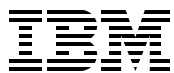


IBM Cloud Private Application Developer's Guide

Ahmed Azraq
Wlodek Dymaczewski
Fernando Ewald
Luca Floris
Rahul Gupta
Vasfi Gucer
Anil Patil
Joshua Packer
Sanjay Singh
Sundaragopal Venkatraman
Zhi Min Wen



In partnership with
IBM Academy of Technology



IBM Redbooks

IBM Cloud Private Application Developer's Guide

April 2019

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

First Edition (April 2019)

This edition applies to IBM Cloud Private Version 3.1.2

© Copyright International Business Machines Corporation 2019. All rights reserved.

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

Contents

Notices	vii
Trademarks	viii
Preface	ix
Authors	x
Now you can become a published author, too!	xiv
Comments welcome	xiv
Stay connected to IBM Redbooks	xiv
Chapter 1. Highly available workloads and deployment on IBM Cloud Private	1
1.1 Highly available workloads on IBM Cloud Private	2
1.1.1 High-availability cluster models for workload deployment	3
1.2 High availability versus failover	6
1.3 Horizontal Pod Autoscaler for IBM Cloud Private	8
1.4 Deploy Spring Boot application on WebSphere Liberty Helm chart	9
1.4.1 Sample application	9
1.4.2 Deploying the application with the WebSphere Liberty Helm chart	12
1.4.3 Deploying with NodePort service exposure	13
1.4.4 Deploying with Ingress Transport Layer Security termination	18
1.5 Zero downtime deployment updates	20
1.5.1 Kubernetes deployment strategy	20
1.5.2 Sample application	21
1.5.3 Deploying the image to IBM Cloud Private	22
1.5.4 Zero downtime update	24
1.5.5 Validation	25
1.6 Other deployment strategies implemented with native Kubernetes	28
1.6.1 Blue/green deployment	28
1.6.2 Canary deployment	31
1.7 Deploying a sample stateful application	34
1.7.1 Prerequisite: Setting up an NFS server	34
1.7.2 Creating a persistence volume and persistence volume claim	36
1.7.3 Creating a Nginx app	42
Chapter 2. Helm and application packaging	49
2.1 Introduction to Helm	50
2.2 Helm chart structure	52
2.2.1 Helm chart metadata	52
2.2.2 Helm templates	53
2.2.3 Helm values	54
2.2.4 Templating language hints and tips	56
2.3 Creating a chart	59
2.3.1 Making your Helm chart flexible	60
2.3.2 Sample application	60
2.3.3 Embedding MongoDB chart as prerequisite	70
2.3.4 Testing Helm charts	73
2.3.5 Upgrading Helm charts	73
2.4 IBM Cloud Paks	74
2.4.1 What is an IBM Cloud Pak?	74
2.4.2 Developing Helm charts for IBM Cloud Private	74

Chapter 3. DevOps and application automation	77
3.1 DevOps overview	78
3.1.1 Benefits of DevOps	78
3.1.2 Continuous business planning	79
3.1.3 Continuous integration and collaborative development	79
3.1.4 Continuous testing	80
3.1.5 Continuous release and deployment	80
3.1.6 Continuous monitoring	80
3.1.7 Continuous customer feedback and optimization	81
3.2 DevOps tooling options	82
3.2.1 Code Editors tooling options	82
3.2.2 Microservice builder tooling options	82
3.2.3 Source code management tooling options	82
3.2.4 Build, Test, and Continuous Integration tools	83
3.3 Introduction to Microclimate	84
3.4 Sample DevOps scenario with Microclimate	84
3.4.1 Installing Microclimate on IBM Cloud Private	84
3.4.2 Creating a hello world app by using Microclimate	88
3.4.3 Creating build and deployment pipelines	92
3.4.4 Continuous integration and deployment in action	94
Chapter 4. Managing your service mesh by using Istio	97
4.1 Introduction	98
4.2 Traffic management and application deployment	98
4.2.1 Setup prerequisites	98
4.2.2 Deploying the application versions 1.0 and 1.1	99
4.2.3 Creating an Istio gateway and destination rule	101
4.2.4 Canary testing with Istio	102
4.2.5 Blue-green testing with Istio	103
4.2.6 A/B testing with Istio	104
4.2.7 Mirroring the traffic by using Istio	106
4.3 Application testing	109
4.3.1 Creating an instance of the Watson Language Translator service	109
4.3.2 Cloning GitHub repository including microservice sample code	111
4.3.3 Deploying and testing the microservice on IBM Cloud Private	112
4.3.4 Defining an Istio egress rule to integrate with an external service	121
4.3.5 Exposing services to be used externally through defining ingress rules	126
4.3.6 Handling an unreliable external service by simulating different failures through Istio abort and delay injection	128
4.3.7 Enhancing the application resiliency by setting request timeouts and adding automatic retry attempts	129
4.4 Enforcing policy controls	131
Chapter 5. Application development with Cloud Foundry	133
5.1 Introduction	134
5.2 What is Cloud Foundry?	134
5.3 Application high availability	134
5.4 Autoscale	135
5.5 Using external services	135
5.6 Application packaging	135
5.7 IBM buildpacks versus community	136
5.8 Zero downtime deployment	136
5.9 Command lines	137
5.10 Sample application to use	137

Appendix A. Additional material	139
Locating the GitHub material	139
Cloning the GitHub material	139
 Related publications	141
IBM Redbooks	141
Online resources	141
Help from IBM	142

Notices

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

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

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

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

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

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

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

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

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

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

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

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


COPYRIGHT LICENSE:

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

Trademarks

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

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

ClearCase®	IBM Cloud™	Redbooks (logo)  ®
Concert®	IBM Watson®	UrbanCode®
DataPower®	Rational®	Watson™
DataStage®	Rational Team Concert™	WebSphere®
Db2®	Redbooks®	
IBM®	Redpaper™	

The following terms are trademarks of other companies:

ITIL is a Registered Trade Mark of AXELOS Limited.

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

Windows, and the Windows logo are trademarks of Microsoft Corporation 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.

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

Preface

IBM® Cloud Private is an application platform for developing and managing containerized applications across hybrid cloud environments, on-premises and public clouds. It is an integrated environment for managing containers that includes the container orchestrator Kubernetes, a private image registry, a management console, and monitoring frameworks.

This IBM Redbooks® publication covers tasks that are performed by IBM Cloud™ Private application developers, such as deploying applications, application packaging with helm, application automation with DevOps, using Microclimate, and managing your service mesh with Istio.

The authors team has many years of experience in implementing IBM Cloud Private and other cloud solutions in production environments. Throughout this book, we used the approach of providing you the recommended practices in those areas.

As part of this project, we also developed several code examples, which can be downloaded from the [IBM Redbooks GitHub web page](#).

If you are an IBM Cloud Private application developer, this book is for you. If you are an IBM Cloud Private systems administrator, you can see the IBM Redbooks publication *IBM Private Cloud Systems Administrator's Guide*, SG248440.

Authors

This book was produced by a team of specialists from around the world working at IBM Redbooks, Austin Center.



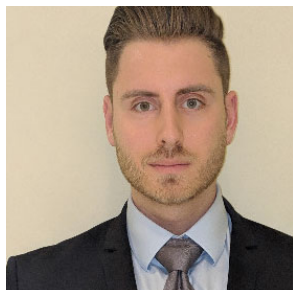
Ahmed Azraq is a Cloud Solutions Leader in IBM. He works as part of IBM Cloud Solutioning Center and his primary responsibility is to help clients across Middle East & Africa (MEA) and Asia Pacific to guide them in adopting IBM Cloud and IBM Watson®. Since joining IBM in 2012, Ahmed worked as a technical team leader, and architect. He has successfully led and won several IBM Cloud Private deals as a pre-sales architect in the region and participated in Academy of Technology studies related to IBM Cloud Private. Ahmed has acquired several professional certifications, including Open Group IT Specialist, and contributed in developing and authoring six professional IBM Cloud Certification Exams. Ahmed also has delivered training on IBM Cloud, DevOps, hybrid cloud Integration, Node.js, and Watson APIs to IBM clients, IBM Business Partners, university students, and professors around the world. He is the recipient of several awards, including Eminence and Excellence Award in the IBM Watson worldwide competition Cognitive Build, the IBM Service Excellence Award for showing excellent client value behaviors, and knowledge-sharing award. Ahmed is a platinum IBM Redbooks publication author as he has authored several other IBM Redbooks publications.



Wlodek Dymaczewski is a Cloud Solution Architect in IBM Poland. He has over 25 years of experience in IT in a field of systems and network management. He was part of the IBM Cloud team since the initial launch of the product, focusing on hybrid cloud management and DevOps solutions. Since 2017, he works as IBM Cloud Private Technical Lead for Central and Eastern Europe. Wlodek holds MSc degree from Pozna University of Technology and MBA degree from Warwick Business School.



Fernando Ewald has 17 years experience with IT solutions. He joined IBM in 2009 as an IT Specialist for IGA Canada - Common Development and Test (CDT), supporting internal IBM Accounts as a member of the Innovation and Technical Leadership Team. Fernando's area of focus is middleware support, including IBM WebSphere® IBM DataPower® Appliances, Information Server IBM DataStage®, reverse proxy, and other products. Before joining IBM, Fernando worked with a Sugar and Alcohol Company, creating high availability solutions for the industry. He also worked as a teacher at Universidade de Franca - Brazil, where he taught Computer Science and System of Information and internet and Network Computer courses. Currently, he works as the IBM Cloud Private Technical Team Lead - L2 Support in IBM Austin.



Luca Floris is a Cloud Technical Consultant in IBM EMEA. Luca joined IBM 4 years ago as a graduate through the IBM Graduate Scheme and he has 8 years of experience in IT, graduating from Plymouth University with a degree in Computer Science. His primary focus is containerization and application modernization through IBM Cloud Private and related technologies. He is well-recognized as a technical focal for IBM Cloud Private from IBM Cloud Innovation Labs in EMEA and has written extensively about IBM Cloud Private on Kubernetes.



