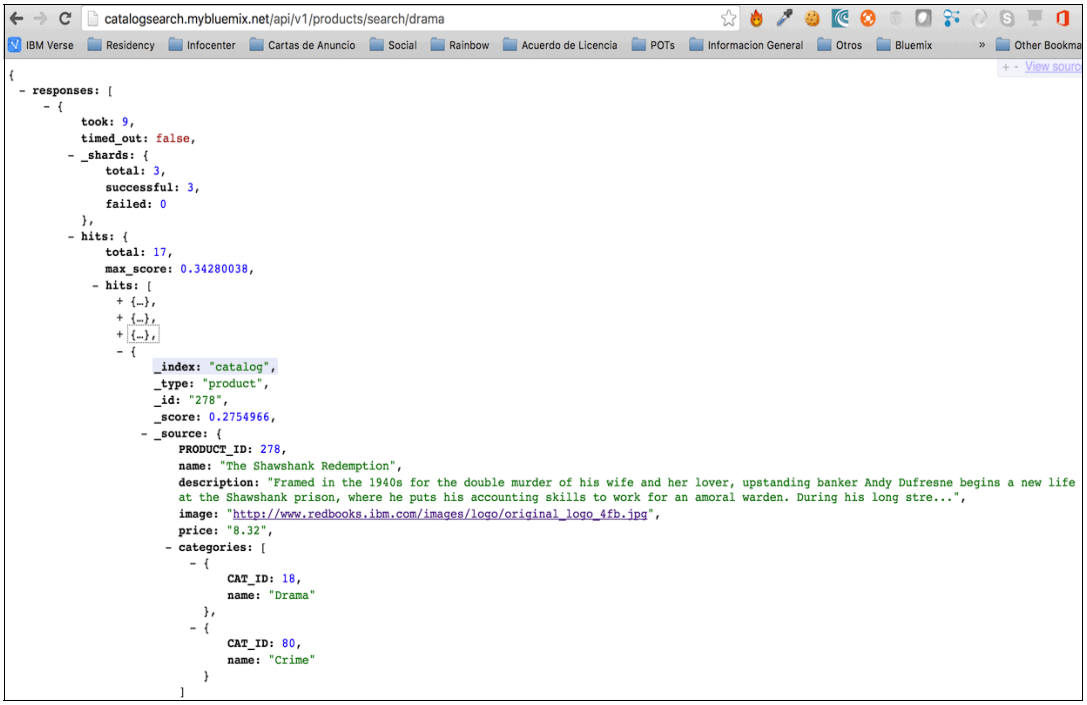


Figure 4-19 New Catalog Search API response

See Appendix A, “Additional material” on page 119 for the `catalogsearch` application source code.

The following parts are the important parts of the application:

- The connection variables are defined in the `catalogsearch\config\config.js` file.
- The search API route is defined in the `catalogsearch\routes\api\v1\products` file and is called by entering the URL with the following format:

  `http://<appurl>/api/v1/products/search/<search terms>`

  In the following example, *drama* is the text that the user enters when searching for the movie:

  `https://catalogsearch.mybluemix.net/api/v1/products/search/`*drama*
- The search to the Elasticsearch documents is made using the `elasticsearch` module. This is done by calling the Elasticsearch API to look for the entered text in all the documents in the index, which includes its name, categories, price, and description. This part of the code is at line 64 in the `catalogsearch\controllers\products.js` file.
- You can deploy this application on Bluemix by modifying the `name` and `host` variables in the `catalogsearch\manifest.yml` file and then running the **`cf push`** command from the Cloud Foundry CLI.

Summary

The new Catalog search microservice is now created using the Elasticsearch database with all the product information that was in the monolith application relational database using DB2. The example now continues the evolution of the monolith application with the evolution of the accounts data model.

www.dbooks.org

4.5.2 Evolution of the account data model

As described in Chapter 1, “Overview of microservices” on page 15, the application must also be evolved by adding social network data to the customer information. This provides a better view of clients and allows you to make detailed suggestions about the company’s products based on the customer’s social interactions.

This microservice is defined as a hybrid microservice (in 3.5.2, “Architecture: To be” on page 47). It allows you to continue working with the personal information of the client in the DB2 on-premises database, and it allows you to add the social networks data in a new database.

Selecting the technology stack

For this microservice, continue working with IBM Bluemix as the PaaS provider, and continue working with its Secure Gateway service for the integration with the on-premises database. The following technologies are selected for this example:

- Database

  If you look at the information provided by the User APIs of two large social networks, you see that this data changes all the time and that it differs for every user; it does not have a predefined schema that fits everyone. Therefore, for this example, a NoSQL database is selected because it allows for a schema-less data model with the ability to change easily and quickly. IBM Cloudant® is the selected database.

- Runtime

  Continue using Java as the runtime of this microservice because this microservice provides information from two data sources (DB2 on-premises and Cloudant). Also, the methods to get the personal information of the clients in the monolith application are already developed in Java. This takes advantage of all the code and knowledge you already have.

On-premises database Integration

To have a secure integration to get the personal information of the customers, use the same Secure Gateway infrastructure that was installed previously (see “On-premises database connection” on page 66).

Figure 4-20 on page 75 shows the new architecture of the evolved Accounts microservice.

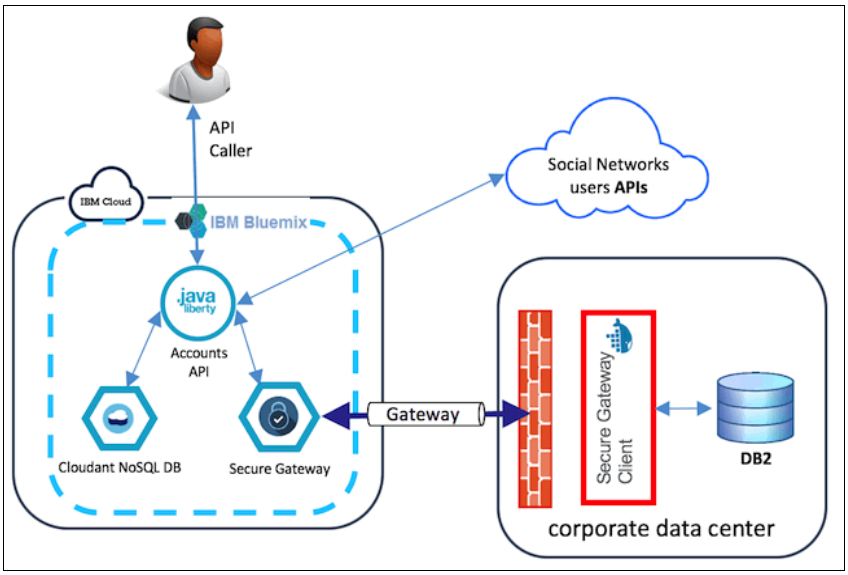


Figure 4-20 Accounts microservice architecture

Social information data model

Cloudant is already selected as the database for the social information. Now, create the service and the accounts database to use later with the Node.js application to expose the accounts API:

1. Open the IBM Bluemix website:

   `http://www.bluemix.net`

   Click either **LOG IN** (to log in with your IBMid) or click **SIGN UP** (to create an account).

2. Create the Cloudant service:
   a. Go to the Bluemix Catalog.
   b. Select the **Cloudant service in the Data and Analytics** category.
   c. Click **Create**.

3. Create the database:
   a. Select the service under the Bluemix Console.
   a. Go to the Cloudant Dashboard by clicking **Launch**.
   b. Click **Create Database** at the top right of the dashboard.
   c. Enter `accounts` in the Create Database field (Figure 4-21 on page 76).

www.dbooks.org

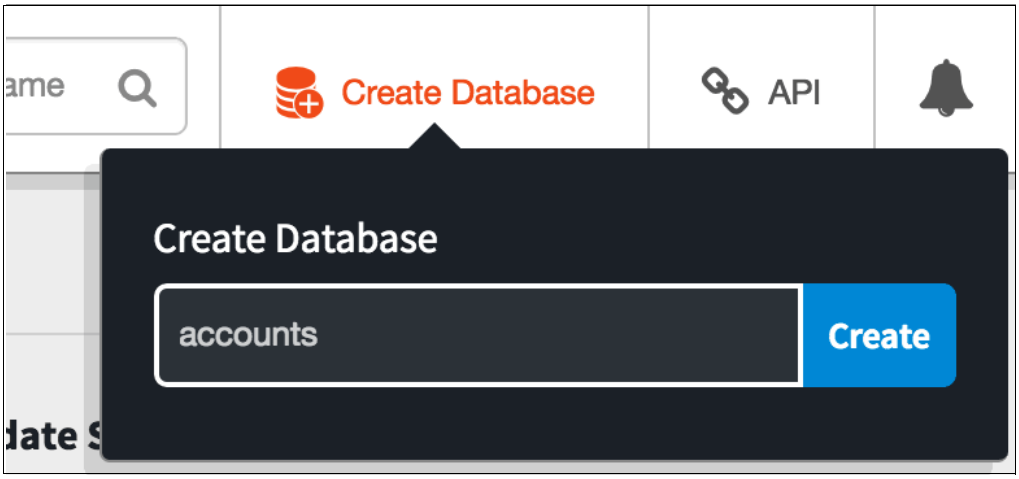


Figure 4-21 Create Cloudant database

4. Create the accounts view:
   a. Under the accounts options, click the plus sign (+) to the right of Design Documents and then select **New View**.
   b. Provide the following information for the fields (Figure 4-22 on page 77):

Design Document	New document
Index name	accounts
_design/	all
Map function	The function that emits the user name as the key and the complete document as the value
Reduce (optional)	None

   c. Click **Create Document and Build Index**.

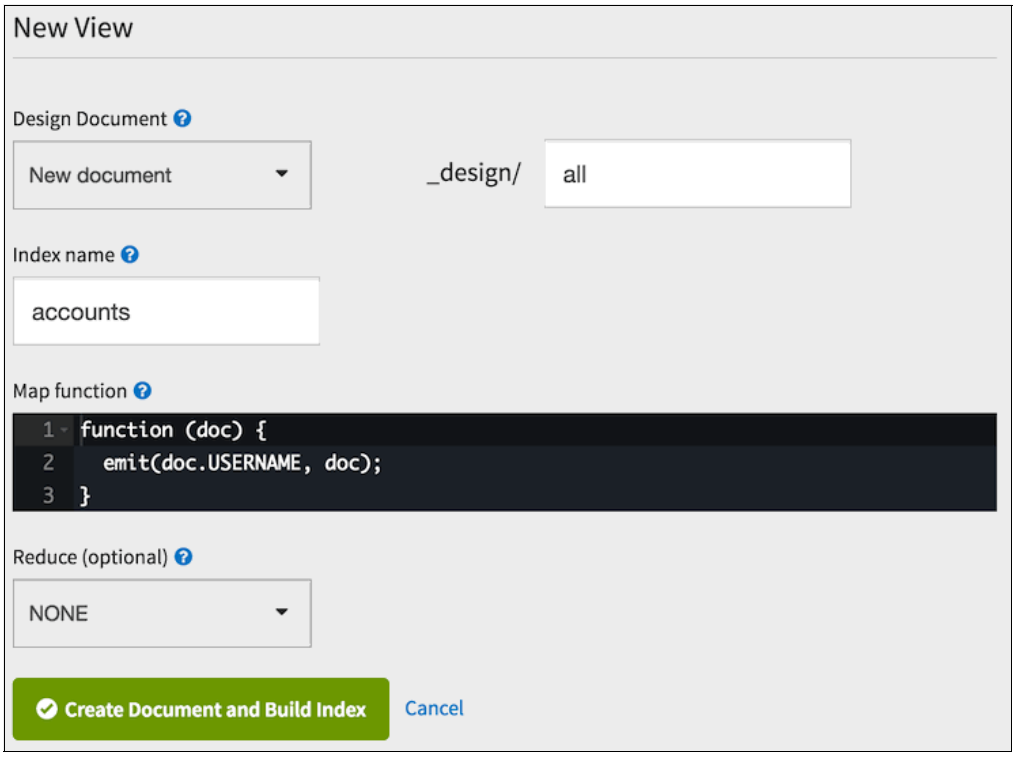


Figure 4-22 Cloudant Accounts design document view

d. An example of the social data JSON model extracted from two of the biggest social networks is in “Social networks JSON data model example” on page 119.

   You can add this information as a new JSON Document in the accounts database by clicking the plus sign (+) next to the All Documents option and then clicking **New Doc**.

e. In the New Document window, add a comma after the `_id` that is generated by Cloudant and then paste the information from “Social networks JSON data model example” on page 119. Figure 4-23 on page 78 shows the result.

www.dbooks.org

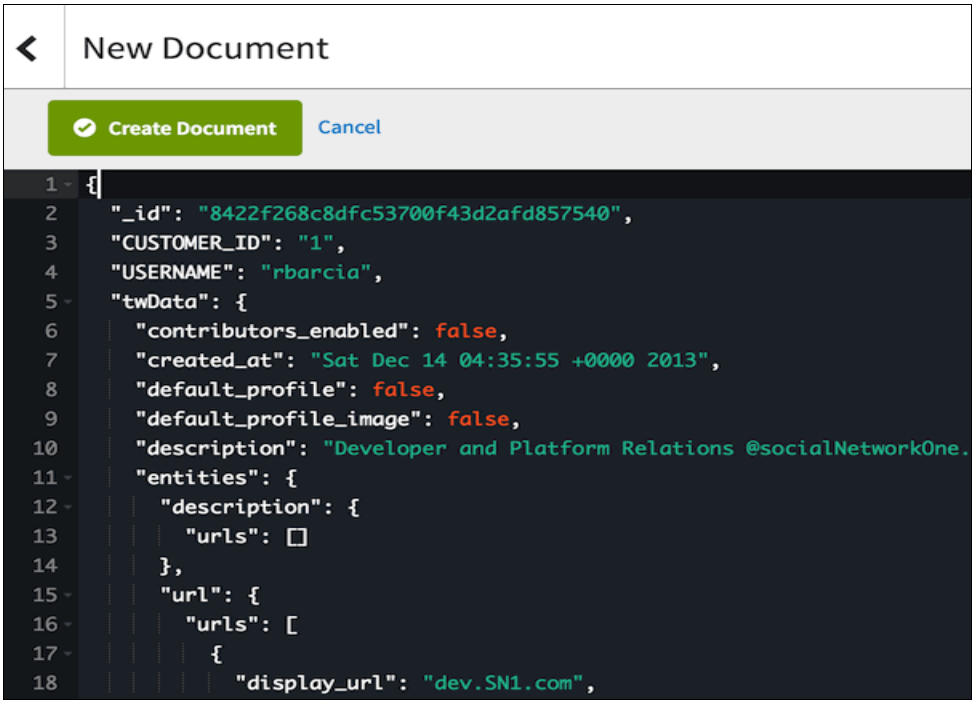


Figure 4-23 Cloudant New Document with the Social networks information of the customer

Note: This example copies the social network data manually from the APIs provided by two big social networks. For information about how to do this from your application, see the following websites:

`https://developers.facebook.com/docs/graph-api`
`https://dev.twitter.com/overview/api`

Creating the new Accounts API

The NoSQL database service to save the new social information of the customers is selected and created so you have a secure integration between IBM Bluemix and the on-premises database. The next procedure is to create the API that gets the information from the two data sources (on-premises DB2 for personal information and Cloudant for social information) and creates a consolidated JSON model to send to the users.

To create this evolved microservice, follow the strategy described in 3.4.3, “How to refactor Java EE to microservices” on page 43. Here are the steps:

1. At this point, you know what needs to be split, so create a new WAR application responsible for the Accounts information by using Eclipse.

   Note: Dependencies must be added to the project because the monolith application is using the Apache Wink JAX-RS implementation to expose REST services. This implementation is not supported in WebSphere Liberty (since mid-2015) because it started supporting Java EE release 7, which provides its own native JAX-RS implementation.

   a. Create a Dynamic Web Project.

b. In the New Dynamic Web Project options window select **WebSphere Application Server Liberty** as the Target Runtime, clear the **Add Project to an EAR** check box, and click **Modify** for the Configuration option.

c. Select these options and then click **OK** (Figure 4-24):

- Dynamic Web Module
- Java, JavaScript
- JAX-RS (Web Services)
- JAXB
- JPA

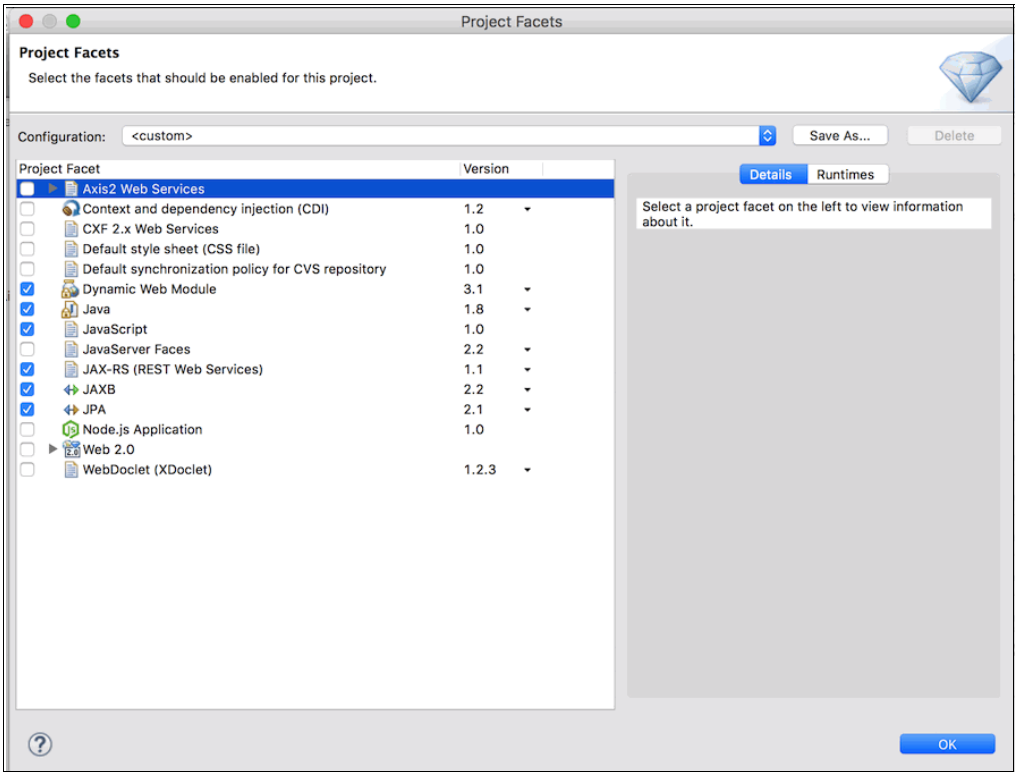


Figure 4-24 Dynamic Web Project configuration

d. Click **Finish**.

www.dbooks.org

The New Dynamic Web Project configuration window opens (Figure 4-25).

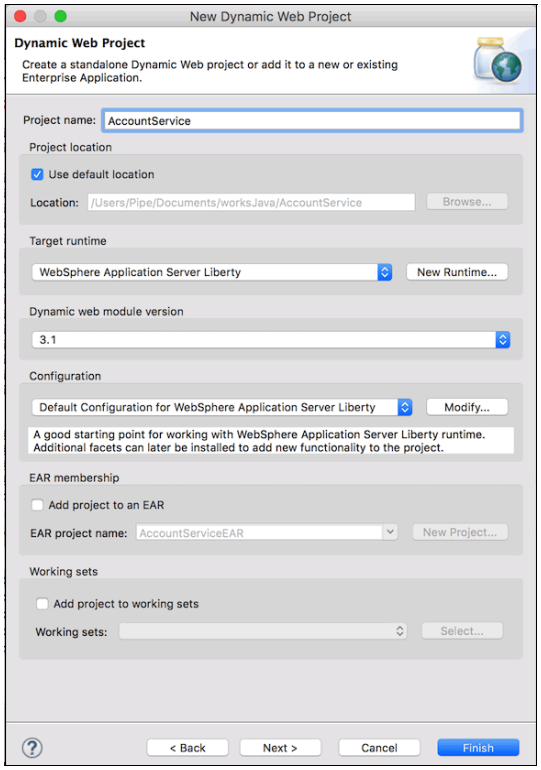


Figure 4-25 Dynamic Web Project page

2. Create the same package structure as in the monolith application to save the standards and practices used by the Java development team.
3. Copy all the code that is related to Accounts from the monolith project and put it in the new Java project.
4. In the new project, modify the source code to consolidate the social network information in the Cloudant database with the DB2 information.
5. Create a new REST service to get the complete user information.

For this microservice, the AccountService application was developed. For information about how to get the application, see Appendix A, “Additional material” on page 119.

5

Security and governance

With monolithic architecture, governance and security are simplified because of the relatively isolated runtime environment on which the application is operated. Microservices architecture, with its defining revolutionary characteristics, brings more challenges to governing activities and protecting the application from security threats.

This chapter includes the following sections:

- Security in microservice architecture
- Governance in microservice architecture
- The case study
- Summary

© Copyright IBM Corp. 2016. All rights reserved.

www.dbooks.org

5.1 Security in microservice architecture

Microservices architecture gains the advantage of flexibility by the defining distributed characteristics where services in the system can be independently developed and deployed in a dispersed manner. A trade-off for that opened architecture from a security point of view is that the system now is more vulnerable because the attack surface is increased. More ports are opened, APIs are exposed, and security, therefore, becomes complicated because it needs to be undertaken at multiple points. This section gives a brief overview of some security aspects to consider when you design a microservices-inspired system.

5.1.1 Authentication and authorization

Authentication and authorization are two core processes that are usually triggered when attempts are made to interact with an IT system. These core processes ensure the security of the system against attacks:

- *Authentication* is the process of confirming that stakeholders of the system are who they claim to be. In the human world, the stakeholder is typically authenticated by providing a user name and password pair. Advanced complex mechanisms exist to perform the authentication; these can include biometric authentication, multi-factors authentication, and others. The object (either human or a particular subsystem) that is being authenticated is usually called the *principal*.
- *Authorization* is the mechanism used to decide what actions a principal is allowed to perform on a system or what resources the principal can access. The authorization process is usually triggered after an authentication process. Often, when a principal is authenticated, information about the principal is provided to help decide what that principal can and cannot do.

In a monolithic application, authentication and authorization are straightforward and common because they are effectively handled by the application; there is no need to have an advanced mechanism to provide a better secure user experience. However, in a microservices architecture with the defining distributed characteristics, more advanced schemes must be embraced to avoid repeated intercepting between service calls to provide credentials. You want to have a single identity for the principal that can be authenticated one time. This single identity simplifies the authentication and authorization processes, leverages automation, and improves scalability.

One well known solution to avoid redundantly providing a user name and password is to use the single sign-on (SSO) method. Several popular implementations for SSO are available that provide this capability. Examples include OpenID Connect, SAML, OAuth2, and others. Each implementation is suitable for a particular business case. Table 5-1 compares the methods.

Table 5-1 Comparison of SAML, OpenID Connect, and OAuth2

Consideration	SAML	OpenID Connect	OAuth2
Authentication	Yes	Yes	Pseudo-authentication
Authorization	Yes	No	Yes
Transport	HTTP Redirect (GET) binding, SAML SOAP binding, HTTP POST binding, and others	HTTP Get, Post	HTTP
Token format	XML	JSON	JSON or SAML2

Consideration	SAML	OpenID Connect	OAuth2
Security risks	XML Signature Wrapping to impersonate any user	Phishing. OAuth 2.0 does not support signature, encryption, channel binding, or client verification. Instead, it relies completely on TLS for confidentiality.	Phishing. OAuth 2.0 does not support signature, encryption, channel binding, or client verification. Instead, it relies completely on TLS for confidentiality.
Best for this use	SSO for enterprise but difficult to adopt for a mobile platform	SSO for public consumer applications	API authorization

One technique to consider when applying SSO in a microservices architecture is to create an SSO gateway and place it between the services and identity providers. The gateway can then proxy for authentication and authorization traffic from the system to outside and from the outside to the system. This technique can help avoid the situation where every microservice has to do handshaking separately with the same identity provider, resulting in redundant traffic. One disadvantage of this technique is the additional coupling to the gateway, which violates the evolutionary principals of microservices architecture.

5.1.2 Service-to-service authentication and authorization

Consider the following service-to-service authentication and authorization practices when you build a security strategy for microservices architecture:

- Trusted perimeter

  One option might be to implicitly trust all calls made among services within a well-known, insolated perimeter. This technique at first seems to be unsafe because of the risks of being opened for the typical man-in-the-middle attack. But in reality, with the availability of mature tooling and technologies exclusively designed for microservices, the technique is becoming more reasonable.

  You can mitigate the risks by making use of containerization technologies like Docker. Much of what Docker is offering allows developers to flexibly maximize the security of the microservices and the application as a whole at different levels. When building the services code, developers can freely use penetration test tools to stress test any part of the build cycle. Because the source for building Docker images are explicitly and declaratively described in the Docker distribution components (the Docker and Docker Compose files), developers can handle the image supply chain easily and enforce security policies as needed. Additionally, the ability to easily harden services to make them immutable by putting them into Docker containers adds strong security insurance for the services.

  With the adoption of software-defined infrastructure, a private network can be created and configured quickly using a scripting language, and a strong security policy can be forced at the network level. Terraform® from HashiCorp is a good example of a scripting language that can help you quickly create your entire infrastructure in different levels from virtual machines, networks, and so on.

- Leverage SSO

  This practice is straightforward if you already have an SSO scheme in place that is being used as an authentication and authorization pattern for your IT system. You can use SSO for internal interaction between services in your microservices architecture. This approach can make use of existing infrastructure and also make the access controlling the services

www.dbooks.org

simpler by centralizing all of the access controls into one enterprise access directory server.

- Hash-Based Message Authentication Code (HMAC) over HTTP

  One commonly used pattern for service-to-service authentication is basic authentication over HTTP. Its use is common because it is straightforward, light weight, and less expensive. One disadvantage, though, is that if it is used with plain HTTP, risks exist that credentials are easily “sniffed.” The risks can be mitigated by forcing the use of HTTPS in every interaction among services. But, the HTTPS-only approach adds much overhead in both the traffic and certificates management. HMAC can be a candidate to be used as an alternative to basic authentication.

  In HMAC, the request body is hashed together with a private key, and the resulting hash is sent along with the request. The other endpoint of the communication then uses its copy of the private key and received request body to recreate the hash. If it matches, it allows the request to be passed over. If the request was tampered with, the hash does not match and the other endpoint knows and reacts appropriately.

- Manage secrets with special purpose services

  To eliminate the overhead of credentials management in a distributed model like microservices architecture and benefit from the rich security for the system you build, one option is to use a comprehensive secrets management tool. This tool allows secrets (for example, passwords, API keys, and certificates) to be stored, dynamically leased, renewed, and revoked. These actions are fundamentally important in microservices because of the defining automation principal. Several initiatives are available in the market that have capabilities in this area; Vault (by HashiCorp) or Keywhiz (by Square) are two examples.

- The confused deputy problem

  When one service (client) tries to call another service (server), after authentication is done, the client starts receiving information from the server as requested. Often, however, to fulfill the requests from the client, the server needs to establish other connections with other downstream services for additional information that the server does not have. In this scenario, the server is trying to access the information from the downstream services on behalf of the client. If a downstream service allows access to the information that the client is not allowed to have, this is a called a *confused deputy problem*.

  You need to be aware of this type of situation and have strategies to overcome the confused deputy problem. A solution can be to securely distribute a comprehensive token that can help identify the client and also can be used to get more insight about what resources the client is authorized to access, accepting the traffic overhead.

5.1.3 Data security

Data is considered the most important asset in your IT system, especially when it is sensitive data. Various mechanisms are available by which secure information can be protected. However, whichever approach you select, be sure to consider the following general guidelines for data security:

- Use well known data encryption mechanisms.

  Theoretically, although no unbreakable data encryption method exists, proven, tested, and commonly used mechanisms are available (such as AES-128 or AES-256, and others). Use these mechanisms as part of your security consideration, rather than internally creating your own methods. Also, keep up to date and patch the libraries you are using that implement the mechanisms.

- Use a comprehensive tooling for managing your keys.

  All effort you invest for data encryption is wasted if your keys are not managed properly. One lead practice is not to store your keys in the same place you store your data. Also, do not let the keys management complexity violate the flexibility principal of your microservices architecture. Try to use comprehensive tooling that is a microservices-inspired design that does not break your continuous integration and continuous delivery pipeline. Vault from HashiCorp or Amazon Key Management Service are examples of “microservices-friendly” keys management tools.

- Align security strategy with business needs.

  Security comes with costs. Do not encrypt everything, rather pick the most valuable or most sensitive data to focus your effort on. Base your security decisions on business needs, and revise them over time because strategic goals can change over time and so can the technologies that are part of your solutions.

5.1.4 Defence in depth

Because no single solution exists that can remedy all security problems at the same time, the best way to strengthen security for a microservices architecture is to combine all available proven techniques together. And more importantly, apply them to different levels of the architecture.

Besides authentication, authorization, and data security, consider the following techniques for further protection:

- Setting up firewalls

  Firewalls are commonly used to look outward and stop unwanted attempts from getting into the system. If used properly, firewalls can significantly filter out malicious attacks and minimize the work to be done at downstream levels inside the system. Firewalls also can be placed at different levels. One commonly used pattern is setting a firewall at the edge of the system and then locally on each host. This might be a runtime environment for a microservice given the high adopting pace of the one-service-per-host model. You can further define some IPtables rules to allow only a certain known IP range to access the host. With firewalls, you can enforce some security enforcement such as access control, rate limiting, HTTPS termination, and so on.

- Network segregation

  The distributed nature of microservices architectures has security challenges. The granularity and highly focused characteristics of each service, however, means you can easily place microservices into different isolated segments in the network to control how they interact with each other. With the adoption of cloud technologies and software-defined infrastructure, the network segregation idea is even easier, in that it can be scripted and automated easily. Technologies like OpenStack have a comprehensive networking stack named Neutron. This stack allows you to quickly provision virtual private networks and place your virtual servers in separate subnets that can be configured to interact with each other the way you want. You can enforce security policies on the private separated networks and strengthen security for your services with scripts by using a raising tool such as Terraform.