Rahul Gupta is a Cloud Native Solutions Architect in IBM Cloud Solutioning Centre in the US. Rahul is an IBM Certified Cloud Architect with 14 years of professional experience in IBM Cloud technologies, such as Internet of Things, Blockchain, and Container Orchestration Platforms. Rahul has been a technical speaker in various conferences worldwide. Rahul has authored several IBM Redbooks publications about messaging, mobile, and cloud computing. Rahul is an IBM Master Inventor and also works on MQTT protocol in OASIS board for open source specifications.



Vasfi Gucer is an IBM Redbooks Project Leader with the IBM International Technical Support Organization. He has more than 23 years of experience in the areas of cloud computing, systems management, networking hardware, and software. He writes extensively and teaches IBM classes worldwide about IBM products. His focus has been on cloud computing for the last three years. Vasfi is also an IBM Certified Senior IT Specialist, Project Management Professional (PMP), IT Infrastructure Library (ITIL) V2 Manager, and ITIL V3 Expert.



Anil Patil is a senior Solution Architect at IBM US. He is a Certified Cloud Solution Architect and Solution Advisor - DevOps with more than 18 years of IT experience in Cognitive Solution, IBM Cloud, Microservices, IBM Watson API, and Cloud-Native Applications. His core experience is in Microservices, AWS, Cloud Integration, API Development, and Solution Architecture. He is Lead Solution Architect and Cloud Architect for various clients in North America. Anil has been an IBM Redbooks publication author and technical contributor for various IBM material and blogs, such as Cognitive Solution, IBM Cloud, API Connect, and Docker Container.



Joshua Packer has been a Senior Software Developer with IBM Cloud technologies for more than 10 years. He has helped architect and implement Public, Dedicated, and Private cloud products. He has over 19 years of software development experience at IBM, working on many different products and collaboration and using several programming languages. He has written a number of internal and external publications (IBM Cloud blog and Medium).



Sanjay Singh is a senior Software Engineer. He has worked on various cloud products over 11 years of his career with IBM. As a technical consultant and a Lab Advocate, he has been engaged with customers in making their Cloud Adoption and implementation successful. He has lead adoption of IBM Cloud Private in one of the largest Telecom providers in India over an eight-month engagement. His core experience is in Kubernetes, OpenStack, AWS, Node.JS, JavaScript, Python, and Scripting. He holds a Btech degree in Computer Science from NIT Trichy (India).



Sundaragopal Venkatraman (Sundar) is a cloud evangelist and a Thought Leader on application infrastructure, application modernization, performance, scalability, and high availability of enterprise solutions. With experience spanning over two decades, he has been recognized as a trusted advisor to various MNC Banks and other IBM clients in India/Asia-Pacific. He is recognized as technical focal point for IBM Cloud Private from IBM Cloud Innovation Labs in India. He has delivered deep dive sessions on IBM Cloud Private on International forums and conducts bootcamps worldwide. He also pioneered IBM Proactive Monitoring toolkit, a light-weight Monitoring solution that was highlighted on IBM showcase events. He has authored various IBM Redbooks and Redpaper™ publications and is a recognized author.



Zhi Min Wen is a senior managing consultant at IBM Singapore. He has over 20 years of experience in the IT industry, specializing in IT service management and cloud. He is passionate about open source solution and he enjoys exploring the new edge of technology. He was an early explorer of Docker and Kubernetes and he has written extensively about IBM Cloud Private and Kubernetes. He attained IBM Outstanding Technical Achievement Awards 2018.

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

Ann Lund, Erica Wazewski
IBM Redbooks, Poughkeepsie Center

Robin Hernandez
Atif Siddiqui
Jeff Brent
Budi Darmawan
Eduardo Patrocinio
Eswara Kosaraju
David A Weilert
Aman Kochhar
Surya V Duggirala
Kevin Xiao
Brian Hernandez
Ling Lan
Eric Schultz
Kevin G Carr
Rick Osowski

Sam Ellis
IBM USA

Juliana Hsu
Radu Mateescu
Stacy Pedersen
Jeffrey Kwong
IBM Canada

Simon Casey
IBM UK

Brad DesAulniers
Hans Kristian Moen
IBM Ireland

Raffaele Stifani
IBM Italy

Ahmed Sayed Hassan
IBM Singapore

Santosh Ananda
Rachappa Goni
Shajeer Mohammed
Sukumar Subburaj
IBM India

Qing Hao
Xiao Yong AZ Zhang
IBM China

The team would like to express thanks to the following IBMers for contributing content to the Cloud Foundry section of the book while continuing to develop the next IBM Cloud Private Cloud Foundry release:

Chris Ahl
Subbarao Meduri
Dominique Vernier
IBM US

Kevin Cormier
Roke Jung
Colton Nicotera
Lindsay Martin
IBM Canada

Also thanks to the following SMEs from IBM US for contributing content to the Helm and application packaging section of the book:

Nicholas Schambureck
Ivory Knipfer
Chris Johnson
Joe Huizenga
Jon Huhn
Nancy Heinz
Rick Junkin

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 IBM Redbooks 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, IBM Redbooks
Dept. HYTD Mail Station P099
2455 South Road
Poughkeepsie, NY 12601-5400

Stay connected to IBM Redbooks

- ▶ Find us on Facebook:

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

- ▶ Follow us on Twitter:

<http://twitter.com/ibmredbooks>

- ▶ Look for us on LinkedIn:

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

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

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

- ▶ Stay current on recent Redbooks publications with RSS Feeds:

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



Highly available workloads and deployment on IBM Cloud Private

In this chapter, highly available workloads and deployments on IBM Cloud Private are described.

This chapter includes the following topics:

- ▶ 1.1, “Highly available workloads on IBM Cloud Private” on page 2
- ▶ 1.2, “High availability versus failover” on page 6
- ▶ 1.3, “Horizontal Pod Autoscaler for IBM Cloud Private” on page 8
- ▶ 1.4, “Deploy Spring Boot application on WebSphere Liberty Helm chart” on page 9
- ▶ 1.5, “Zero downtime deployment updates” on page 20
- ▶ 1.6, “Other deployment strategies implemented with native Kubernetes” on page 28
- ▶ 1.7, “Deploying a sample stateful application” on page 34

Note: If you are not familiar with IBM Cloud Private, we suggest that you first read the chapter “Introduction to IBM Cloud Private” in *IBM Private Cloud Systems Administrator's Guide*, SG2-48440.

If you want to follow the code examples that are used in this IBM Redbooks publication, you can download the GitHub repository of this book. For more information, see Appendix A, “Additional material” on page 139.

1.1 Highly available workloads on IBM Cloud Private

To understand how the Kubernetes orchestrator supports the high availability of workloads that run on IBM Cloud Private, developers must know a few key characteristics of Kubernetes. In an imperative configuration approach, developers define actions to modify the state of systems. Kubernetes uses a declarative configuration approach, where a developer defines the wanted state. Kubernetes then ensures that the state that you want becomes the actual state.

This capability of Kubernetes is the basis of its self-healing behaviors that keep applications running without interactions from developers. For example, Kubernetes can meet a service level response time by auto-scaling or allowing for failure and automatically redeploying pods.

You can think of an application that is deployed in a Kubernetes environment as an application that is made by one or more deployments. Example 1-1 shows a snippet of a deployment manifest `.yaml` file.

Example 1-1 Application deployment sample snippet

```
...
kind: Deployment
metadata:
  ...
  name: my_name
  namespace: default
spec:
  replicas: 2
  selector:
    ...
    restartPolicy: Always
```

In this manifest file, developer declares to Kubernetes that the application pod should be deployed with two replicas and that if a failure occurs, Kubernetes must restart a pod. Kubernetes deploys the pods and makes sure that two instances are always running. If a pod instance crashes, Kubernetes redeploys a new instance of the pod. The developer declares the status and Kubernetes does the rest. Typically, the restart of a pod happens quickly.

A developer might wonder: What about the data and storage? Who protects the data from a storage failure? Kubernetes can integrate several storage types, including the cloud storage, but the storage redundancy is assumed to be managed outside the Kubernetes cluster.

Note: For more information about storage persistence, see the chapter “Managing persistence in IBM Cloud Private” in *IBM Private Cloud Systems Administrator’s Guide*, SG24-8440.

Kubernetes uses defined persistent volumes (PVs) and binds them to persistent volume claims (PVCs) to assign storage to pods, even when the pod is moved to a new node. However, Kubernetes does not provide resiliency of the underlying storage resources. This task is performed by the selected storage provider.

Because PVs are Kubernetes resources, view them as any other Kubernetes resource; for example, a node. As Kubernetes decides on which node to deploy a pod, it also associates a PVC (the request of storage from a pod) to a PV that is based on the configurations that are defined.

The storage is mounted in the containers of the pod. When a pod is relocated or restarted, Kubernetes configures the correct mount points.

1.1.1 High-availability cluster models for workload deployment

The following classification models are used for highly available IBM Cloud Private clusters:

- ▶ Intra-cluster
- ▶ Intra-cluster with multiple availability zones
- ▶ Inter-cluster with federation on different availability zones

Intra-cluster

A cluster is composed of at least master nodes and worker nodes. This model consists of HA inside an IBM Cloud Private cluster. The redundancy is implemented by deploying multiple nodes for master and for workers that are distributed in a single site.

This scenario uses Kubernetes functions but cannot protect applications from site failure. If you need site-based protection, combine this model with other protection solutions, including a disaster recovery solution.

Figure 1-1 shows the intra-cluster topology for workload deployment.