- Containerizing the services

  One lead practice that has been widely adopted in building microservices is to embrace immutable infrastructure. This can be achieved by leveraging technologies like Docker to containerize your services. Each microservice can be deployed into a separated Docker container where you can granularity-enforce security policy into each service. This makes the container as a whole immutable, as with creating a hardened read-only deployment

www.dbooks.org

unit. Each base image that a particular service deployed can be put into a scanning process to ensure the security of the runtime environment for the service at a low level.

5.2 Governance in microservice architecture

Generally, governance refers to activities to establish and enforce how the services in a microservices architecture work together to achieve the strategic goals that the system was built for. Lacking cohesive governance against the services in a microservices-based system quickly knocks the microservices initiative off track. Two types of governance are commonly used in software development:

- Design-time governance: Defining and controlling the creation of software components including designs and implementations of the components based on a set of common policies and standards
- Runtime governance: Enforcing the component policies during execution at runtime

In microservices architecture, the services are independently built and deployed. They are highly decoupled services probably built using a variety of technologies and platforms. Therefore, theoretically, defining a common standard for services design and development is not necessary because each service can be built differently as long as it fulfills its focused responsibility. Microservices architecture implies only decentralized runtime governance capabilities. Services must align clearly to a business capability that each of them is responsible for; services must not try to do something outside of their context boundary. Otherwise, the outcomes are that the services are in a tightly coupled architecture, resulting in delivery entropy and cohesion chaos.

Some aspects commonly used in runtime governance for microservices architectures might be monitoring, service discovery, common security requirement, and so on.

5.3 The case study

You created the new Search Catalog and the evolved Accounts microservices in 4.5, “Practical example: Creating the new microservices” on page 63. Now, you can make other changes to integrate the monolith Orders system with the new microservices under a secure environment.

In the example monolith application, the authentication process was made by using basic authentication by the validation of credentials compared with the user name and password stored in the relational database. After that authentication, the user was able to see all the options of the application.

To review, the following are the three major parts of the system, described by the types of data they have, who is (and who is not) allowed to see that data, and how to protect the data:

- Catalog Search microservice

  The Catalog Search microservice has all the information about the products. This information is available to everyone, so the API does not need authentication. Users are allowed to see the catalog without logging in; the catalog is also exposed in HTTP to allow the information to be cached for faster interactions. Thus, for this microservice no changes are necessary. In the future, you might use an API Management solution to expose the Catalog API to other companies under a controlled and secure strategy.

- Accounts microservice

  This microservice has all the information about the clients, and this information is private to each of them. For this microservice, the information must be protected to be sure only authenticated users can see and update only their own information.

- Orders monolith system

  The Orders monolith system must be integrated with the other microservices. You must ensure that only authenticated users can create and view their orders, and that creating or viewing orders of other users is not possible.

Based on this review, you can see that you must protect two of the components. To do this, create an SSO gateway to authenticate the users by using the same authentication method used in the monolith application; configure the Accounts and Orders microservices to allow calls only from this single sign-on. First, expose the monolith application Orders methods as an API inside your cloud environment reachable by this SSO gateway in a secure way. To do this, use the same Secure Gateway created to connect to the DB2 of the monolith system, and the *API Connect* service in IBM Bluemix.

5.3.1 Integrating and securing the monolith system

To integrate the services of the monolith application, you need to create a secure connection from Bluemix to the on-premises data center and then create an API Gateway to present the services inside Bluemix in a secure way.

Connect to the services of the monolith application

In “On-premises database connection” on page 66, you created a Secure Gateway service in Bluemix and configured one destination to have a secure connection with the DB2 database. Now, you can configure a second destination to connect to the REST services exposed by the monolith Orders application by using the following steps:

1. Log in to IBM Bluemix using your IBMid:

   http://www.bluemix.net

2. Create a new Destination:

   a. In the Bluemix console click **your Secure Gateway** service to go to the dashboard.

   b. In the Secure Gateway dashboard, click the gateway that was created to connect to the on-premises data center (see “On-premises database connection” on page 66).

   c. Click **Add Destination**.

   d. In the Add Destination Wizard, select **On-Premises** and click **Next**.

   e. Enter the host name (or IP address) and port of the resource you want to connect to, in this case the local IP address or host name of the application server where the monolith application is deployed.

   f. Select **HTTPS** or **HTTP** as the protocol to connect to the services. Click **Next**.

   g. Select **Mutual Auth** as the authentication method and select the **Auto-generate** option for the certificates. Click **Next**.

      This option ensures a secure authentication between the client that connects and the Secure Gateway client. With this option, you can control which clients can connect to the back-end services by giving them the certificates that are generated by using Transport Layer Security (TLS) mutual authentication.

   h. Leave the IP tables rules empty. Click **Next**.

www.dbooks.org

In this option, you can also create rules to allow connections only from your known IP addresses, for example, the IP addresses from the Development and Quality Assurance teams. In that case, you also must configure the certificates that were generated in step g on page 87.

i. Enter the name you want to use for the destination, in this case **Orders Monolith**, and click **Finish**.

The monolith application gateway window is shown in Figure 5-1. The *Orders Monolith* destination shows a little red hand (at the top right of the tile), which indicates that the access is blocked by the access control list.

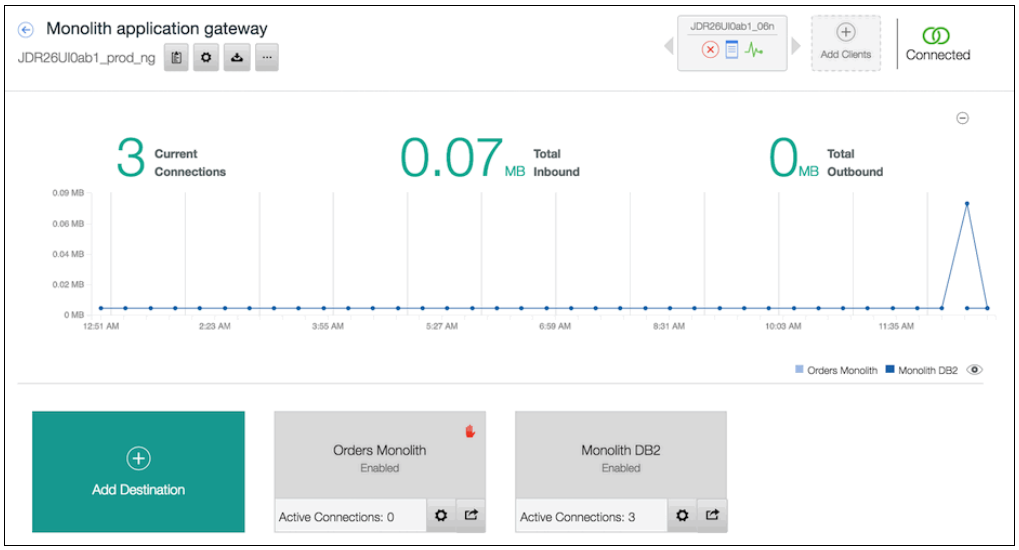


Figure 5-1 Secure gateway dashboard with Database and Services destinations

j. The Orders Monolith destination window, shown in Figure 5-2, shows the Cloud Host and port that are used to connect to the on-premises REST services. Click **Download Authentication Files** (certificates) and save them in a local folder.

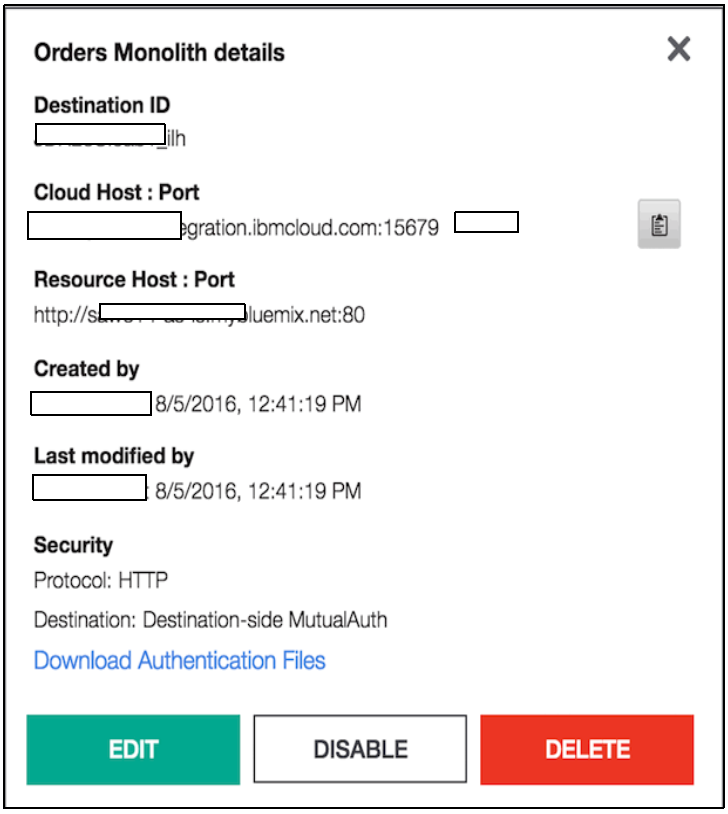


Figure 5-2 Back-end services destination information

k. Configure the access control list (ACL) to allow the traffic to the on-premises services by entering the following command in the gateway client console:

```
acl allow <services ip address or hostname>:<port>
```

For this example, this is the command:

```
acl allow orders.xxxx.net:80
```

3. Create a *keystore* and *truststore* to use the generated certificates:

a. Open an SSH or Telnet client.

b. To create a key store for the client, enter the following command in the directory where you extracted the downloaded certificates:

```
openssl pkcs12 -export -out "sg_key.p12" -in JDR26UIOab1_iIh_destCert.pem
-inkey JDR26UIOab1_iIh_destKey.pem -password pass:"<a password>" -name
BmxCliCert -noiter
```

Now you have the key store, `sg_key.p12`, to use for mutual authentication.

c. To create a truststore with all servers and CA certificates, issue the following command in the directory where you extracted the downloaded certificates:

```
keytool -import -alias PrimaryCA -file DigiCertCA2.pem -storepass password
-keystore sg_trust.jks
(answer ‘yes’ to trust it)
```


www.dbooks.org

```
keytool -import -alias SecondaryCA -file DigiCertTrustedRoot.pem -storepass
password -keystore sg_trust.jks
keytool -import -alias GtwServ -file secureGatewayCert.pem -storepass
password -keystore sg_trust.jks
keytool -importkeystore -srckeystore sg_trust.jks -srcstoretype JKS
-deststoretype PKCS12 -destkeystore key_trust_store.p12
(enter a password for the trustore, repeat the password and leave empty the
source password)
```

Now you have the `sg_trust.jks` and the `key_trust_store.p12` files to use for mutual authentication.

Expose the monolith services as APIs in the cloud environment

In “Connect to the services of the monolith application” on page 87, you created a secure connection between Bluemix and the monolith on-premises application services. Now you can use the API Connect Bluemix service to expose the back-end services as APIs that are easy to use inside your Bluemix space.

1. Log in to IBM Bluemix with your IBMid:

   http://www.bluemix.net

2. Create the API Connect service:
   a. Go to the Bluemix Catalog.
   b. Select **API Connect service** in the *APIs* category.
   a. Click **Create**.
3. Configure a TLS Profile to authenticate with the Secure Gateway client:
   a. In the API Connect dashboard, click the **Navigation to** icon at the top left and then click **Admin** (Figure 5-3).

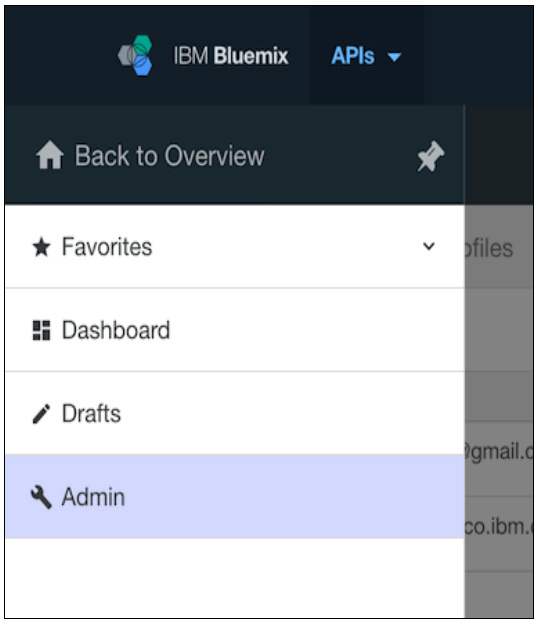


Figure 5-3 Navigation To on API Connect

   b. Click **TLS Profiles** at the top menu and then click **Add**.
   c. Enter the name and description of your TLS Profile.

d. Save the profile by clicking the **Save** option at the top right of the dashboard.

e. Click the plus sign (+) next to the **Present Certificate** option.

f. Click **Select File** and search for the `sg_key.p12` file that you generated in step b on page 89.

g. Enter the password that you configured when you created the `sg_key.p12` file (step b on page 89).

h. Click **Upload**.

i. Validate the certificate by moving the **Request and validate the certificate against the supplied CAs in the truststore** slider to the **On** position.

j. Click **Save**.

k. Click the plus sign (+) next to the **Trust Store** option.

l. Click **Select File** and search for the `key_trust_store.p12` file that you generated in step c on page 89.

m. Enter the password you supplied when you created the `key_trust_store.p12` file.

n. Click **Upload**.

Your TLS Profile now looks like the profile in Figure 5-4.

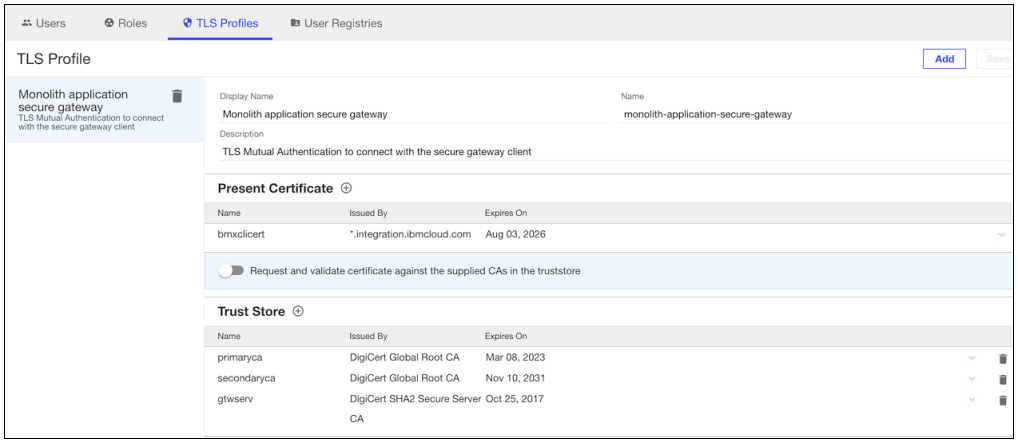


Figure 5-4 TLS Profile configuration for API Connect

4. Create the monolith services APIs by using the Secure Gateway Cloud host with Mutual Authentication:

a. Go to the API Connect Dashboard.

b. In the API Connect Dashboard click the **Navigation to** icon at the top left and then click **Drafts**.

c. Click **APIs**.

d. Click **Add** → **New API**.

e. Enter the API title, select **Add to a new product**, enter the new product title, and then click **Add**.

f. In the Design API option, enter the general information in contact, terms and license, and External Documentation.

g. In the Schemes field, select **HTTP**.

h. In the Host field, enter the Cloud host:port provided by the secure gateway.

www.dbooks.org

i. Enter the base path of the monolith services. In this example, it is the following path:

   `/CustomerOrderServicesWeb/jaxrs/`

j. Select the type of data returned by the services, in this case **application/json**.

k. In the Security Definitions option, keep the clientID (API Key) option as it is.

l. In Security, select option **1 clientID (API Key)**.

m. In the Paths option, enter the service path, in this case `/customer/openorder`, then click **add** operation and select **POST**.

n. For the Parameters option, select the information you want to send to your service, which in this case is the following information:

ID	ID of the order to create
Date	Date of the order
Line Item	JSON object with the product to add to the order
Total	Price of the product

o. Return to the API Dashboard and select **Drafts and the products**.

p. Select the product you created in step e on page 91.

q. Select the options you want to use for this API, which in this case is the following information:

Visibility	Authenticated users
Subscribable by	Authenticated users

r. Click **Stage** at the top right of the window and then select the catalog where you want to publish your API.

s. On the Products tab, click the three dots at the right of the product staged and then click **Publish.** In the Edit visibility, select **Authenticated Users** for both options and then click **Publish**.

t. On the API Connect Dashboard, select the catalog where you staged your API, click the **Developers** tab, and then click **Add Bluemix Organization**.

u. Enter the email address of the user who uses the APIs exposed in this catalog.

v. Accept the invitation to use the API by clicking the link in the email sent by API Connect.

w. Select the organization in which you want to use the API.