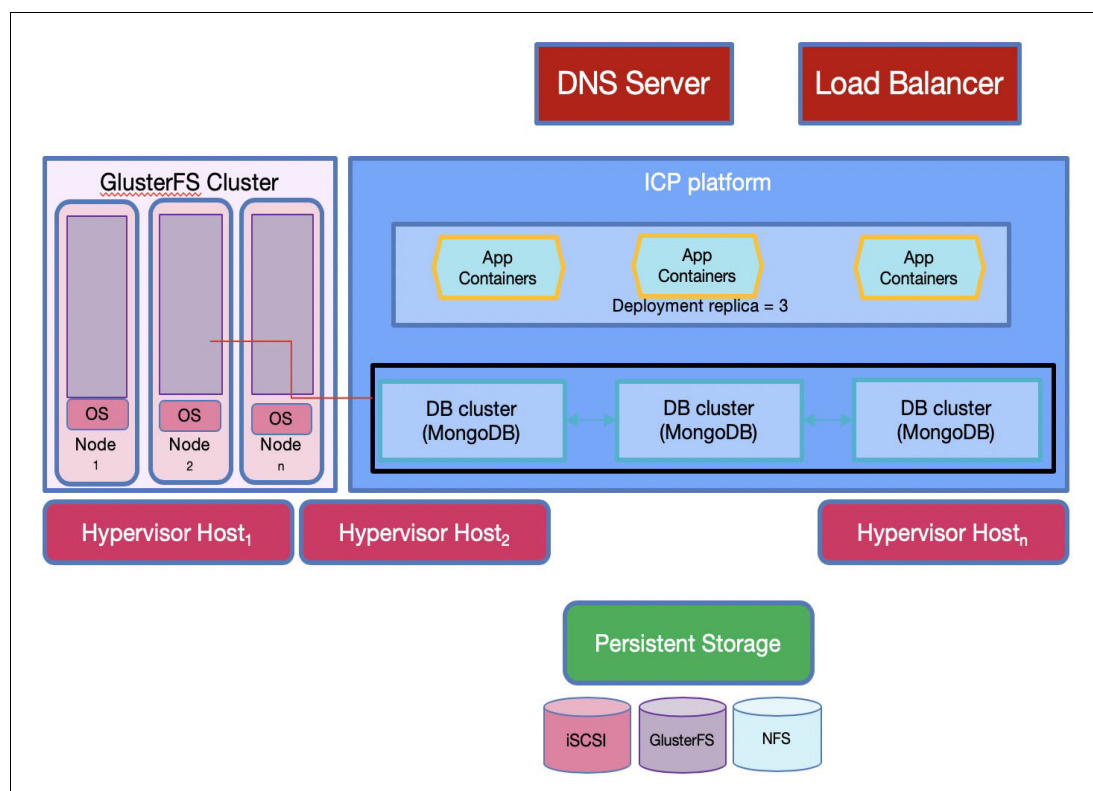


Figure 1-1 Intra-cluster topology

Figure 1-1 on page 3 also shows a highly available DB cluster that use three replicas of MongoDB. Persistence of storage is provided by the GlusterFS distributed filesystem. GlusterFS can use local disks in the storage cluster nodes (GlusterFS Cluster) or use disk volumes from external persistent storage (for example, sdisk array). Multiple storage classes and providers can coexist within one IBM Cloud Private cluster.

Intra-cluster with multiple availability zones

This kind of scenario is often referred to as “*business continuity*”. It combines intra-cluster HA with the capability to protect from a site failure.

The cluster is distributed among multiple zones. For example, you might have three, five, or seven master nodes and several worker nodes that are distributed among three zones. *Zones* are sites on the same campus or sites that are close to each other. If a complete zone fails, the master survives and can move the pods across the remaining worker nodes.

Note: This scenario is possible, but might present a few challenges. It must be implemented carefully.

Potential challenges in this form of workload deployment

This form of workload deployment includes the following potential challenges:

- ▶ The spread across multiple zones must not introduce latency. A high latency can compromise the overall Kubernetes work with unpredictable results. For example, because of latency, the master might consider a group of workers as unreachable and start to uselessly move pods. Or, one of the master nodes might be considered down only because of a long latency.
- ▶ In any failure condition, make sure that most of the master nodes survive. Distributing the cluster in only two zones is almost useless. However, configuring three zones implies more costs and complexity.

Figure 1-2 shows the intra-cluster with multiple availability zones topologies for workload deployment.

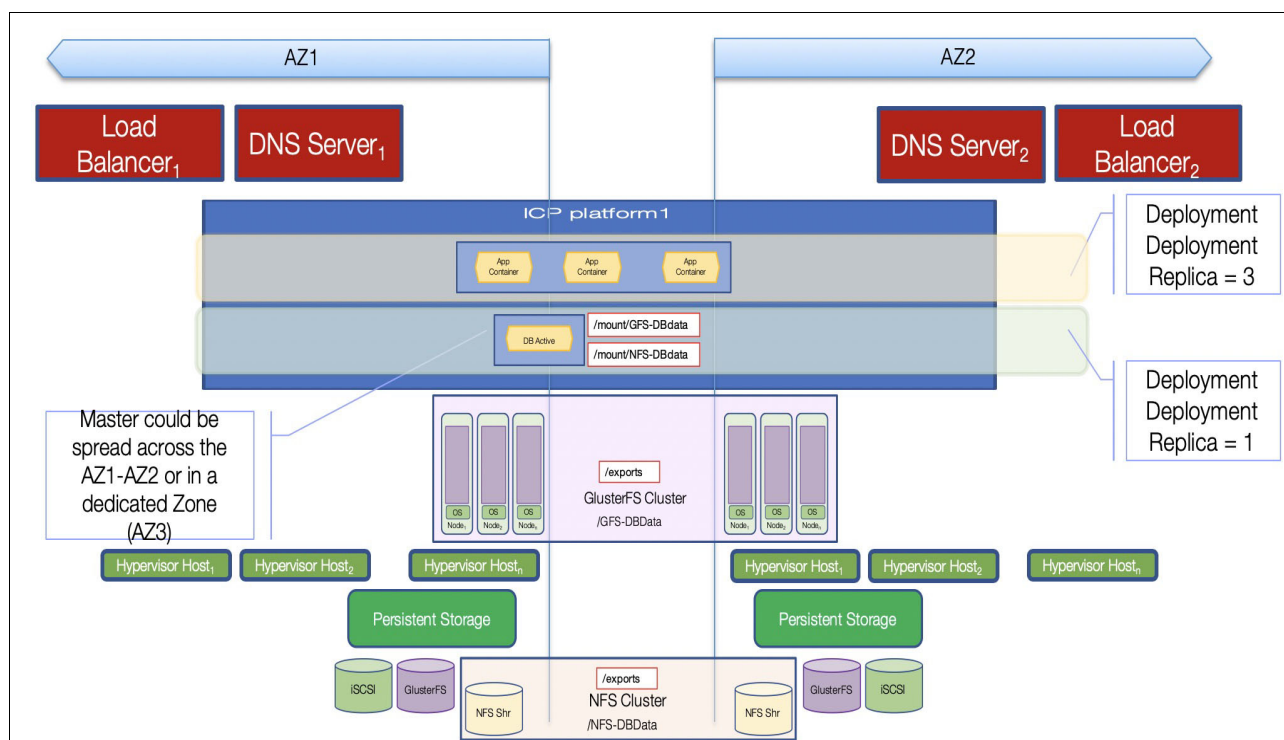


Figure 1-2 Intra-cluster with multiple availability zones topologies

Inter-cluster with federation on different availability zones

For this model, think of two Kubernetes clusters, each with its own master and worker nodes. If one cluster fails, its pods are run on the other cluster. Kubernetes supports this model by implementing a cluster federation. A higher-level master is deployed as a federated control plane. If a cluster fails, the master control plane instructs the masters of the surviving cluster to redeploy the failed pods.

The federation model is possible; however, beyond the orchestrator, you must consider all the other components of IBM Cloud Private to recover. For example, you must recover all the tools to manage the logs and to monitor your platform and workloads.

Although the federation for Kubernetes is a built-in feature, you still must take care of all the other components. As in the Intra-cluster with multiple zones model, you must also be aware of possible latency problems.

Support for federation is relatively new in Kubernetes. Before you apply federation to business-critical workloads, review its evolution and maturity.

Figure 1-3 shows the inter-cluster federation on different availability zones for workload deployment.