Now, you have the Private API to create orders using your monolith application.

Create the SSO gateway

Now you have a secure connection to the monolith application and you exposed its APIs as private services inside the Bluemix Organization. You can create the SSO service that authenticates the users, and then call the Orders APIs and Account microservice on behalf of the users.

5.4 Summary

Figure 5-5 on page 93 shows the final application architecture.

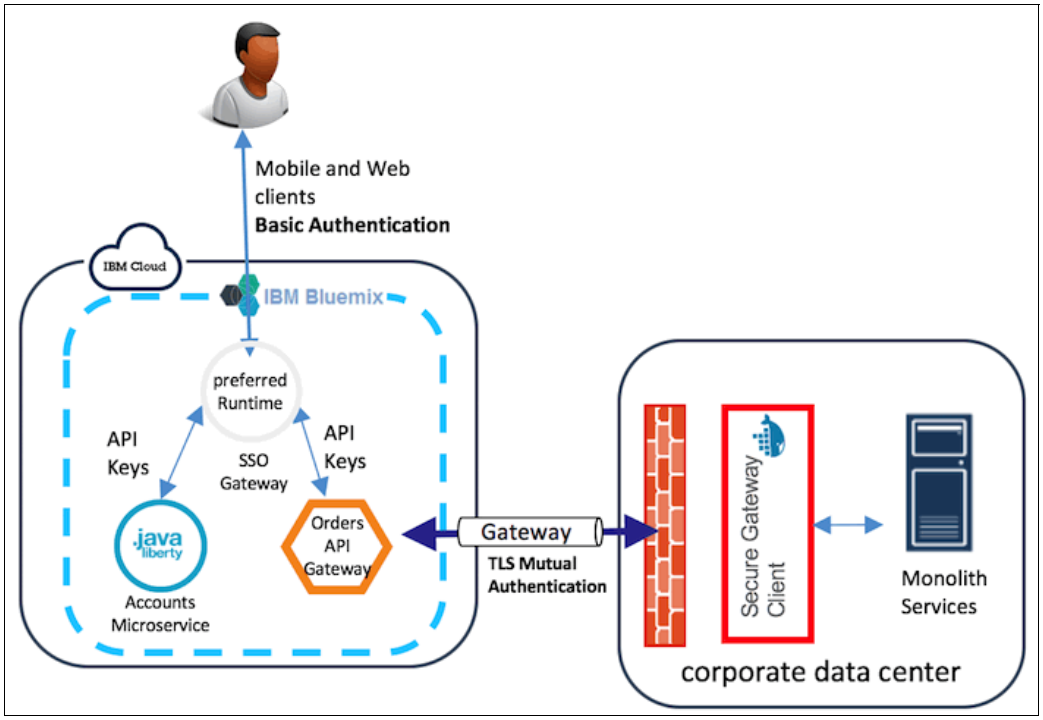


Figure 5-5 Secure architecture

www.dbooks.org

6

Performance

This chapter discusses performance cost, a major concern about moving to microservices. The components that coexisted on the same computing unit now are spread across several locations and no longer can trust each other. This is discussed in Chapter 5, “Security and governance” on page 81.

This chapter includes the following sections:

- Common problems
- Antipatterns to avoid
- Lead practices to improve performance

© Copyright IBM Corp. 2016. All rights reserved.

www.dbooks.org

6.1 Common problems

A big performance concern of breaking apart a monolith is to understand how the individual services might behave when there is an additional networking layer between them.

Because the example in this book is evolving a sample Java application, a good idea is to go back in time to year 2000, when you had only EJBs with remote interfaces. That scenario led to the same situation you deal with when implementing a microservices: Every interaction is chatty and you must be as efficient as possible on those communications.

Figure 6-1 shows an example of how monolith decomposition imposes performance penalties for the new application

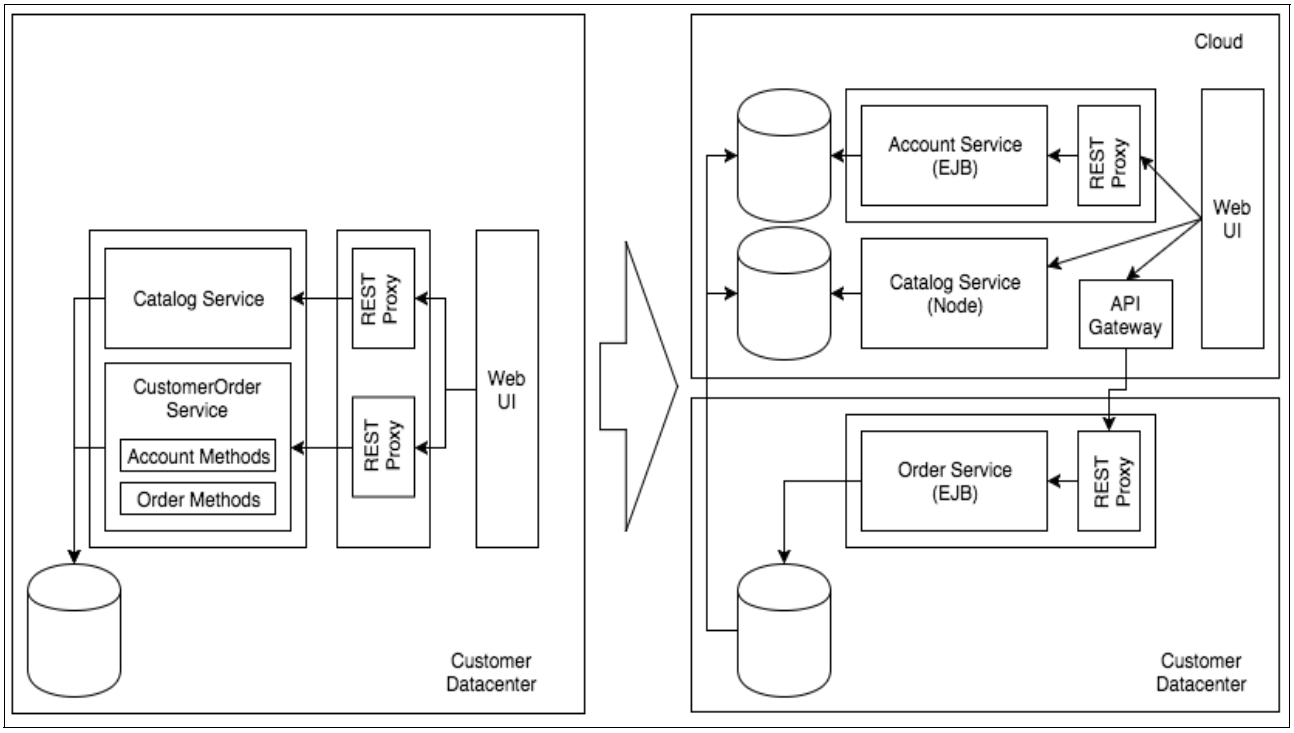


Figure 6-1 Application architectural changes

The figure shows the following information:

- On the left side, all service components are being executed on the same JVM. Hence, when the CustomerOrder service needs a customer's information, a method of that domain was invoked directly within the code, with no extra precaution. That is not the case on the right side. If the information of another domain is needed, a network trip is necessary to fulfill that requirement.
- On the left side, an SKU is added to the shopping cart, calling the Catalog service to obtain an item's price and availability. This only invokes a method on a different class that resides in the same JVM.
- Although not a good practice anywhere, the N+1 anti-pattern is not as problematic on the left as it is on the right side because for each query, a network trip (or even several) is also necessary.

6.2 Antipatterns to avoid

Avoid the patterns described in this section to mitigate the risks of causing a performance problem in a microservices architecture system.

6.2.1 N+1 service call pattern

When a *client* service makes calls to another *server* service to fulfill its responsibility, a common pitfall can cause a serious performance problem: the client calls create many more round trips than it does between itself and the server. The N+1 service call pattern is one example. For example, perhaps you have a service named Customer and another service named Order. Customer service receives a request to get the amount of all the orders that belong to a particular customer. To fulfill that request, the Customer service makes several calls to the Order service as follows:

1. First call is to get all the orders of the particular customer.
2. After receiving the orders (N orders), the Customer service loops through all the orders and, with each order in the list, it makes another call to Order service to get the amount of that particular order. At the end of the loop, it has made N calls to Order service.
3. The Customer service ends up making N+1 calls to the Order service for fulfilling the original request.

Figure 6-2 on page 97 describes the sequence diagram of the calls being made between Customer and Order services. This practice makes the communication between the two services unnecessarily chatty, which consequently generates a performance problem to the system.

Alternatively, in the second call, the Customer service can get the amount for all customers at the same time and do the lookup later on its side.

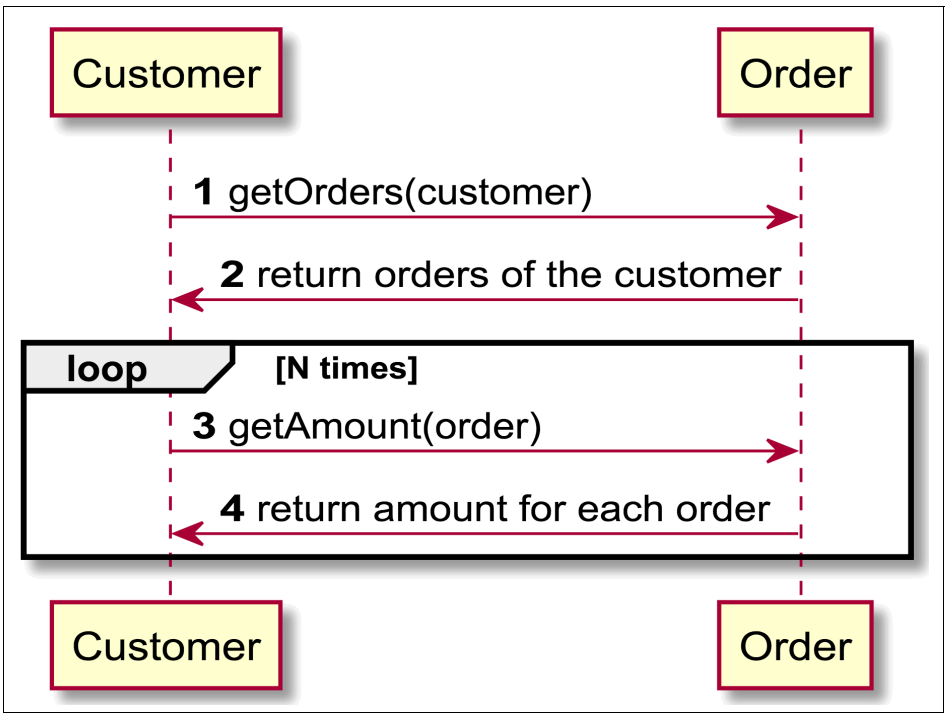


Figure 6-2 Example of N+1 service call anti-pattern

www.dbooks.org

6.2.2 Excessive service calls

A normal practice in microservices architecture is that one service needs to make calls to other surrounding services to fulfill its responsibility. Too many outbound calls like that, however, can generate a performance problem. To say how many is too many depends on how you design services in your system; if an end-to-end case causes a service to interact with tens of other services for fulfilling a particular request, that is a candidate to reconsider the bounding context of that service.

In microservices, a common practice is to create microcosmos or silos within the organization, with several teams working on different services and having little or no communication flowing between them.

When you read about the N+1 situation you might have thought, “That does not happen here because we know how to make services.” You are probably correct, to a certain extent. Consider the scenario illustrated in Figure 6-3 when you migrate from a Java application.

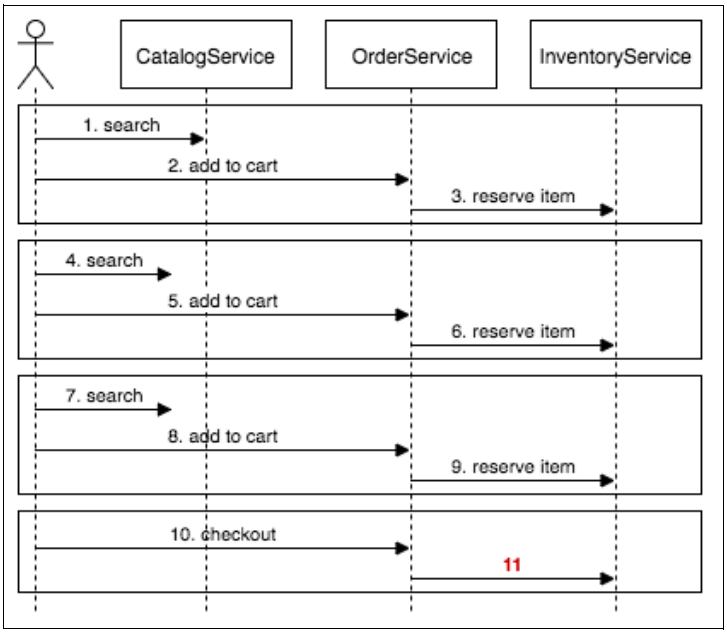


Figure 6-3 User shopping on website

This simplified sequence diagram shows a user navigating and shopping on your website. The user is searching and adding products to the shopping cart and now wants to check out. After payment confirmation (highlighted in step 11 in Figure 6-3), you must deduct those products from the inventory.

In a monolithic java application, although not a best practice, having a method signature **“**`deductFromInventory(sku, quantity)`**”** might not be a bad practice, because all the code is being executed inside the same JVM.

When a network trip must be made for each and every call, the response time of your service can be impacted considerably. Therefore, you must create interfaces (methods) that make the communication between the peers less chatty. Example 6-1 shows such an interface.