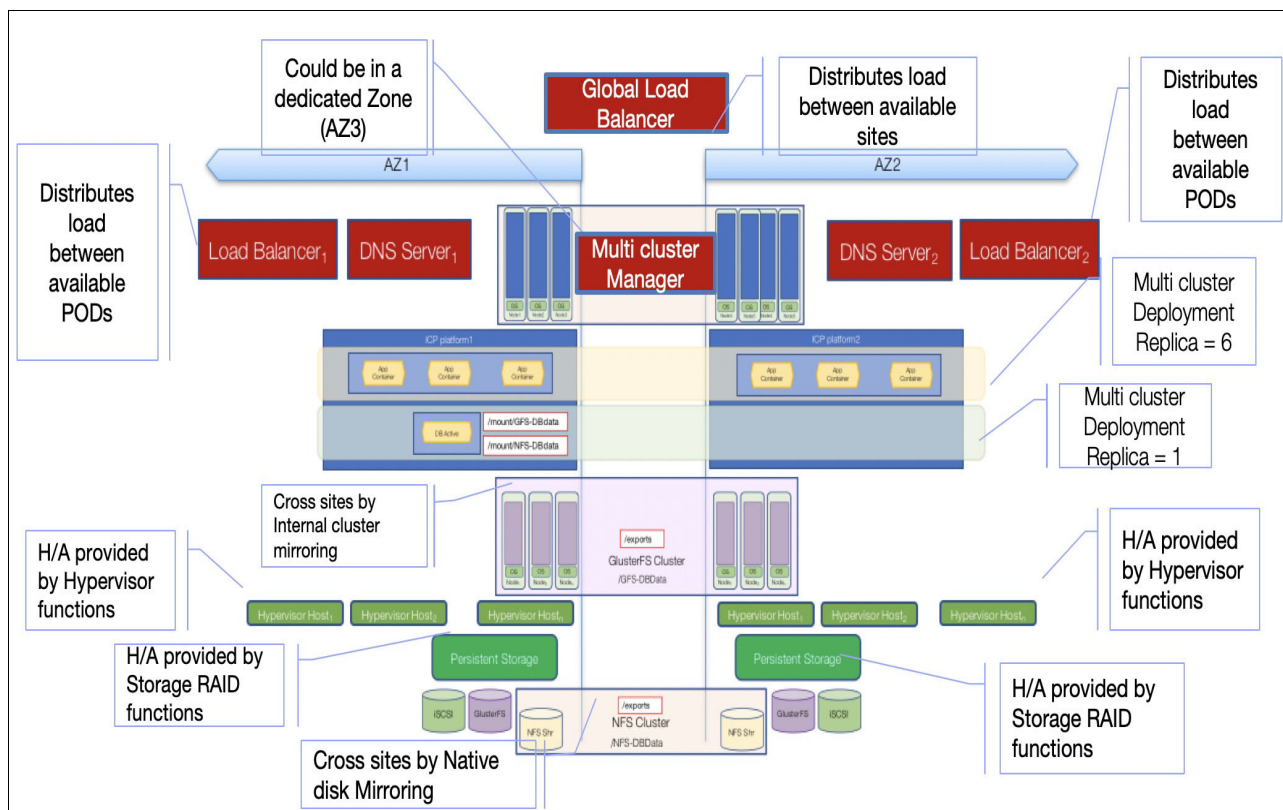


Figure 1-3 Inter-cluster with federation on different availability zone topology

1.2 High availability versus failover

At the application level, it is the pods that provide high availability in a Kubernetes environment. Kubernetes allows you to run multiple pods (redundancy) and if a pod fails, Kubernetes spins up a replacement pod to reflect the wanted state of the application.

Next, we analyze an example in which a developer runs an application that is composed of stateless front-end and stateful back-end pods.

A stateless Kubernetes pod is a pod that does not keep a state in the Kubernetes cluster. When a stateless pod is unhealthy, it is deleted and new pod is started.

Application state often is represented by the data in the database that is hosted on persistent storage on a file system that is outside of Kubernetes' control. Kubernetes provides the ability to define stateful pods, which maintain their identity across the restarts. If a stateful pod fails, it is restarted.

To implement the scenario that is shown on Figure 1-4 on page 7, the developer must define the following resources:

- ▶ A PV where the database data files are hosted (assume a relational database). The PV provides data persistency.
- ▶ The front-end pods that run as a stateless replicaset.

- ▶ A back-end database pod that runs as a statefulset.
- ▶ Services to expose the pods to the incoming traffic. A Kubernetes service allows the developer to select a mechanism for locating target pods in the network. For more information about the Kubernetes services, see [IBM Knowledge Center](#).

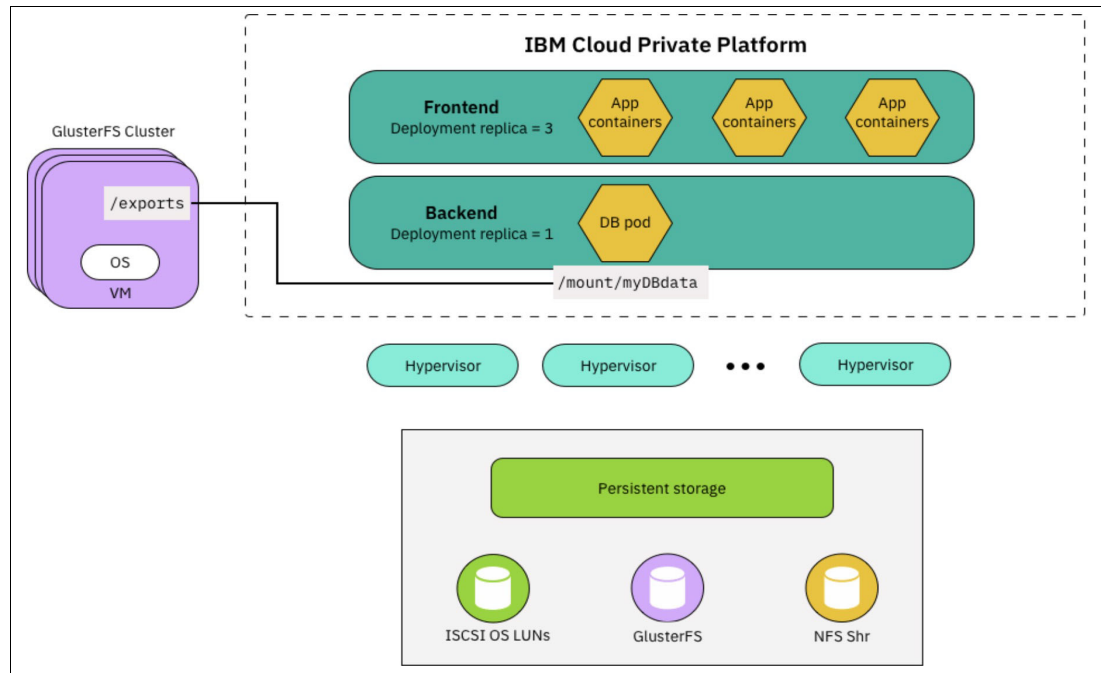


Figure 1-4 Stateless workload deployed in multiple pods in the same cluster

Figure 1-4 shows that the database pod uses a persistent volume claim, where the storage for the permanent volume is provided by GlusterFS. The data files are on that volume.

The high availability for this application is achieved by meeting the following requirements:

- ▶ The front-end deployment is defined as replica=3; therefore, whenever one pod crashes, two other instances are still available and Kubernetes automatically starts a new pod to meet the wanted quantity.
- ▶ The database application deployment is defined as replica=1; therefore, if that a pod fails, Kubernetes restarts it. If the worker node crashes, the pod is restarted on a new node.
- ▶ GlusterFS is self-redundant and you do not manage its high availability. The storage volume that you use is redundant by GlusterFS.
- ▶ In this example, the worker nodes are VMs that are made available by hypervisors. Most of the hypervisors provide an internal HA mechanism; however, Kubernetes redeployes the workloads. If a worker node crashes, you add a worker to restore the overall compute capacity.
- ▶ For the storage, you can also have another NFS or an internet Small Computer Systems Interface (iSCSI), most likely for the VMs. Either way, you can assume that the redundancy of the storage is provided by the storage systems.

In this scenario, the back-end component has a high availability limitation. Its reliability is equivalent to a traditional failover approach that is based on the high availability between two servers. In case of database pod restart (because of a failure or upgrade) the application experiences some downtime.

1.3 Horizontal Pod Autoscaler for IBM Cloud Private

The Horizontal Pod Autoscaler (HPA) automatically scales the number of pods in a replication controller, deployment, or replica set based on observed CPU utilization (or, with custom metrics support, on some other application-provided metrics).

Note: HPA does not apply to objects that cannot be scaled; for example, DaemonSets.

The HPA is implemented as a Kubernetes API resource and a controller. The resource determines the behavior of the controller. The controller periodically adjusts the number of replicas in a replication controller or deployment to match the observed average CPU utilization to the target specified by user.

The HPA in IBM Cloud Private allows the system to automatically scale workloads up or down based on the resource usage. This automatic scaling helps to guarantee service level agreements (SLAs) for your workloads.

By default, the HPA policy automatically scales the number of pods based on the observed CPU utilization. However, in many situations, you might want to scale the application based on other monitored metrics, such as the number of incoming requests or the memory consumption.

The basic principle is that you set the CPU requests (or other metrics) on the deployment and then, apply an autoscale component to it. This component then monitors the deployments resource consumption and if one of the specified metrics breaches the usage percentage threshold, it scales the number of pods (up to the specified maximum) until the load per pod is below the usage percentage again.

Kubernetes keeps monitoring the load and runs calculations on how many pods it needs, and increases or reduces the number to suit the load. Pods are autoscaled based on the first resource it finds to be in breach of the percentage threshold.

The command that is shown in Example 1-2 shows how to enable autoscaling on a deployment. This sets the CPU utilization threshold to 50% of CPU resources that were defined in the deployment, with a minimum of 1 and maximum of 10 pods to which to scale.

Example 1-2 *Kubectl command to configure horizontal pod autoscale*

```
kubectl autoscale deployment <name> --cpu-percent=50 --min=1 --max=10
```

To check the autoscaling for deployment, run the command as shown in Example 1-3.

Example 1-3 *List horizontal pod autoscaler*

```
kubectl get hpa
```

For more information about HPA, see the following resources:

- ▶ [Kubernetes website](#)
- ▶ [IBM Knowledge Center](#)

1.4 Deploy Spring Boot application on WebSphere Liberty Helm chart

This section covers how to deploy a sample “hello world” Spring Boot application with the WebSphere Liberty Helm chart running in IBM Cloud Private.

We examine different options of how the service is exposed through NodePort and Ingress.

1.4.1 Sample application

First, we create a Spring Boot application from scratch.

Create the project template

You can go to the <https://start.spring.io/> website to construct your Spring Boot project with the template that is provided in the website or run the command that is shown in Example 1-4 to automate it.

Example 1-4 Command to start a Spring Boot project

```
curl https://start.spring.io/starter.tgz -d baseDir=app -d name=demo -d
groupId=com.ibm.redbooksproject.example -d language=java -d type=maven-project
dependencie=web| tar -xzf -
```

The directory structure that is shown in Example 1-5 is then made available.

Example 1-5 File tree of initial project

```
app
... mvnw
... mvnw.cmd
... pom.xml
... src
...   main
...     ... java
...       ... com
...         ... ibm
...           ... redbooksproject
...             ... example
...               ... demo
...                 ... DemoApplication.java
...     ... resources
...     ... application.properties
...   test
...     ... java
...       ... com
...         ... ibm
...           ... redbooksproject
...             ... example
...               ... demo
...                 ... DemoApplicationTests.java
```

Notice that the pom.xml file is created. You can import this .xml file into your preferred IDE to continue to work on the project.