Example 6-1 Single batch execution

```
public class DeductionItem {
            SKU id;
   Int quantity;
}
deductFromInventory(DeductionItem[]);
```

In Example 6-1, instead of calling the method to remove items from inventory several times, one for each stock keeping unit (SKU), a single batch execution is called, with all items that must be deducted being sent, at the same time.

Under the “InventoryService” team perspective, providing only the original method can suffice the needs from the service consumer (caller), but this is where “understanding the whole picture” concept lands: If you do not understand the caller’s needs, you might provide inefficient interfaces for the service consumer’s case.

6.2.3 Garrulous services

Figure 6-3 shows that a new object was created to send data to the InventoryService, and that decision was made because of the same principle or limitation regarding network calls: Keeping interactions as simple and efficient as possible.

In a monolith application, a common approach is to create that method using the following signature:

```
deductFromInventory(shoppingCart);
```

Again, although not impractical in the monolithic world, with microservices it might result in much unnecessary information being sent over the network between peers. That scenario must be avoided to increase the communication performance.

Another common situation is having a `getUserProfile(userId)` method that returns even the user’s nick name and other unnecessary information to the consumer that is calling to your service. That situation must be avoided in microservices. A preferable way is to have specializations of that method to return only the information necessary for that particular call.

That does not mean that you must grow your microservice into a macroservice. Rather, you must balance the implementation to be more effective. For instance, you might provide the endpoints that are listed in Table 6-1.

Table 6-1 Endpoints and type of data returned

Endpoint	Type of information returned
/user/{userId}	All information regarding that particular user
/user/{userId}/address	Only the user’s known addresses
/user/{userid}/address/home	Only the user’s home address

Use of endpoints can convert your microservices to a huge component so use this suggestion cautiously: Implement these alternative endpoints only when the need from a caller exists.

www.dbooks.org

6.2.4 Lack of caching

If you keep adding services into the system and simply expose data only through APIs for other services to consume, it is usually not enough. An example is handling calls from other services by making redundant database accesses instead of caching data across multiple service calls to avoid extra database accesses and data round trips to and from the database. Data accesses are usually expensive in terms of performance impact.

Also, you must be careful in the use of caching. Use it sparingly across services. In general, data must only be written to by one service at any one time. Controlling cache logic and cache inconsistencies is a challenging task. An alternative approach is to use service stores that can synchronize with your application that feeds the data as needed in a leasing model.

6.3 Lead practices to improve performance

This section presents suggestions to improve the performance of the resulting microservices system in the transformation journey from a monolithic architecture. Significant impact caused by the network isolation introduced between the services can potentially be reduced.

6.3.1 Let the service own its operational data

Data access performance is usually one of the most challenging aspects to conquer before having a performant system. Microservices architecture uses a distributed model, where each service is usually responsible for a small, focused domain problem. Each service calls to other services to fulfill its business responsibility as needed. Generally, to improve system performance, you must be sure each service in the system is always in its best performance status.

One practice for improving performance of a particular service is to place the data and logic close together in a trade-off cohesive environment; that is, avoid separation of logic and data components.

6.3.2 Avoid chatty security enforcement

Often, a security enforcement produces a performance tradeoff. Certain patterns and best practices that, if you can apply appropriately into your architecture, can help to harmonize security and performance together. One example is to minimize interaction and round trips of data between one service and another service in the system, or between a service and an external service.

For more detail about the security aspect in a microservices architecture and more options that are suitable for your needs, see Chapter 5, “Security and governance” on page 81.

6.3.3 Leverage messaging and REST

Messaging is a critical component to application development, enabling developers to divide an application into its components, and loosely couple them together.

Messaging supports asynchronous coupling between the components; messages are sent by using message queues or topics.

A messaging publish/subscribe pattern reduces 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 is available.

REST and messaging do not have to be an either/or proposition. The two approaches can complement each other. In some cases, an optimal approach is to combine applications performing REST calls with applications that use message transport to achieve a combination 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 do not 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, a preferred approach is often 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 more efficient 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, a high availability (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, because failures might occur at some point for either method. Acting in conjunction 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 send messages to notify about data changes, and clients who subscribe to 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 forward slash (/) character delimiters 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, use of the endpoint 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

www.dbooks.org

also subscribe for the topic that matches the endpoint syntax. With this technique, you can reduce REST frequent polling and still have the application be up-to-date.

By using messaging in communication among services in the system, you can potentially improve the throughput of the data being communicated and see better performance.

6.3.4 Fail fast and gracefully

In microservices architecture, because services are supposed to work together in fulfilling requests, risks of failures can be everywhere. One service might be out of order, a network might be unreliable at some point causing calls between services to fail, and so on. For these reasons, when you design for microservices architecture, you need keep a “design for failure” mind set. The design must cover a failure situation so that when it happens to particular services, they better fail fast rather than slowly responding and causing resource-holding and bottleneck problems. Quick failures lead to graceful response and to better understanding and problem solving.

The following design patterns, among others, can be used to gracefully handle failures:

- Circuit breaker

  The circuit breaker pattern is commonly used to ensure that when a failure occurs, the failed service does not adversely affect the entire system. This might happen if the volume of calls to the failed service is high, and for each call, you must wait for a timeout to occur before moving on. Making the call to the failed service and waiting uses resources that eventually render the overall system unstable. The circuit breaker pattern behaves just like a circuit breaker in your home electrical system. It “trips” (shuts off) 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 attempt is reached. At that point, the circuit breaker for that service trips, and any further calls are short-circuited and do not result in calls to the failed service. This setup saves valuable resources and maintains the overall stability of the system.

- 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, the failure is limited to that bulkhead alone, rather than taking down the entire ship. This kind of partitioning approach of isolating failures to small portions of the system can be used in software also. The service boundary (that is, the microservice itself) serves as a bulkhead to isolate any failures. Dividing functionality (as is also done 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. As an example, consider a thread pool being used to reach two existing systems. If one of the existing systems starts to experience a slowing down and causes the thread pool to become exhausted, access to the other existing system is also affected. Having separate thread pools ensures that a slowdown in one existing system exhausts only its own thread pool and does not affect access to the other existing system.

7

DevOps and automation

This chapter describes the operational practices you can adopt to be successful when implementing a microservices-oriented application. By adopting this construction paradigm, the complexity of each deployment decreases. However, the number of decoupled and independent deployments you have to make are so large that you cannot achieve them without proper automation.

This chapter focuses on how to implement DevOps practices when evolving your application to a microservices set.

This chapter includes the following sections:

- Deployment automation
- Monitoring
- Performance

© Copyright IBM Corp. 2016. All rights reserved.

www.dbooks.org

7.1 Deployment automation

This section describes how to implement continuous integration and continuous delivery of the recently migrated components. An analogy for this topic is the car manufacturing process: A car is built in several dependent and orchestrated steps. To develop microservices, you use an analogous “delivery pipeline.”

In developing microservices, each stage needs special care and tools. Although by regulation you might be obliged to have human approval or validation in the middle of the process, most of the process must be fully automated. This type of development and operations is called DevOps.

DevOps promotes collaboration and automation in the development process. For more information about DevOps, see the *DevOps for Dummies* book at the following website:

https://public.dhe.ibm.com/common/ssi/ecm/ra/en/ram14026usen/RAM14026USEN.PDF

7.1.1 Development

At this stage the traditional toolset used by developers to construct an application does not change. The cornerstone of an automated deployment process, however, is to have all your codebase persisted on a source code management tool. This tool might be IBM Rational® Team Concert™, Bluemix DevOps Services, or GitHub.

Figure 7-1 shows the Bluemix DevOps Services project used to host the application code. It can be accessed through Git protocols.

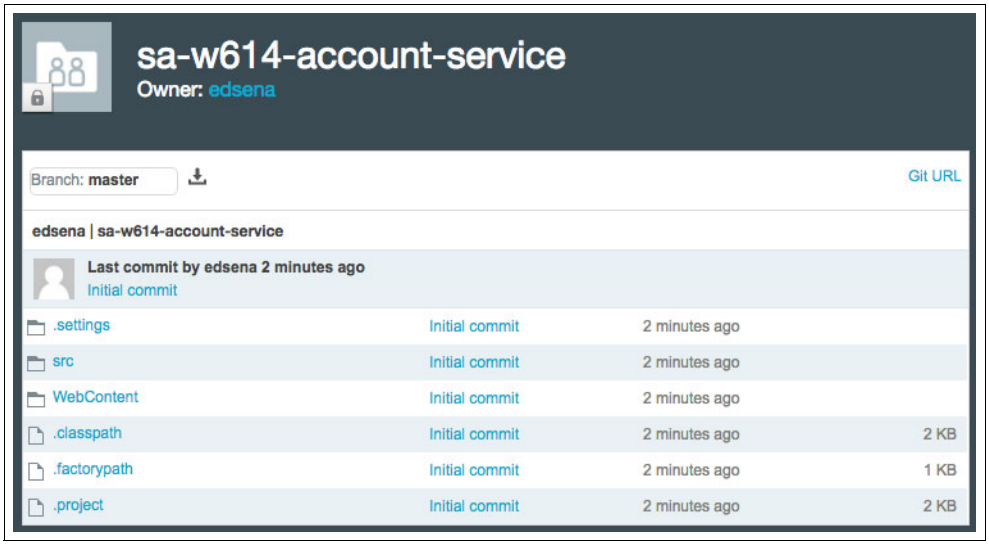


Figure 7-1 Application files uploaded to the code repository

7.1.2 Build

Often, the build stage of a monolith is done by one developer who *exports an EAR file* from the Eclipse environment. That developer then sends this file through email to an administrator who deploys it. Although this section focuses on developers, challenges to administrators during the development process are discussed In 7.1.4, “Deploy” on page 109.

The human-based process can introduce errors because each developer of a team might export the application differently from the way other developers export it. Errors can be increased if configuration data exists inside that application. This configuration data might be properties files with hardcoded URLs or resources.

The following figures show a delivery pipeline created on Bluemix DevOps services. This pipeline automatically builds the Account microservice into a WAR file each time a developer commits a change to the project:

- Figure 7-2, Configuring the input of the build stages
- Figure 7-3 on page 106, Adding a job to the stage
- Figure 7-4 on page 107, The build result
- Figure 7-5 on page 107, The artifacts produced by the build stage