Adding hello REST service

Create a file named `HelloController.java` in the same folder of `DemoApplication.java`, as shown in Example 1-6.

Example 1-6 HelloController.java

```
package com.ibm.redbooksproject.example.demo;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HelloController {
    @RequestMapping("/hello")
    public Hello hello(@RequestParam(value="name", defaultValue="World") String
name) {
        return new Hello(name);
    }
}
```

This code takes the name HTTP parameter from the `/hello` url and constructs a `Hello` object, which is automatically delivered by Spring Boot as a JSON response.

Create the second file `Hello.java` in the same folder of `DemoApplication.java`, as shown in Example 1-7.

Example 1-7 Hello.java

```
package com.ibm.redbooksproject.example.demo;

import org.springframework.http.MediaType;
import org.springframework.web.HttpMediaTypeNotAcceptableException;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ResponseBody;

import java.util.Date;

public class Hello {
    public String name;
    public Date date;
    public String greetings;

    public Hello(String name) {
        this.name = name;
        this.date = new Date();
        this.greetings = String.format("Hello, %s!", name);
    }

    public String getName() {
        return name;
    }

    public Date getDate() {
        return date;
    }
}
```

```

    }

    public String getGreetings() {
        return greetings;
    }

    @ResponseBody
    @ExceptionHandler(HttpMediaTypeNotAcceptableException.class)
    public String handleHttpMediaTypeNotAcceptableException() {
        return "acceptable MIME type:" + MediaType.APPLICATION_JSON_VALUE;
    }
}

```

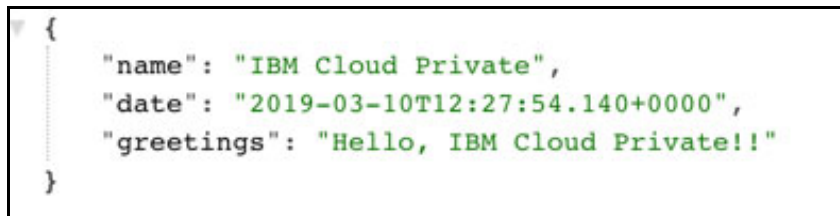
The Hello class constructs the fields, such as name, date, and greetings. The response body is automatically set as JSON. We have an exception handler in case other types of requests are present.

Test locally

Complete the following steps to test this configuration locally:

1. Run `cd app; maven clean install`. The use of this command creates the REST service JAR file that was created.
2. Run the service with `java -jar target/demo-0.0.1-SNAPSHOT.jar`.
3. Start your browser with `http://localhost:8080/hello?name=IBM%20Cloud%20Private`.

You should see output that is similar to the output that is shown in Figure 1-5.



```

{
  "name": "IBM Cloud Private",
  "date": "2019-03-10T12:27:54.140+0000",
  "greetings": "Hello, IBM Cloud Private!!"
}

```

Figure 1-5 Sample JSON output

Building the docker image

We host this app on the WebSphere Liberty container, which is running inside the IBM Cloud Private. First, we build the docker images and push them to the IBM Cloud Private private registry.

Use the Spring Boot tag of the Liberty image as the base and create the Dockerfile, as shown in Example 1-8.

Example 1-8 Dockerfile

```

FROM websphere-liberty:springBoot2
ADD app/target/demo-0.0.1-SNAPSHOT.jar /config/dropins/spring/demo.jar

```

Note: The default user of the Liberty image is a non-root user. If you need to create a directory that is owned by root, you must change the user to root, run your task, set the proper access for the normal user, and then, switch back to the default user.

It is a recommended best practice to run your container process as non-root. In fact, the default pod security policy in the current IBM Cloud Private release blocks the containers that run as root.

Build the Docker image by using the **docker build -t hello-liberty:v1** command.

Tip: The last character is period (.), which instructs docker to search the Dockerfile in the current directory.

Your Docker image is created locally on your development machine. Next, we push it to the IBM Cloud Private Registry.

For more information about how to set up your local docker client so that you can push the image, see [IBM Knowledge Center](#).

Pushing the docker image to IBM Cloud Private registry

Before pushing the image, make sure that the target namespace is created on the IBM Cloud Private cluster because the images must be in the specific namespace. In our example, we use exp as a namespace.

Run the commands that are shown in Example 1-9.

Example 1-9 Command to push the docker image

```
docker login <Your Cluster>:8500
docker tag hello-liberty:v1 <Your Cluster>:8500/exp/hello-liberty:v1
docker push <Your Cluster>:8500/exp/hello-liberty:v1
```

You should have your Docker image pushed into the IBM Cloud Private's private registry.

1.4.2 Deploying the application with the WebSphere Liberty Helm chart

To deploy our application, we use the command-line approach. In this section, we describe the high-level steps that are used to deploy the application through a Helm chart. Later, in “Deploying with NodePort service exposure” on page 13, and “Deploying with Ingress Transport Layer Security termination” on page 18, we complete these steps to specify different deployment options through the `values.yaml` overrides.

Setting up Helm repository for command line

Log in with `cloudctl`, which sets up the kubectl and Helm client for you. Then, run the command as shown in Example 1-10.

Example 1-10 Adding Helm repo

```
helm repo add ibm https://raw.githubusercontent.com/IBM/charts/master/repo/stable
helm repo update
```

A Helm repository is created that is named “ibm” that points to the online [IBM Helm charts repository](#).

Preparing override values.yaml file

To customize the variable values that are used in the Helm chart installation, you can create a .yaml file that contains a variable with values that you use to override the default values. For more information about the values.yaml file, see 2.2.3, “Helm values” on page 54.

Deploying the Helm chart

Run the command that is shown in Example 1-11 to deploy the Helm chart. The flag -f takes as an argument the file name with your override values for the Helm chart variables.

Example 1-11 Deploying the Helm chart

```
helm install --name {{ .releaseName }} --namespace {{ .namespace }} -f {{
.valueFile }} --tls ibm/ibm-websphere-liberty
```

Validating that the application is running

Run the following command to validate that the Helm chart was deployed successfully:

```
helm list --tls
```

Look for the releaseName that you provided during deployment. You can also access the REST services to validate that the application pod is working.

In the following sections, we use these steps to examine the two ways of service exposure, namely NodePort and Ingress.

1.4.3 Deploying with NodePort service exposure

In this deployment scenario, we use the NodePort to expose the application. All of the exposed interfaces are HTTPS-based. As a prerequisite, complete the steps that are described in “Deploying the application with the WebSphere Liberty Helm chart” on page 12 to build the Docker images.

Variable overrides

Prepare the .yaml file as shown in Example 1-12 and name it as vars.nodeport.yaml.

Example 1-12 Variables for NodePort service exposure

```
resourceNameOverride: redbooksproject
```

```
deployment:
  labels:
    environment: dev
    application: demo
    version: "1.0"
```

```
pod:
  labels:
    environment: dev
    application: demo
    version: "1.0"
```

```
service:
  enabled: true
  name: demo

  type: NodePort
  port: 9443
  targetPort: 9443
  labels:
    environment: dev
    application: demo
    version: "1.0"

image:
  repository: <Your ICP Host>:8500/exp/hello-liberty
  tag: vl@sha256:6aa245f6ad5c17766a2bccca3c0a71bcb07d56a68f9c41149269bc219bb61eecs

readinessProbe:
  httpGet:
    path: /
    port: 9080
  initialDelaySeconds: 40
  periodSeconds: 10

livenessProbe:
  httpGet:
    path: /
    port: 9080
  initialDelaySeconds: 40
  periodSeconds: 10

ssl:
  enabled: true
  useClusterSSLConfiguration: true
  createClusterSSLConfiguration: false

ingress:
  enabled: false

monitoring:
  enabled: true

resources:
  constraints:
    enabled: true
  requests:
    cpu: 500m
    memory: 512Mi
  limits:
    cpu: 500m
    memory: 512Mi
```

Recommended practices for the settings

The following recommended settings practices are used in this scenario:

- ▶ `resourceNameOverride`: Overwrite the default “ibm-websphere-liberty” to provide a more meaningful name.
- ▶ It is always a good practice to add labels on your deployment objects, such as deployment, pod, and services.
- ▶ Expose the services as NodePort. You can then access your application on a proxy node (or any other cluster node) by using automatically assigned port 3xxxx.
- ▶ Use the image that is pushed into the IBM Cloud Private’s private registry, as described in “Pushing the docker image to IBM Cloud Private registry” on page 12. If you use an external image registry, you must create a whitelist to allow the external images to run. For other private registries, you might also need to define the pull secret for them.
- ▶ Define your own readiness probe and liveness probe. Depending on your environment, these two probes might be different.
- ▶ Specify the resource constraints for this request carefully and use near real situation values so that the scheduler can select the correct nodes to run the pods and monitor and stop them when the application misbehaves.
- ▶ We turned off ingress for this scenario.
- ▶ SSL is set enabled. We provide the certificate and keystore and truststore instead of using the system self-created ones, which is the common practice in most of the environments. This is achieved by setting the option `useClusterSSLConfiguration` as true.

Keystore and truststore

In a production environment, it is common to get a certificate/key pair that is signed by the Enterprise Certificate Authority (CA).

Here, we use a self-signed certificate. You can use a tool, such as `openssl` or `cfssl`, to create and sign the certificate.

We do not provide the steps in this scenario to create and sign a self-signed certificate. It is assumed that you received the certificate and the key file that is stored in the Privacy Enhanced Mail (PEM) format that is named `demo.pem` and `demo-key.pem`. In this example, we have both Common Name (CN) and Subject Alternative Name (SAN) in the certificate set as `liberty.demo.redbooksproject.ibm`, as shown in Figure 1-6 on page 16.