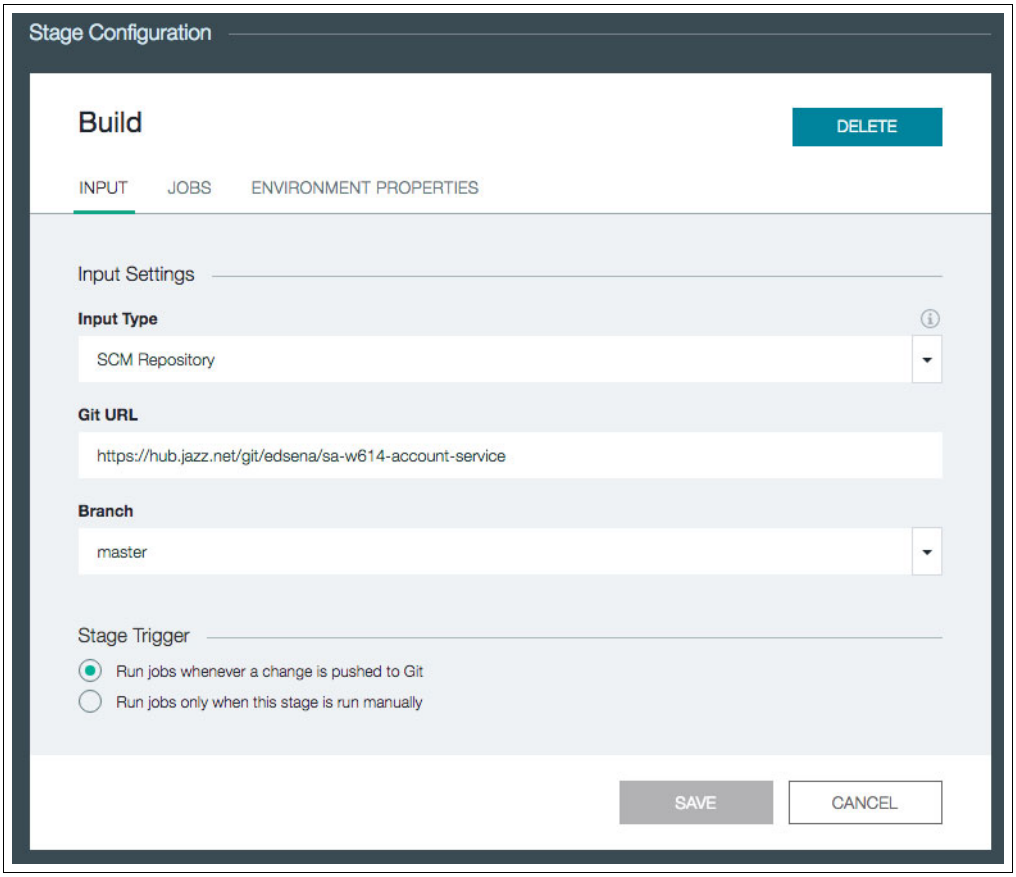


Figure 7-2 Configuring the input of the build stages

www.dbooks.org

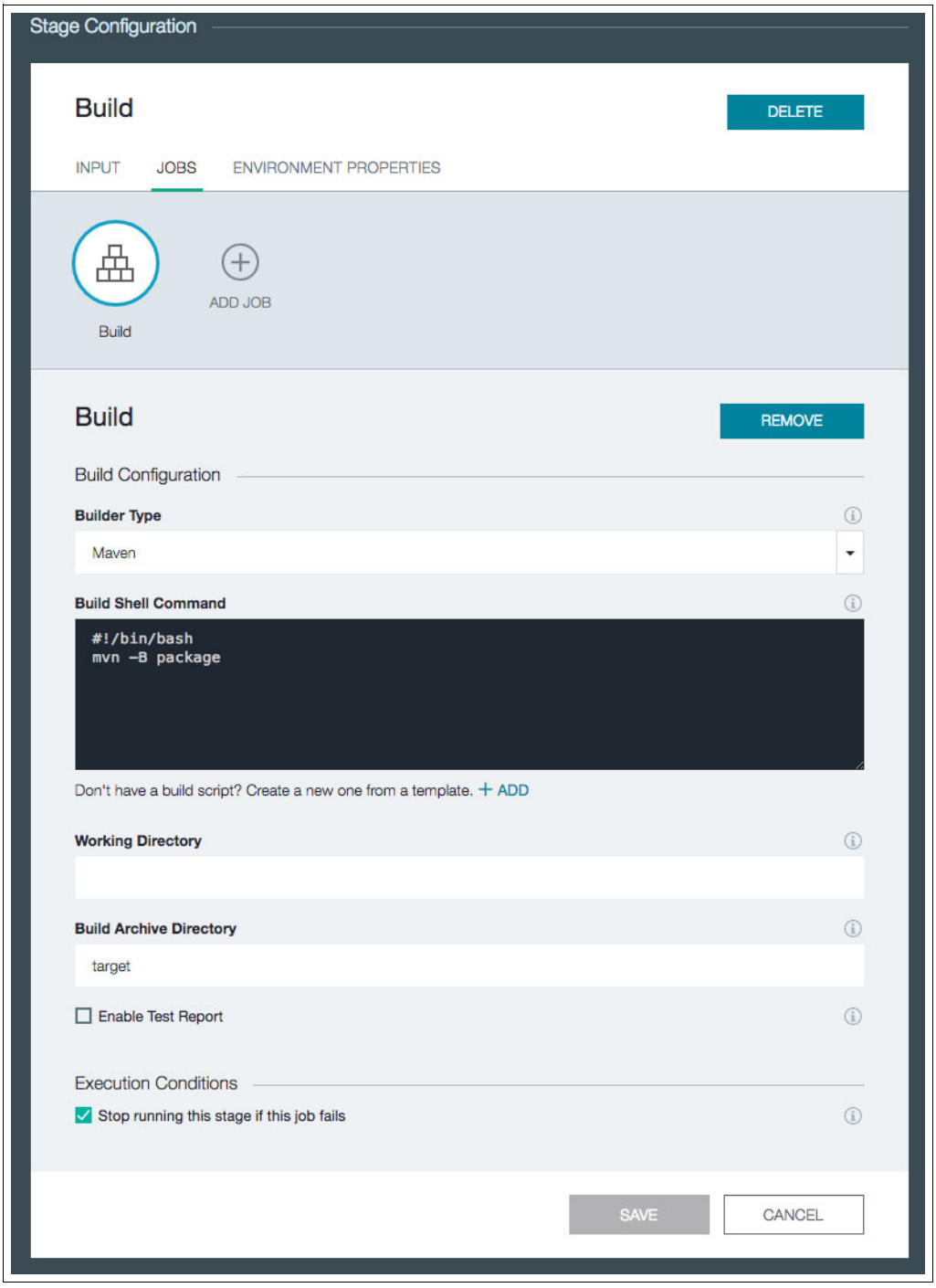


Figure 7-3 Adding a compile (build) job using maven type to the stage

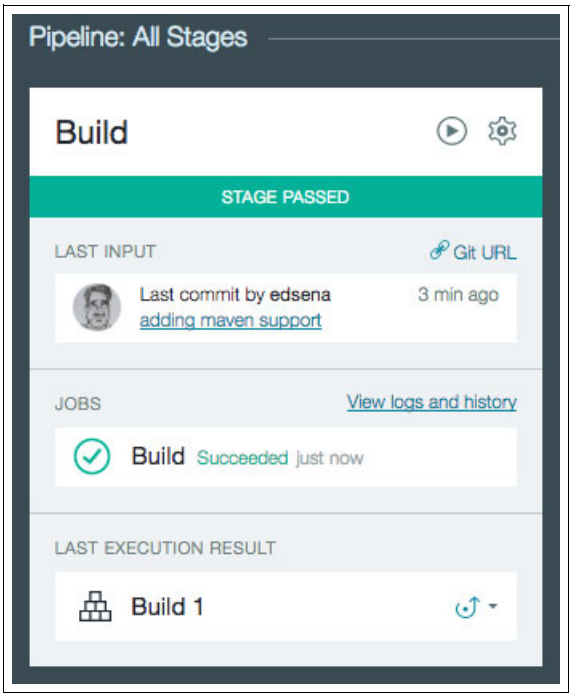


Figure 7-4 Build result

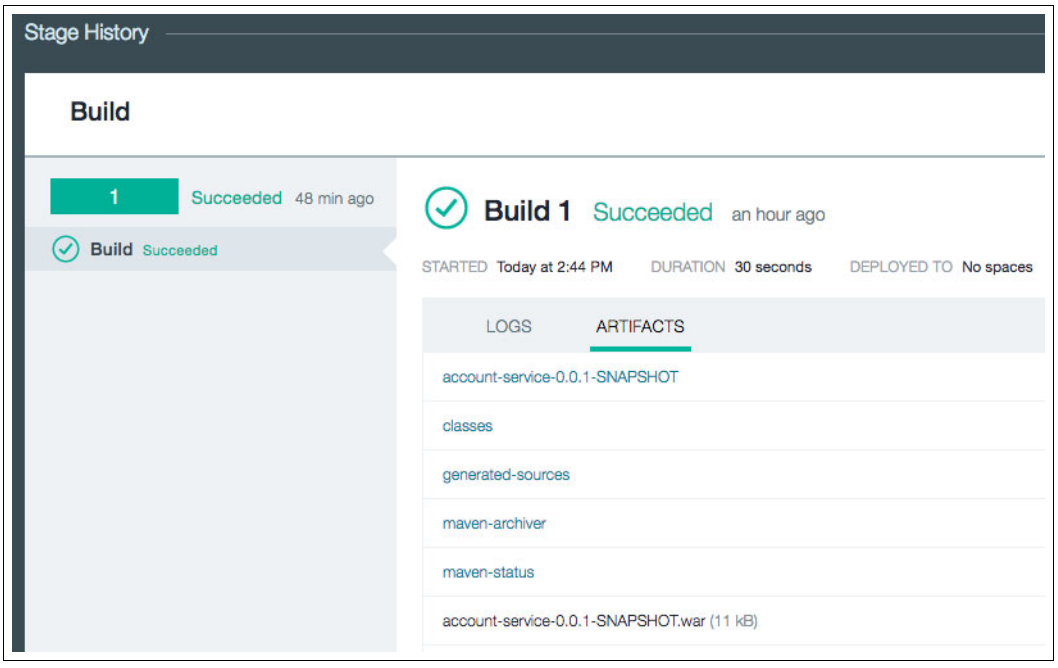


Figure 7-5 The artifacts produced by the build stage

www.dbooks.org

7.1.3 Test

Testing a monolith system is complex and difficult. Even a small change in one of its sub-components dictates several regression tests on the entire system. These regression tests can take days or weeks. If you do not have automated tests and you rely on a team of people to navigate through the application using scripts for testing, testing might take months.

One of the advantages of evolving to microservices is that when you change one piece of code, your unit test scope is much smaller than the unit test scope of a monolith system. However, that does not speed the process if you try to run all of your tests manually. Automation is key for scaling the deployment process, hence, you need to rely on one test framework that automatically provides reports.

Figure 7-6 shows the changed delivery pipeline calling an automated white box security test that is provided by an IBM Bluemix service named IBM Application Security on Cloud.

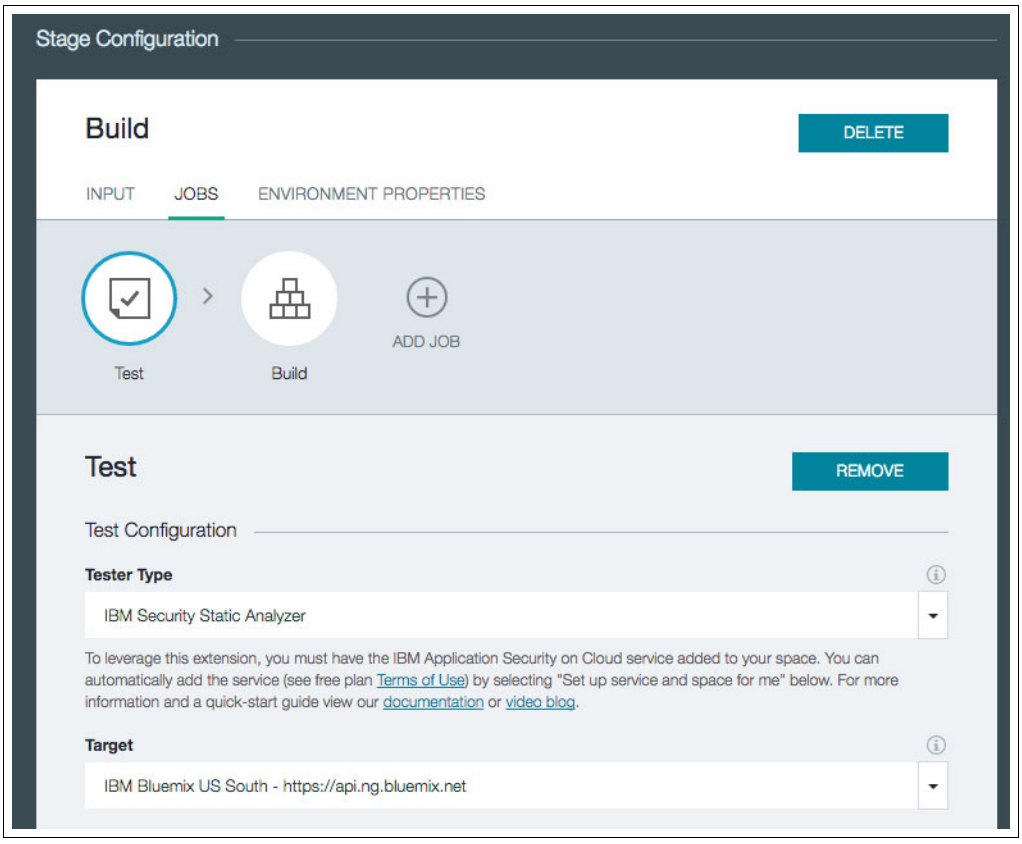


Figure 7-6 Stage configuration with Test job selected

The Application Security on Cloud dashboard (Figure 7-7) displays the results of the test, classifying by the severity of the threats that are found.

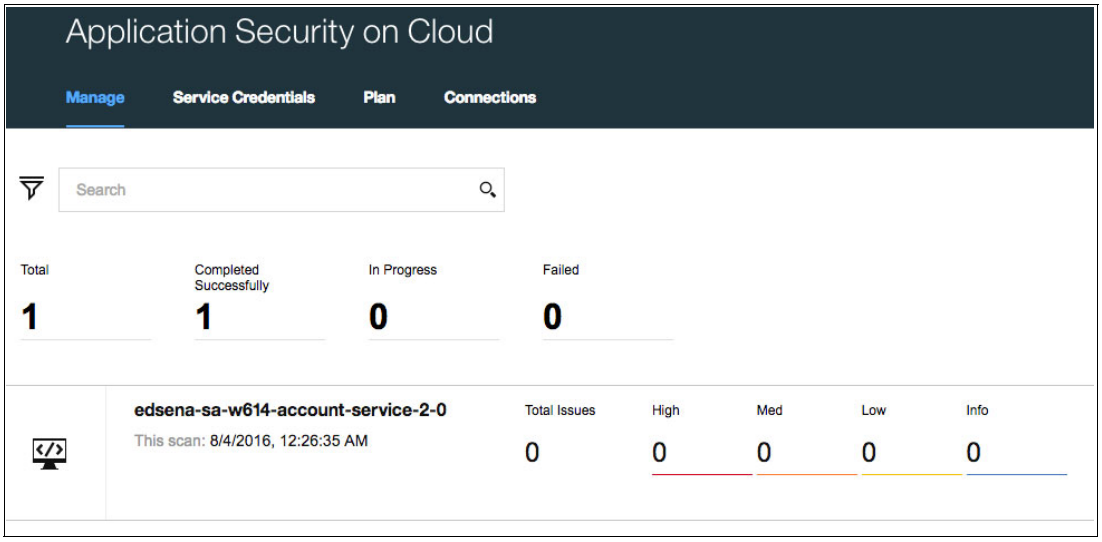


Figure 7-7 Application Security on Cloud dashboard

Important: Although your unit tests become simpler with microservices, in general, your other tests grow in complexity. Because your integration tests touch many more components, an important approach is to implement a suite of automated tests that simulate real users navigating in your applications. With that approach, understanding behavioral changes in the application and cross-referencing them with recent modifications on the supporting microservices becomes less complicated.

7.1.4 Deploy

This section describes how to configure an automated delivery pipeline for your application.

Regarding the manual process described in 7.1.1, “Development” on page 104, a more difficult issue exists than those that are described in that section. That issue is that the process might work only if the administrator owns a handful of applications and has a few deployments to do.

With hundreds of applications (in deployment, each microservice is an application) and an uncountable number of deployments taking place simultaneously, this job cannot be done manually.

Automation is key, and the only way to succeed in this evolution is to have everything from development through deployment automated. This allows you to build and deploy frequently, possibly on a daily basis.

www.dbooks.org

Figure 7-8, shows a new stage being created.

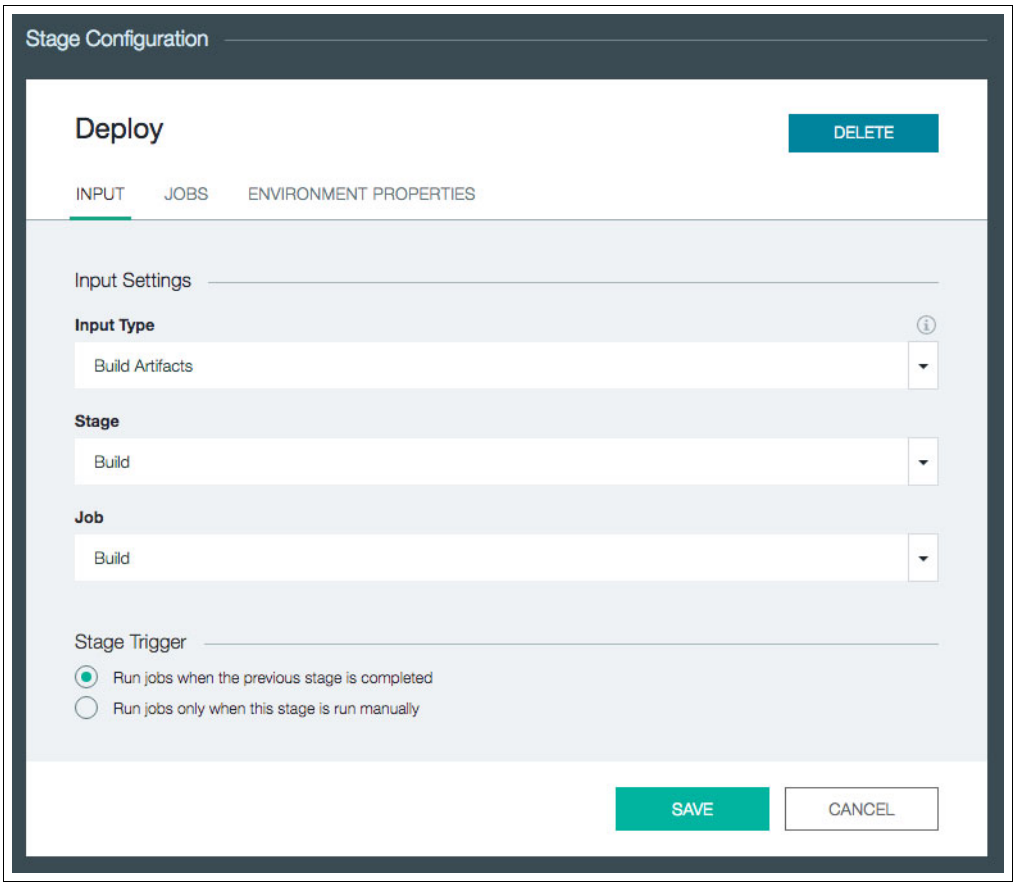


Figure 7-8 Deployment stage using as its input the outcome of the Build stage

Note that this stage uses the output of the last job of the previous stage as its input. Also note that it is configured to be executed automatically upon successful execution of that stage. This allows a “Continuous Integration” scenario for this application because the previous stage is configured to start automatically.

This way, the delivery pipeline automatically tests, builds, and deploys the application on every change.

Figure 7-9 shows that the deployment configuration points to IBM Bluemix.

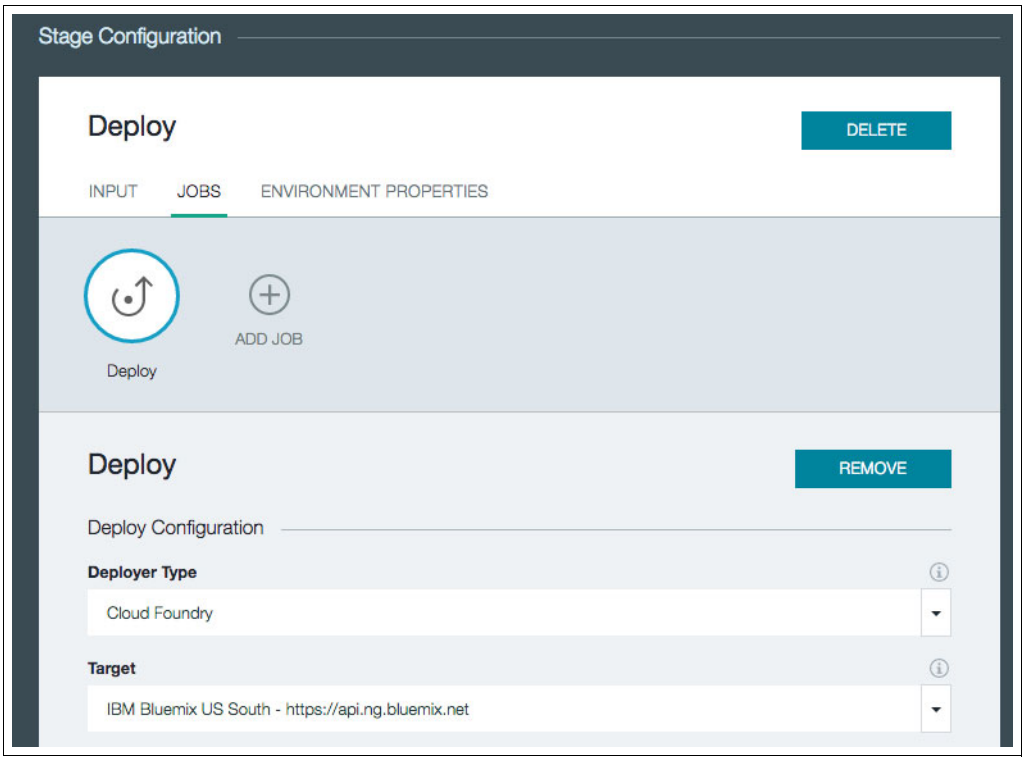


Figure 7-9 Deploy job of Deployment stage with target defined to Bluemix

You can also deploy this application as a Docker container on IBM Bluemix by changing the Deployer Type field from `Cloud Foundry` to `Docker`. You can also choose a Bluemix data center where you want to install the application.

www.dbooks.org

Figure 7-10 shows the final version of the delivery pipeline, after successfully deploying a modification on the application to IBM Bluemix.

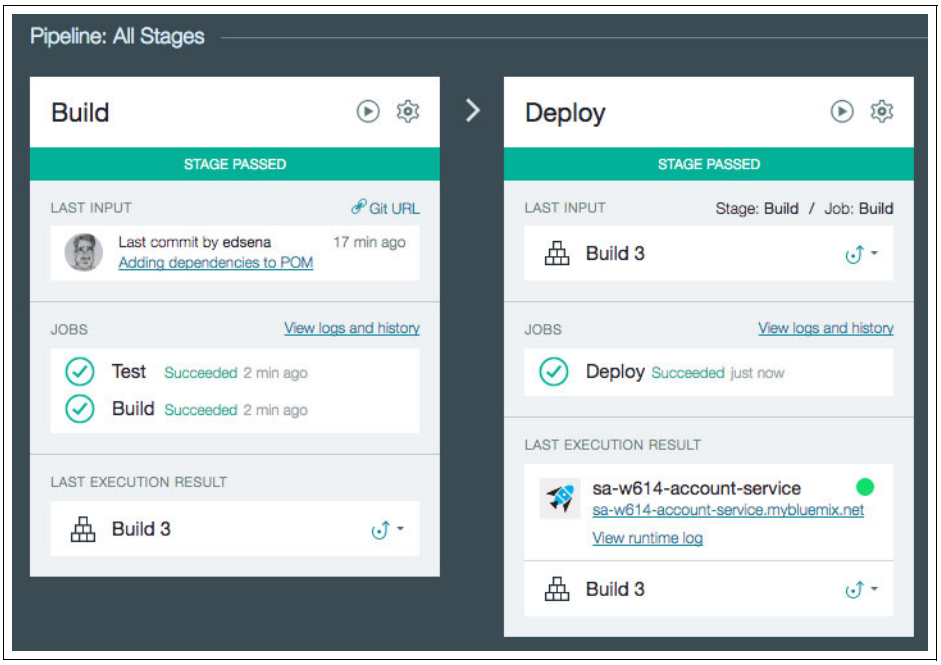


Figure 7-10 Successful execution of the pipeline

7.2 Monitoring

The main difference between a monolithic application and a microservices-oriented one is that in the microservices architecture you have several independent parts that you cannot control by looking at a status indicator on a WebSphere Application Server console.

The evolution from monolith to microservices was caused by a profound change in the way people consume software. Acceptable practices used to be to have yearly releases, response times that were measured in seconds, and downtime and maintenance windows that were measured in hours. Today, however, consumers expect changes to be deployed more quickly.

With microservices architecture, you can have several versions of services running. Understanding the behavioral difference between those versions is crucial to determine which one rolls out for the customer base.

7.2.1 Collecting data

You can never have too much data; the more data you have, the more you understand your application.

When you are monitoring a microservices application, collect more than the usual metrics (for example, CPU and memory) in different points of the execution chain to understand how it works.

For example, collect the following metrics, at a minimum:

- Number of simultaneous requests being served
- Execution time of each request (under the provider perspective)

- Response time of each request (under the consumer perspective)
- Network latency introduced
- Number of simultaneous connections being served
- Token expirations
- Authentication errors
- Execution errors
- Circuit breaker activations
- Number of workers per service
- Load distribution

That information is the minimum information required. In addition to those technical, system-oriented metrics, it is imperative that you also collect information about your customer's behavioral data, such as navigation patterns, clicks, and visualizations to analyze the behaviors of the customers and the application.

7.2.2 Dashboard

After collecting that data, the next important step is to be able to display the data in a way that someone can easily understand where bottlenecks exist and what failures are taking place at a specific moment in time. To display the data, you need a visualization tool that is highly customizable and capable of querying large chunks of data. Frequently, both data collection and visualization are done by the same tool, but that is not a requirement.

Figure 7-11 on page 114 shows the visualization dashboard of the IBM monitoring solution Application Performance Management (APM).

www.dbooks.org



Figure 7-11 IBM APM dashboard

The most important consideration when you plan to implement a dashboard is its purpose. It must not be a “bits and bytes” panel meant only for technical people. The dashboard must have a business continuity view, where the application owners can understand the current behavior of the system. The dashboard must also allow people with technical skills to investigate issues and problematic situations.

7.2.3 Error handling

You must be proactive in terms of error handling and platform recovery because the failure of a single component or microservice might cause the entire system to go down. For example, you must use a circuit breaker pattern (described in 6.3.4, “Fail fast and gracefully” on page 102) implementation to avoid a collapse of the whole system when one of the components fails. However, without proper alerts, you might encounter a situation where you do not know that the short circuit protection was triggered until you start receiving complaints from your customers. Therefore, in addition to dashboards, alerts are crucial for a safe operation of a microservices application.

Figure 7-12 on page 115 shows an example from an Alert Notification service that notifies the relevant people when an incident occurs in your application.

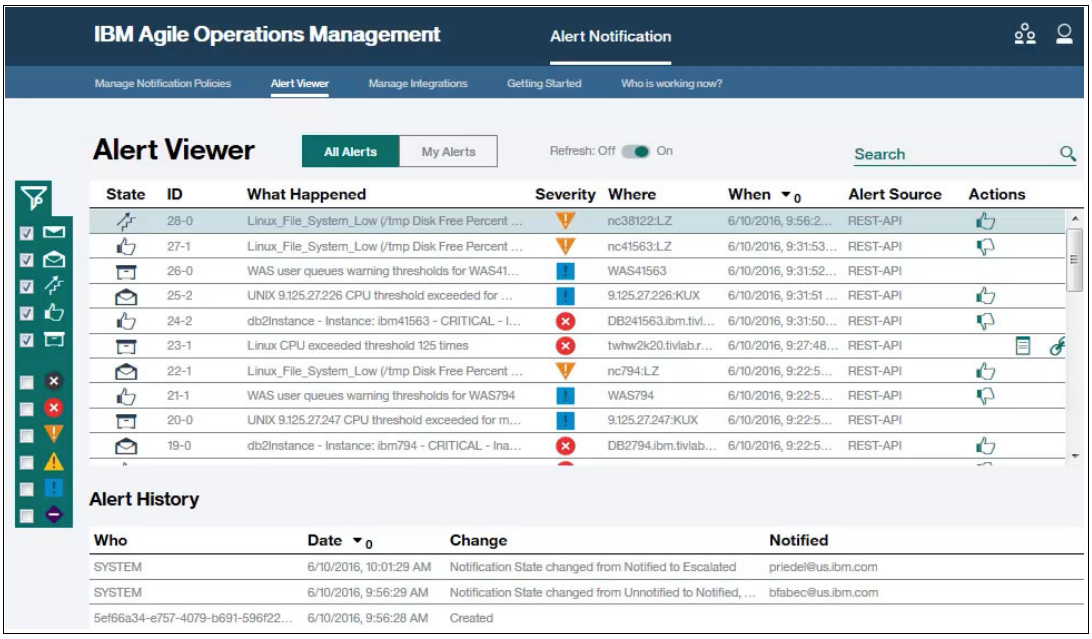


Figure 7-12 IBM Alert viewer dashboard

In addition to error handling and alerts, the way that you write your code influences the overall availability of your application. Remember, *100% availability* does not exist. Hence, this is not a matter of *if*, but *when* your application fails.

Application downtime can be caused by failures of the underlying infrastructure, by the middleware it relies on, or by bad coding. In any case, if you need to provide business continuity, you must plan for those events with proper coding, monitoring, and alerts.

7.3 Performance

Chapter 6, “Performance” on page 95 describes how performance is a concern when you run a microservices-based application. The amount of network communication between the components increases exponentially, and the overall response time increases in a bigger proportion.

Now that you understand what must be done for coding, this section describes what you need to know from the operational perspective.

7.3.1 Automatic scalability

When your services are live, you want them to perform in such a way that your business can adapt to peak loads and seasonal demands. You can attach an auto-scaling policy to your components to allow them to scale horizontally when needed.

Assuming that your microservice is written to protect itself from its dependency failures, when it reaches its processing capacity, it must be allowed to grow. Figure 7-13 on page 116 shows an example of a scalability policy that is attached to a Java microservice on IBM Bluemix using the Auto Scaling service.

www.dbooks.org

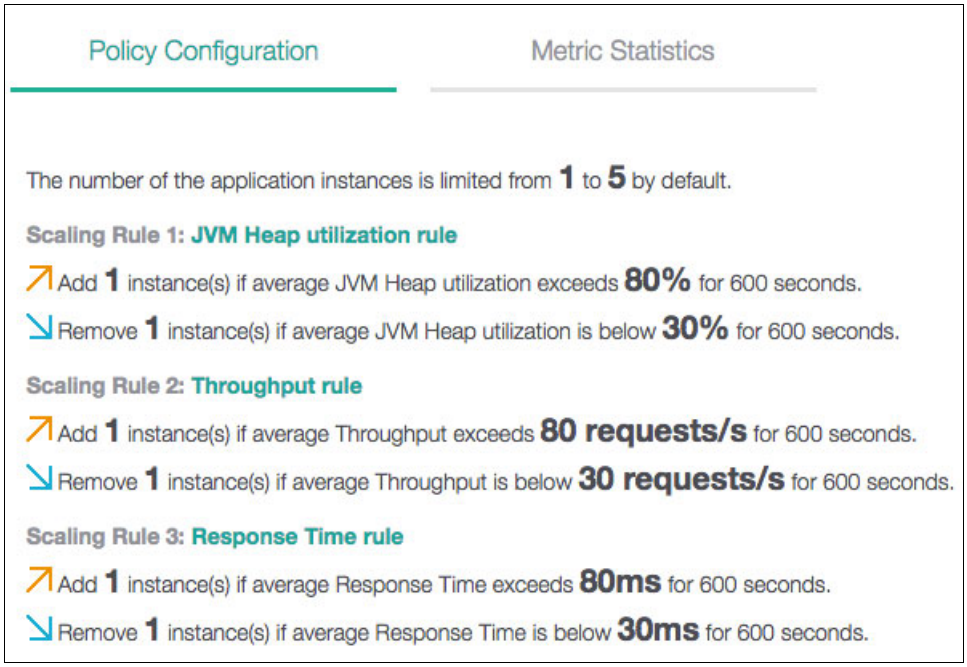


Figure 7-13 Auto-scaling rules

As is evident, several rules are assigned to the service, and each rule takes care of a different aspect:

- Memory

  The usual way of dealing with out-of-memory errors, which is by adding massive amounts of physical memory, cannot be applied in microservices. You must tune your service, understand the amount of memory that is consumed by the application under normal operation and under load, and create a heap than can handle the load at nearly 60% of utilization. That way, you have room to accept more load when the system is scaling.

- Throughput

  You also need to measure the throughput rate that the service can handle without starting to show degradated performance and create a scalability rule that scales horizontally whenever the load reaches 80% of that capacity. That way, you have a buffer to avoid disruption.

- Response time

  Another indicator that the microservice is in need of more power is the increase in response times. However, this metric can mislead you to increase a problem because your service might have been held back by of one of the components (or other services) that it depends on. That is why circuit breaker patterns are important.

The Cloud Foundry infrastructure of Bluemix can help to scale your services automatically by attaching a policy to your application. The platform then monitors and takes actions to guarantee that your service remains stable and uses the minimum amount of resources to save you money.

If you do not plan to run your microservice on Bluemix, consider a scheduling infrastructure that allows your services to grow automatically. Two popular frameworks for that are Apache Mesos and Kubernetes, each with their advantages or disadvantages. For more information, see the following websites:

- http://pdf.th7.cn/down/files/1602/Apache%20Mesos%20Essentials.pdf

- https://www.digitalocean.com/community/tutorials/an-introduction-to-kubernetes

7.3.2 Service discovery

When you create an application composed of countless small components (microservices), you need a way to track where those components are run to be able to communicate with them. Those microservices might be running on ephemeral IPs or ports, they might have been scaled up or down, and they might have moved to another server from the last time you called them. The service discovery pattern can help you track where those components are running.

The *service discovery* is a registry that allows a microservice instance to register itself when it starts. That way, a caller can query the registry to discover the available instances of the service before invoking the service. Figure 7-14 shows a service discovery service from Bluemix with the registration information of three instances of a particular service.