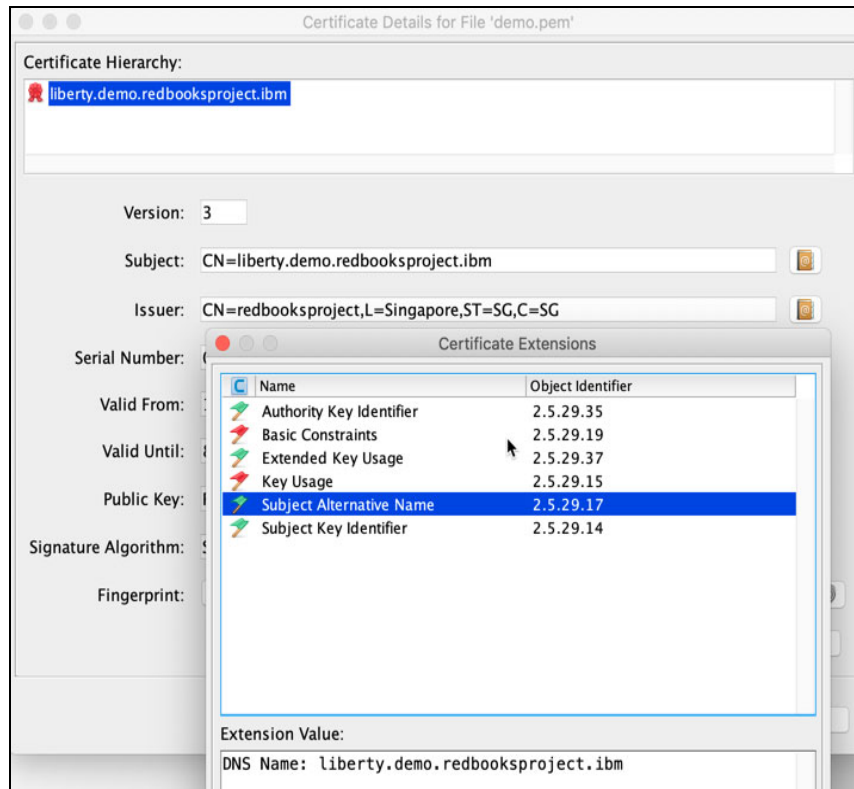


Figure 1-6 Certificate CN and SAN

Complete the following steps:

1. Use the pair of keys that we created for the keystore and truststore for Liberty. To avoid any format incompatibility issues, we use the IBM JDK that is inside the Liberty docker images to create the keystore and truststore.
2. Assuming your certificate files are stored in the keystore directory, run the Docker command that is shown in Example 1-13 to import the certificate and create the keystore and truststore.

Example 1-13 Docker exec command to generate keystore and truststore

```
docker run --rm -i -v $(pwd)/keystore:/keystore websphere-liberty sh -c 'cd
keystore && rm -rf key.p12 key.jks trust.jks && openssl pkcs12 -export -in
demo.pem -inkey demo-key.pem -out key.p12 -name default -passout pass:password &&
keytool -importkeystore -deststorepass password -destkeypass password
-destkeystore key.jks -srckeystore key.p12 -srcstoretype PKCS12 -srcstorepass
password -alias default && keytool -importcert -keystore trust.jks -storepass
password -file demo.pem -alias default -noprompt'
```

Now, your key.jks and trust.jks is created in the keystore/ directory.

3. Run the command that is shown in Example 1-14 to import the certificate files into the Kubernetes' namespace exp.

Example 1-14 Secret for keystore and truststore

```
kubectl -n exp create secret generic mb-keystore --from-file=keystore/key.jks
kubectl -n exp create secret generic mb-truststore --from-file=keystore/trust.jks
```



```
kubectl -n exp create secret generic mb-keystore-password
--from-literal=password=password
kubectl -n exp create secret generic mb-truststore-password
--from-literal=password=password
```

Note: The password can be encoded with XOR or AES format.

These secrets are expected for the Helm chart when you use your own keystore and truststore.

Deploying and validating the Helm chart

Complete the following steps to deploy the Helm chart:

1. Run the command that is shown in Example 1-15.

Example 1-15 Command to deploy a Helm chart

```
helm install --name demo --namespace exp -f vars.nodeport.yaml --tls
ibm/ibm-websphere-liberty
```

2. Run the **kubectl -n exp get pods** command to verify that the service is ready (the READY column goes to 1/1), as shown in Example 1-16.

Example 1-16 Status of Liberty Pod

NAME	READY	STATUS	RESTARTS	AGE
demo-redbooksproject-7f67bf94d-g2rdf	1/1	Running	0	96s

3. Determine the NodePort number by running the **kubectl -n exp get svc** command, as shown in Example 1-17.

Example 1-17 Identify NodePort

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)
demo	NodePort	172.21.179.28	<none>	9443:32057/TCP
demo-http-clusterip	ClusterIP	172.21.7.17	<none>	9080/TCP

4. In a browser, enter `https://<Proxy Node IP>:32057/hello?name=IBM Cloud Private`. You see the output that is shown in Figure 1-7.

```
{
  "name": "IBM Cloud Private",
  "date": "2019-03-10T12:27:54.140+0000",
  "greetings": "Hello, IBM Cloud Private!!"
}
```

Figure 1-7 Sample output with NodePort service exposure

The certificate is the one that we imported into the truststore (see Figure 1-8).

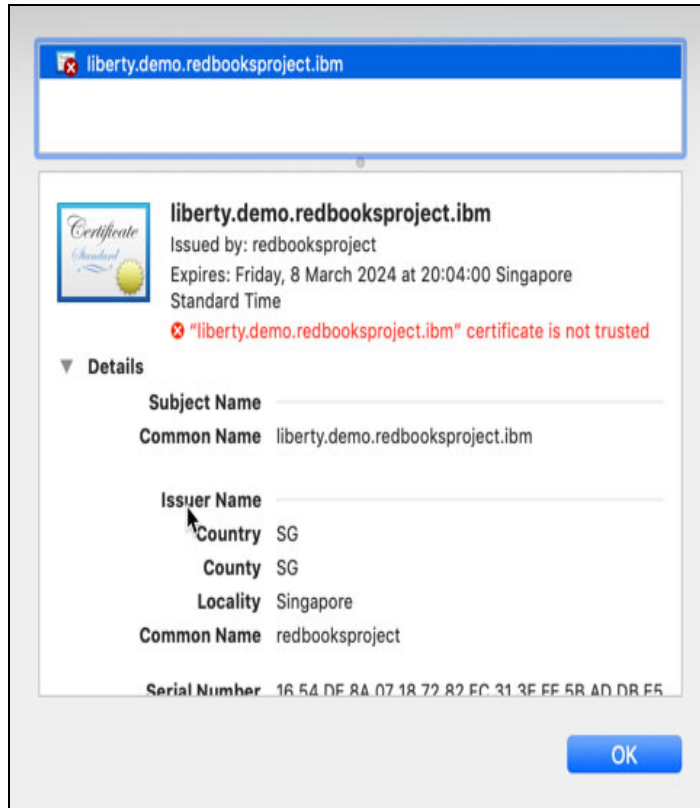


Figure 1-8 Certificate used for Nodeport

1.4.4 Deploying with Ingress Transport Layer Security termination

In this scenario, we expose the service not as the NodePort, but through Ingress with TLS termination instead. If the Host field in the request header matches what we defined, Ingress uses a predefined certificate to communicate with the client.

Variable overrides

Most of the settings are the same as the NodePort example, except those fields that are listed in Example 1-18.

Example 1-18 Variables for Ingress service exposure

```
service:
  enabled: true
  name: demo

  type: ClusterIP
  port: 9080
  targetPort: 9080
  labels:
    environment: dev
    application: demo
    version: "1.0"

ssl:
```

```
enabled: true
useClusterSSLConfiguration: true
createClusterSSLConfiguration: false
```

```
ingress:
  enabled: true
  host: "liberty.demo.redbooksproject.ibm"
  secretName: demo-ingress-tls
```

Recommended practices on the settings

We enabled Ingress. When the Host field matches `liberty.demo.redbooksproject.ibm`, the traffic is redirected to the Liberty pod that we created. Notice that the user can access the normal URL, such as `https://liberty.demo.redbooksproject.ibm`, by using the default port 443, instead of the 3xxx which is automatically assigned to the NodePod.

We create a TLS secret that is named `demo-ingress-tls` by running the command that is shown in Example 1-19.

Example 1-19 TLS secret

```
kubectl -n exp create secret tls demo-ingress-tls --key keystore/demo-key.pem
--cert keystore/demo.pem
```

Between the browser and Ingress, the traffic is encrypted with this TLS secret. After passing through Ingress, the traffic is not encrypted. Because the service that Ingress forwards the traffic to is HTTP, we define it by using the port 9080.

Note: So that Ingress is working properly, the Subject Alternative Name (SAN) must match the “host” value that is defined in Ingress.

In the current Helm chart version, SSL must be enabled to allow the Ingress TLS work.

Deploying and validating the Helm chart

Deploy the Helm chart. Assume that the DNS name of `liberty.demo.redbooksproject.ibm` in the enterprise is registered and resolves to the IBM Cloud Private proxy node. Here, we simulate it by updating the local `/etc/hosts` file (see Figure 1-9).



Figure 1-9 Browse through Ingress service exposure

Validate that the certificate is the certificate that we defined in the secret, as shown in Figure 1-10 on page 20.

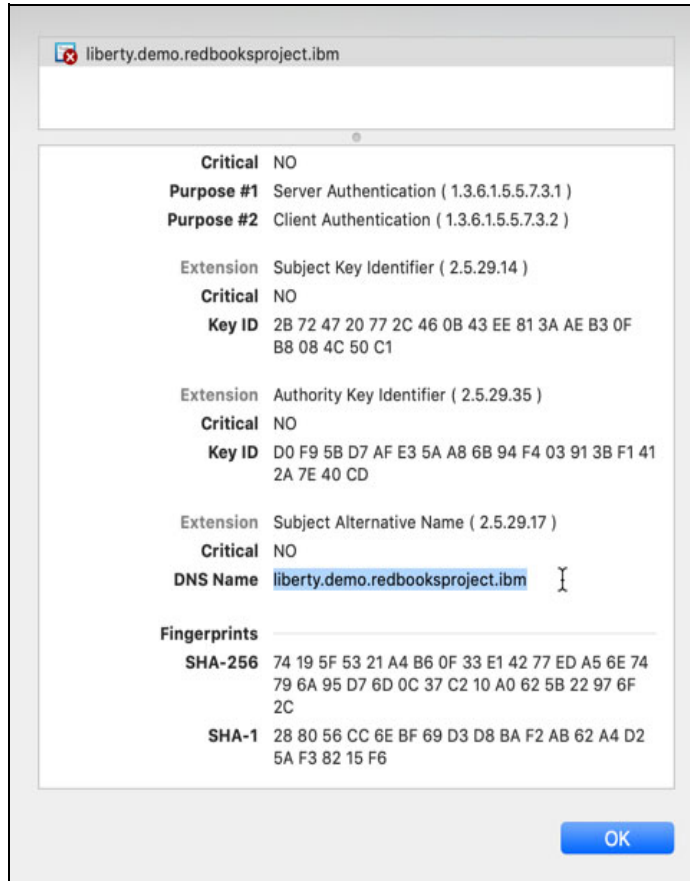


Figure 1-10 Certificate used for Ingress

1.5 Zero downtime deployment updates

It is a basic requirement to update the application; for example, to fix a bug or release a new feature. Kubernetes provides the zero downtime application updates by default.

Next, we demonstrate this feature by using an example.

1.5.1 Kubernetes deployment strategy

In a Kubernetes Deployment object definition, the default value is set as shown in Example 1-20 if no specific strategy is defined.

Example 1-20 Default update strategy

```
strategy:
  rollingUpdate:
    maxSurge: 25%
    maxUnavailable: 25%
  type: RollingUpdate
```

In the deployment definition (JSON file), if any field under the tree structure (for example `.spec.template`) is changed, a rolling update is triggered that follows the default rolling update strategy.

In the default setting that is shown in Example 1-20 on page 20, Kubernetes creates pods with the updates, such as the new image version and new environment variables. The number of new pods is limited to up to 25% of the total pods.

After the new pods are created, the liveness probe and the readiness probe help to detect if the newly created pod is ready. This strategy ensures that the availability of the overall pods is 75% percent. Accordingly, based on this value, old pods are stopped until all the pods are replaced.

Another type of strategy, named *Recreate*, is available. It deletes the old pods and creates pods where the application downtime is required.

We demonstrate the zero downtime approach in the next sections.

1.5.2 Sample application

In this section, we create a hello-world example with Golang and deploy it into IBM Cloud Private.

The source code is listed in Example 1-21.

Example 1-21 Sample Golang application

```
package main

import (
    "fmt"
    "log"
    "math/rand"
    "net/http"
    "os"
    "time"
)

var version string

func init() {
    version = "1.0"
}

func greet(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello World! from host:%s ver:%s", os.Getenv("HOSTNAME"),
    version)
}

func ready(w http.ResponseWriter, r *http.Request) {
    time.Sleep(time.Duration(rand.Intn(10)) * time.Second)
    fmt.Fprintf(w, "ok")
}

func main() {
    http.HandleFunc("/", greet)
```

```

http.HandleFunc("/healthz", ready)
http.HandleFunc("/readyz", ready)

port := os.Getenv("LISTEN_PORT")
if port == "" {
    port = "8080"
}
log.Printf("starting server on port:%s", port)
err := http.ListenAndServe(fmt.Sprintf(":%s", port), nil)
if err != nil {
    log.Fatalf("Failed to start the handler:%v", err)
}
}

```

Create two HTTP handlers. The `greet()` handler prints the “Hello world” greeting with the version information. The `read()` handler is used for Kubernetes readiness and the liveness probe.

Compile and build the binary. Create the Dockerfile, as shown in Example 1-22.

Example 1-22 Dockerfile

```

FROM alpine
RUN adduser -D app -u 1000
USER 1000
WORKDIR /home/app/hello
COPY --chown=app hello .

CMD ["/hello"]

```

We create a non-root user with `id 1000` to follow the recommended practices for image security. Build the image and push it to the image registry. (In this example, we use Docker hub with `zhiminwen` account).

1.5.3 Deploying the image to IBM Cloud Private

Complete the following steps to deploy the image to IBM Cloud Private:

1. Create a namespace that is called `deploy-demo` by using the **`kubectl create ns deploy-demo`** command.
2. Add an image whitelist policy for this namespace so that the `dockerhub` image can be used, as shown in Example 1-23.

Example 1-23 Image policy

```

apiVersion: securityenforcement.admission.cloud.ibm.com/v1beta1
kind: ImagePolicy
metadata:
  name: my-cluster-images-whitelist
  namespace: deploy-demo
spec:
  repositories:
    - name: docker.io/zhiminwen/*

```

3. Create a deployment file (as shown in Example 1-24) that is named `deploy.yaml`.

Example 1-24 Deployment file

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: demo
  labels:
    app: demo
spec:
  replicas: 10
  selector:
    matchLabels:
      app: demo
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 0
      maxSurge: 2
  template:
    metadata:
      labels:
        app: demo
    spec:
      containers:
      - name: demo
        image: zhiminwen/update-demo:v1.0
        env:
        - name: LISTEN_PORT
          value: "8080"
        livenessProbe:
          httpGet:
            path: /healthz
            port: 8080
        readinessProbe:
          httpGet:
            path: /readyz
            port: 8080
```

Note: Instead of using the default parameters of the rolling update, we set the `maxUnavailable` as 0 and `maxSurge` as 2.

4. Apply it to IBM Cloud Private by running the `kubectl -n deploy-demo apply -f deploy.yaml` command.
5. Create the service and Ingress objects to access the application, as shown in Example 1-25.

Example 1-25 Service and Ingress

```
---
apiVersion: v1
kind: Service
metadata:
  name: demo-svc
```

```

    labels:
      app: demo
spec:
  type: NodePort
  ports:
  - port: 80
    targetPort: 8080
    protocol: TCP
    name: http
  selector:
    app: demo
---
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  annotations:
    kubernetes.io/ingress.class: nginx
    ingress.kubernetes.io/rewrite-target: /
  labels:
    app: demo-svc
    name: demo-svc
spec:
  rules:
  - host: demo.52.116.21.122.nip.io
    http:
      paths:
      - path: /
        backend:
          serviceName: demo-svc
          servicePort: 80

```

6. Access the application from the browser or use the command that is shown in Example 1-26.

Example 1-26 Access the application

```

curl demo.52.116.21.122.nip.io
Hello World! from host:demo-58c7d5dc67-26zqk ver:1.0

```

1.5.4 Zero downtime update

Complete the following steps to test the zero downtime update:

1. Update the application with a version information change, as shown in Example 1-27.

Example 1-27 Version update

```

// skipped...
func init() {
    version = "1.1"
}

func greet(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello World! from host:%s ver:%s", os.Getenv("HOSTNAME"),
version)
}

```



```
// skipped...
```

2. Build the application and the Docker image. Deploy the image into the image registry with the tag of v1.1.
3. Update the pod image in the deployment by modifying the .yaml file from the IBM Cloud Private console to replace the new image tag, as shown in Figure 1-11.



Figure 1-11 Update image from console

4. You can edit your deployment .yaml file and apply it again. You also can run the following command to update the image:

```
kubectl -n deploy-demo set image deploy/demo demo=zhiminwen/update-demo:v1.1
```

1.5.5 Validation

After the image is updated, immediately run a series of commands (**kubectl -n deploy-demo get pods**) to list the pods. We do not use the watch option. Instead, we rerun the **kubectl** command so that the number of pods can be clearly seen.

You can see the output of the pods as shown in Example 1-28. Consider the following points:

- ▶ At any time, 10 pods are always in the READY state. We defined the maxUnavailable as 0, which means that Kubernetes must make sure that 10 out of 10 replicas are always ready.
- ▶ With the availability guaranteed, Kubernetes creates two pods each time and stops the old pod, which updates the pod version.
- ▶ The old pod versions are slowly replaced by the new pod versions, until all of the pods are running in the new version.

Example 1-28 Rolling update of pods

```
# kubectl -n deploy-demo get pods
NAME                                READY   STATUS    RESTARTS   AGE
```