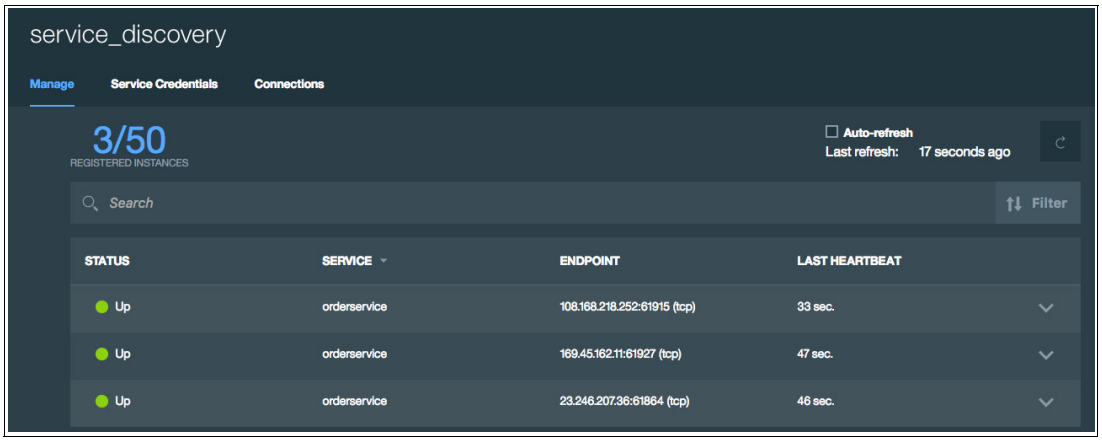


Figure 7-14 Service Discovery dashboard

You must call the REST API of the service discovery when your service instance starts to register it and declare a time to live (TTL) of the registration. After that, in intervals smaller than the TTL, you must call the heart beat API to tell the registry that this particular instance is still alive and ready for receiving requests. If the service discovery does not receive a heart beat in the specified interval, it deletes the service registration information for that instance.

You can finely tune the TTL in accordance with the needs of your application. A TTL interval value that is too long can allow defunct instances into the list, which can trigger short circuits in the callers. An interval that is too short might overwhelm your application with unnecessary heartbeat requests. No guideline exists, so you must choose based on experimentation.

7.3.3 Load balancing

Now the load that is received by your microservices can be split across the instances that are spawned by the auto-scaling mechanism. In 7.3.2, “Service discovery” on page 117, you learned that the service discovery component is capable of holding a list of existing instances of your application and can deliver that list to a caller that needs to invoke your service.

This list solves only part of the problem, which is the discoverability of your service. Callers still need to cache those endpoints, implement a load balancing mechanism across them, and deal with those endpoints that are no longer working despite being in the cache. That is much work and code for basic infrastructure functionality.

www.dbooks.org

IBM Bluemix has a Service Proxy service that can deliver all those features, plus introduce delays and errors, to help you test your application in a real-world scenario. Figure 7-15 shows the management page, indicating the services registered on the service discovery.

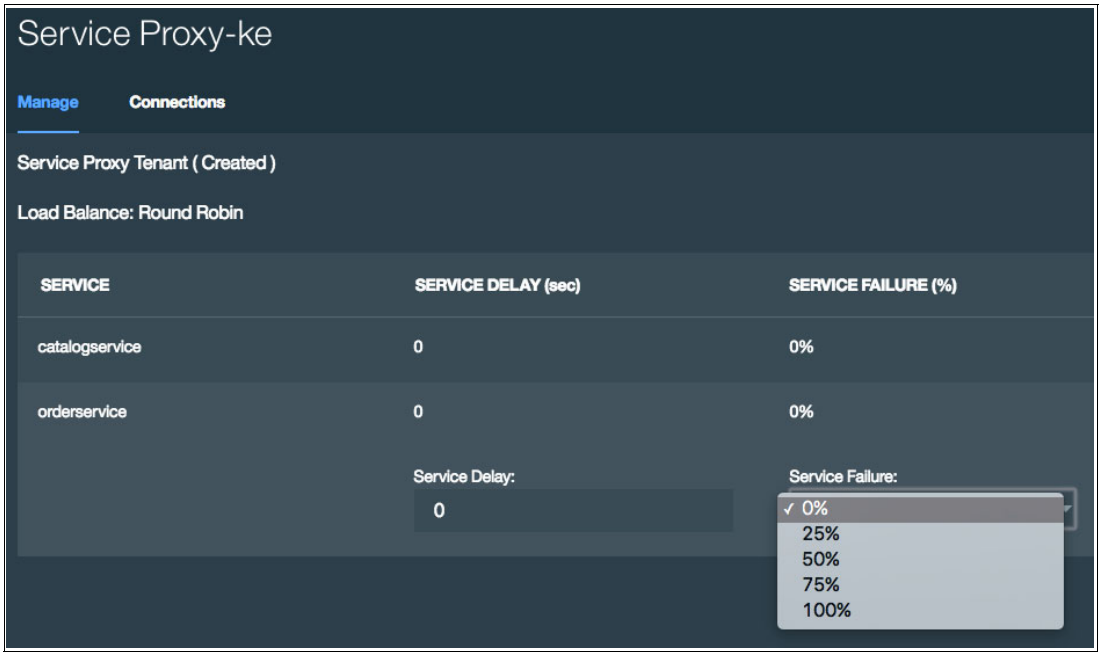


Figure 7-15 Service Proxy dashboard

The Service Proxy instance registers itself with service discovery and uses asynchronous messaging to receive a notification whenever a change occurs in the registry (for example, when a new service instance is registered).

This service acts as a reverse proxy, intermediating all requests between callers and services, simplifying the communication model between your component. Because it also registers itself on the Service Discovery registry, callers can look up its address dynamically to be able to communicate with it.

A

Additional material

This appendix refers to additional material that can be downloaded from the internet.

Locating the Web material

The web material associated with this book is available in softcopy on the internet from the GitHub respository. The repository can be accessed using the following GitHub URL:

`https://github.com/IBMRedbooks/SG248358-Monolith-to-Microservices`

Table 7-1 lists the different projects in the GitHub respository.

Table 7-1 Projects in the GitHub repository

Project	Details
accountmicroservice	Evolved microservice in Java to deploy on WebSphere Liberty using DB2 and Cloudant as data sources
catalogsearch	New Catalog search microservice developed in Node.js to search products in Elasticsearch

Cloning the GitHub repository

To clone the GitHub repository, use the Git clone command:

```
git clone https://github.com/IBMRedbooks/SG248358-Monolith-to-Microservices.git
```

Social networks JSON data model example

Example 7-1 on page 120 is a JSON object created with the relational DB2 `customer_id` and `username`. The information is taken from two of the biggest existing social networks.

© Copyright IBM Corp. 2016. All rights reserved.

www.dbooks.org

Example 7-1 JSON data model

```
"CUSTOMER_ID": "1",
  "USERNAME": "rbarcia",
  "twData": {
    "contributors_enabled": false,
    "created_at": "Sat Dec 14 04:35:55 +0000 2013",
    "default_profile": false,
    "default_profile_image": false,
    "description": "Developer and Platform Relations @XXX. We are developer
advocates. We can't answer all your questions, but we listen to all of them!",
    "entities": {
      "description": {
        "urls": []
      },
      "url": {
        "urls": [
          {
            "display_url": "dev.SN1.com",
            "expanded_url": "https://dev.SN1.com/",
            "indices": [
              0,
              23
            ],
            "url": "https://t.co/66w26cua1O"
          }
        ]
      }
    },
    "favourites_count": 757,
    "follow_request_sent": false,
    "followers_count": 143916,
    "following": false,
    "friends_count": 1484,
    "geo_enabled": true,
    "id": 2244994945,
    "id_str": "2244994945",
    "is_translation_enabled": false,
    "is_translator": false,
    "lang": "en",
    "listed_count": 516,
    "location": "Internet",
    "name": "SN1Dev",
    "notifications": false,
    "profile_background_color": "FFFFFF",
    "profile_background_image_url":
"http://abs.twimg.com/images/themes/theme1/bg.png",
    "profile_background_image_url_https":
"https://abs.twimg.com/images/themes/theme1/bg.png",
    "profile_background_tile": false,
    "profile_banner_url":
"https://pbs.twimg.com/profile_banners/2244994945/1396995246",
    "profile_image_url":
"http://pbs.twimg.com/profile_images/530814764687949824/npQQVkq8_normal.png",
    "profile_image_url_https":
"https://pbs.twimg.com/profile_images/530814764687949824/npQQVkq8_normal.png",
```

```
"profile_link_color": "0084B4",
"profile_location": null,
"profile_sidebar_border_color": "FFFFFF",
"profile_sidebar_fill_color": "DDEEF6",
"profile_text_color": "333333",
"profile_use_background_image": false,
"protected": false,
"screen_name": "SN1Dev",
"status": {
  "contributors": null,
  "coordinates": null,
  "created_at": "Fri Jun 12 19:50:18 +0000 2015",
  "entities": {
    "hashtags": [],
    "symbols": [],
    "urls": [
      {
        "display_url": "github.com/SN1/twi…",
        "expanded_url": "https://github.com/SN1/SN1-for-bigquery",
        "indices": [
          36,
          59
        ],
        "url": "https://t.co/K5orgXzhOM"
      }
    ],
    "user_mentions": [
      {
        "id": 18518601,
        "id_str": "18518601",
        "indices": [
          3,
          13
        ],
        "name": "William Vambenepe",
        "screen_name": "vambenepe"
      }
    ]
  },
  "favorite_count": 0,
  "favorited": false,
  "geo": null,
  "id": 609447655429787600,
  "id_str": "609447655429787648",
  "in_reply_to_screen_name": null,
  "in_reply_to_status_id": null,
  "in_reply_to_status_id_str": null,
  "in_reply_to_user_id": null,
  "in_reply_to_user_id_str": null,
  "lang": "en",
  "place": null,
  "possibly_sensitive": false,
  "retweet_count": 19,
  "retweeted": false,
  "retweeted_status": {
```


www.dbooks.org

```
        "contributors": null,
        "coordinates": null,
        "created_at": "Fri Jun 12 05:19:11 +0000 2015",
        "entities": {
          "hashtags": [],
          "symbols": [],
          "urls": [
            {
              "display_url": "github.com/SN1/twi…",
              "expanded_url": "https://github.com/SN1/SN1-for-bigquery",
              "indices": [
                21,
                44
              ],
              "url": "https://t.co/K5orgXzhOM"
            }
          ],
          "user_mentions": []
        },
        "favorite_count": 23,
        "favorited": false,
        "geo": null,
        "id": 609228428915552300,
        "id_str": "609228428915552257",
        "in_reply_to_screen_name": null,
        "in_reply_to_status_id": null,
        "in_reply_to_status_id_str": null,
        "in_reply_to_user_id": null,
        "in_reply_to_user_id_str": null,
        "lang": "en",
        "place": null,
        "possibly_sensitive": false,
        "retweet_count": 19,
        "retweeted": false,
        "source": "<a href=\"//SN1.com%5C%22\" rel=\"\"nofollow\"\">SN1 Web
Client</a>",
        "text": "SN1 for BigQuery https://t.co/K5orgXzhOM See how easy it is to
stream SN1 data into BigQuery.",
        "truncated": false
      },
      "source": "<a href=\"//SN1.com/download/iphone%5C%22\"
rel=\"\"nofollow\"\">SN1 for iPhone</a>",
      "text": "RT @vambenepe: SN1 for BigQuery https://t.co/K5orgXzhOM See how
easy it is to stream SN1 data into BigQuery.",
      "truncated": false
    },
    "statuses_count": 1279,
    "time_zone": "Pacific Time (US & Canada)",
    "url": "https://t.co/66w26cua1O",
    "utc_offset": -25200,
    "verified": true
  },
  "fbData": {
    "devices": [
      {
```

```
      "hardware": "iPhone",
      "os": "iOS"
    },
    {
      "os": "Android"
    }
  ],
  "education": [
    {
      "school": {
        "id": "231208527081402",
        "name": "Best School"
      },
      "type": "High School",
      "year": {
        "id": "113125125403208",
        "name": "2004"
      },
      "id": "109356269162353"
    },
    {
      "concentration": [
        {
          "id": "101383096608567",
          "name": "Computing Engineer"
        }
      ],
      "school": {
        "id": "109560059062252",
        "name": "Best University"
      },
      "type": "College",
      "year": {
        "id": "144044875610603",
        "name": "2011"
      },
      "id": "109356275829023"
    }
  ],
  "hometown": {
    "id": "102194039822307",
    "name": "Bogotá, Colombia"
  },
  "locale": "es_LA",
  "favorite_athletes": [
    {
      "id": "1603605746517803",
      "name": "FamousAthlete13"
    },
    {
      "id": "115562781814319",
      "name": "Famous Soccer Player"
    }
  ],
  "friends": {
```


www.dbooks.org

```
      "data": [
        {
          "name": "Fulanito de Tal",
          "id": "10152543842351023"
        },
        {
          "name": "Best Friend",
          "id": "695718893"
        },
        {
          "name": "Famous Developer",
          "id": "10152194189655745"
        }
      ],
      "paging": {
        "cursors": {
          "before":
"QVFIUnhBS3BlY2Y1VUhSM2pRX1BYM1JSUXBOWUpXVEpoQWdZAZADZApRDdDZAkYzbVVEUUZANUURCOTRQ
M2IxQ2NHTHM4dTIZD",
          "after":
"QVFIUkF2SWN3X3dWOWdXZAGFRb1FtYTlwS1VaV3IzanhOeHlHOWO4OUdlNjVYUmJ2RXk4LUlNWUZA3eGF
YeVNKdnVEWHoZD"
        }
      },
      "summary": {
        "total_count": 130
      }
    },
    "id": "688727731225203"
  }
```

Related publications

This chapter provides a list of publications that give more detailed information about the topics covered in this book.

Other publications

These publications provide further information:

Martin Abbot and Michael Fisher. *The Art Of Scalability*. 2009.

Eric Evans. *Domain-Driven Design.* 2011.

Martin Fowler. *Strangler application*. 2004.

Scott Ambler and Promod Sadalage. *Refactoring databases: Evolutionary database design*. 2007.

Online resources

These websites also provide further information:

- Apache Mesos:

  http://pdf.th7.cn/down/files/1602/Apache%20Mesos%20Essentials.pdf
- Data sync strategies:

  https://www.elastic.co/blog/found-keeping-elasticsearch-in-sync
- Design pattern:

  http://martinfowler.com/bliki/BoundedContext.html
- DeveloperWorks article Refactoring to microservices:

  https://www.ibm.com/developerworks/cloud/library/cl-refactor-microservices-bluemix-trs-1/index.html
- DevOps for Dummies:

  https://public.dhe.ibm.com/common/ssi/ecm/ra/en/ram14026usen/RAM14026USEN.PDF
- CAP theorem:

  http://ksat.me/a-plain-english-introduction-to-cap-theorem
- Kubernetes:

  https://www.digitalocean.com/community/tutorials/an-introduction-to-kubernetes\

Help from IBM

IBM Support and downloads

ibm.com/support

© Copyright IBM Corp. 2016. All rights reserved.

www.dbooks.org

IBM Global Services

ibm.com/services

www.dbooks.org

Redbooks

Evolve the Monolith to Microservices with Java and Node

(0.2”spine)
0.17”<->0.473”
90<->249 pages

www.dbooks.org



SG24-8358-00

ISBN 0783442119

Printed in U.S.A.

Get connected