```

demo-58c7d5dc67-26zqk 0/1 Running 0 4s
demo-58c7d5dc67-qbm85 0/1 ContainerCreating 0 0s
demo-58c7d5dc67-zjqmq 1/1 Running 0 4s
demo-59b9bfd4-4ktnl 1/1 Running 0 3m35s
demo-59b9bfd4-8h6kg 1/1 Running 0 3m35s
demo-59b9bfd4-gb5sc 1/1 Running 0 3m35s
demo-59b9bfd4-h5j82 1/1 Running 0 3m35s
demo-59b9bfd4-hr7wq 1/1 Running 0 3m35s
demo-59b9bfd4-kdrqx 1/1 Running 0 3m35s
demo-59b9bfd4-qhpsk 1/1 Running 0 3m35s
demo-59b9bfd4-swwtw 1/1 Running 0 3m35s
demo-59b9bfd4-tdcv5 1/1 Terminating 0 3m35s
demo-59b9bfd4-vfg4v 1/1 Running 0 3m35s
# kubectl -n deploy-demo get pods
NAME READY STATUS RESTARTS AGE
demo-58c7d5dc67-26zqk 0/1 Running 0 11s
demo-58c7d5dc67-qbm85 0/1 Running 0 7s
demo-58c7d5dc67-zjqmq 1/1 Running 0 11s
demo-59b9bfd4-4ktnl 1/1 Running 0 3m42s
demo-59b9bfd4-8h6kg 1/1 Running 0 3m42s
demo-59b9bfd4-gb5sc 1/1 Running 0 3m42s
demo-59b9bfd4-h5j82 1/1 Running 0 3m42s
demo-59b9bfd4-hr7wq 1/1 Running 0 3m42s
demo-59b9bfd4-kdrqx 1/1 Running 0 3m42s
demo-59b9bfd4-qhpsk 1/1 Running 0 3m42s
demo-59b9bfd4-swwtw 1/1 Running 0 3m42s
demo-59b9bfd4-vfg4v 1/1 Running 0 3m42s
# kubectl -n deploy-demo get pods
NAME READY STATUS RESTARTS AGE
demo-58c7d5dc67-26zqk 1/1 Running 0 17s
demo-58c7d5dc67-274ln 0/1 Running 0 4s
demo-58c7d5dc67-qbm85 1/1 Running 0 13s
demo-58c7d5dc67-xdjnt 0/1 Running 0 5s
demo-58c7d5dc67-zjqmq 1/1 Running 0 17s
demo-59b9bfd4-8h6kg 1/1 Running 0 3m48s
demo-59b9bfd4-gb5sc 1/1 Running 0 3m48s
demo-59b9bfd4-h5j82 1/1 Running 0 3m48s
demo-59b9bfd4-hr7wq 1/1 Running 0 3m48s
demo-59b9bfd4-kdrqx 0/1 Terminating 0 3m48s
demo-59b9bfd4-qhpsk 1/1 Running 0 3m48s
demo-59b9bfd4-swwtw 1/1 Running 0 3m48s
demo-59b9bfd4-vfg4v 1/1 Running 0 3m48s
# kubectl -n deploy-demo get pods
NAME READY STATUS RESTARTS AGE
demo-58c7d5dc67-26zqk 1/1 Running 0 22s
demo-58c7d5dc67-274ln 1/1 Running 0 9s
demo-58c7d5dc67-ntx4w 0/1 Running 0 3s
demo-58c7d5dc67-qbm85 1/1 Running 0 18s
demo-58c7d5dc67-snpnz 0/1 Running 0 3s
demo-58c7d5dc67-xdjnt 1/1 Running 0 10s
demo-58c7d5dc67-zjqmq 1/1 Running 0 22s
demo-59b9bfd4-8h6kg 1/1 Running 0 3m53s
demo-59b9bfd4-gb5sc 0/1 Terminating 0 3m53s
demo-59b9bfd4-h5j82 1/1 Running 0 3m53s
demo-59b9bfd4-hr7wq 1/1 Running 0 3m53s

```

```

demo-59b9bffd4-qhpsk    1/1    Running    0        3m53s
demo-59b9bffd4-swwtw    0/1    Terminating    0        3m53s
demo-59b9bffd4-vfg4v    1/1    Running    0        3m53s
# kubectl -n deploy-demo get pods
NAME                    READY   STATUS    RESTARTS   AGE
demo-58c7d5dc67-26zqk  1/1    Running    0          29s
demo-58c7d5dc67-274ln  1/1    Running    0          16s
demo-58c7d5dc67-82757  0/1    Running    0           2s
demo-58c7d5dc67-ntx4w  0/1    Running    0          10s
demo-58c7d5dc67-qbm85  1/1    Running    0          25s
demo-58c7d5dc67-snpnz  1/1    Running    0          10s
demo-58c7d5dc67-xdjnt  1/1    Running    0          17s
demo-58c7d5dc67-zjqmq  1/1    Running    0          29s
demo-59b9bffd4-8h6kg   1/1    Running    0           4m
demo-59b9bffd4-gb5sc    0/1    Terminating    0           4m
demo-59b9bffd4-h5j82    0/1    Terminating    0           4m
demo-59b9bffd4-hr7wq    1/1    Running    0           4m
demo-59b9bffd4-qhpsk    1/1    Running    0           4m
demo-59b9bffd4-swwtw    0/1    Terminating    0           4m
demo-59b9bffd4-vfg4v    1/1    Running    0           4m
# kubectl -n deploy-demo get pods
NAME                    READY   STATUS    RESTARTS   AGE
demo-58c7d5dc67-26zqk  1/1    Running    0          37s
demo-58c7d5dc67-274ln  1/1    Running    0          24s
demo-58c7d5dc67-82757  1/1    Running    0          10s
demo-58c7d5dc67-m6l56  0/1    Running    0           6s
demo-58c7d5dc67-ntx4w  1/1    Running    0          18s
demo-58c7d5dc67-qbm85  1/1    Running    0          33s
demo-58c7d5dc67-qn9p6  1/1    Running    0           4s
demo-58c7d5dc67-snpnz  1/1    Running    0          18s
demo-58c7d5dc67-xdjnt  1/1    Running    0          25s
demo-58c7d5dc67-zjqmq  1/1    Running    0          37s
demo-59b9bffd4-8h6kg   0/1    Terminating    0          4m8s
demo-59b9bffd4-h5j82    0/1    Terminating    0          4m8s
demo-59b9bffd4-hr7wq    1/1    Running    0          4m8s
demo-59b9bffd4-vfg4v    0/1    Terminating    0          4m8s
# kubectl -n deploy-demo get pods
NAME                    READY   STATUS    RESTARTS   AGE
demo-58c7d5dc67-26zqk  1/1    Running    0          43s
demo-58c7d5dc67-274ln  1/1    Running    0          30s
demo-58c7d5dc67-82757  1/1    Running    0          16s
demo-58c7d5dc67-m6l56  1/1    Running    0          12s
demo-58c7d5dc67-ntx4w  1/1    Running    0          24s
demo-58c7d5dc67-qbm85  1/1    Running    0          39s
demo-58c7d5dc67-qn9p6  1/1    Running    0          10s
demo-58c7d5dc67-snpnz  1/1    Running    0          24s
demo-58c7d5dc67-xdjnt  1/1    Running    0          31s
demo-58c7d5dc67-zjqmq  1/1    Running    0          43s
demo-59b9bffd4-hr7wq    0/1    Terminating    0          4m14s
# kubectl -n deploy-demo get pods
NAME                    READY   STATUS    RESTARTS   AGE
demo-58c7d5dc67-26zqk  1/1    Running    0          45s
demo-58c7d5dc67-274ln  1/1    Running    0          32s
demo-58c7d5dc67-82757  1/1    Running    0          18s
demo-58c7d5dc67-m6l56  1/1    Running    0          14s

```

demo-58c7d5dc67-ntx4w	1/1	Running	0	26s
demo-58c7d5dc67-qbm85	1/1	Running	0	41s
demo-58c7d5dc67-qn9p6	1/1	Running	0	12s
demo-58c7d5dc67-snpnz	1/1	Running	0	26s
demo-58c7d5dc67-xdjnt	1/1	Running	0	33s
demo-58c7d5dc67-zjqmq	1/1	Running	0	45s

1.6 Other deployment strategies implemented with native Kubernetes

In the previous section, we described the Recreate/Rolling update strategy for the deployment. Now, we examine some commonly available deployment options that can be implemented with the native Kubernetes objects. We continue to use the sample application as was used in the previous section, where two versions of the application are simulated with the Docker image tags of v1.0 and v1.1.

1.6.1 Blue/green deployment

In a blue/green deployment, a fresh new application deployment (green) is created along with the existing version (blue). After the testing is done, you can configure the load balancer to switch the traffic to the new application.

In this section, we build the blue/green deployment example with the native Kubernetes techniques.

Blue deployment

We assume the hello world application is running with the image tag of v1.0. It is served as the blue deployment. Nothing needs to be changed before switching Ingress to the green deployment.

Testing the green deployment

Complete the following steps to test the green deployment:

1. Create a Kubernetes deployment object with the `.yaml` file that is named `green.yaml`, as shown in Example 1-29.

Example 1-29 Green deployment

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: demo-green
  labels:
    app: demo-green
spec:
  replicas: 10
  selector:
    matchLabels:
      app: demo-green
  template:
    metadata:
      labels:

```

```

    app: demo-green
spec:
  containers:
  - name: demo-green
    image: zhiminwen/update-demo:v1.1
    env:
    - name: LISTEN_PORT
      value: "8080"
  livenessProbe:
    httpGet:
      path: /healthz
      port: 8080
  readinessProbe:
    httpGet:
      path: /readyz
      port: 8080

```

Pay attention to the changes by comparing the output with the blue deployment. Other than the different image version we assigned, we also use a different name and label for the green deployment.

2. Deploy the app by using the **kubectl -n deploy-demo apply -f green.yaml** command.
3. Check the pods with their labels, as shown in Example 1-30.

Example 1-30 Pods with the label

kubectl -n deploy-demo get pods -L app

NAME	READY	STATUS	RESTARTS	AGE	APP
demo-58c7d5dc67-26zqk	1/1	Running	0	179m	demo
demo-58c7d5dc67-274ln	1/1	Running	0	178m	demo
demo-58c7d5dc67-82757	1/1	Running	0	178m	demo
demo-58c7d5dc67-m6156	1/1	Running	0	178m	demo
demo-58c7d5dc67-ntx4w	1/1	Running	0	178m	demo
demo-58c7d5dc67-qbm85	1/1	Running	0	179m	demo
demo-58c7d5dc67-qn9p6	1/1	Running	0	178m	demo
demo-58c7d5dc67-spn pz	1/1	Running	0	178m	demo
demo-58c7d5dc67-xdjnt	1/1	Running	0	178m	demo
demo-58c7d5dc67-zjqmq	1/1	Running	0	179m	demo
demo-green-6f8869759f-4jnz7	1/1	Running	0	4m13s	demo-green
demo-green-6f8869759f-4p98r	1/1	Running	0	4m13s	demo-green
demo-green-6f8869759f-7xc2t	1/1	Running	0	4m13s	demo-green
demo-green-6f8869759f-b72lg	1/1	Running	0	4m13s	demo-green
demo-green-6f8869759f-dctgt	1/1	Running	0	4m13s	demo-green
demo-green-6f8869759f-l7m5d	1/1	Running	0	4m13s	demo-green
demo-green-6f8869759f-l96xd	1/1	Running	0	4m13s	demo-green
demo-green-6f8869759f-lnmv8	1/1	Running	0	4m13s	demo-green
demo-green-6f8869759f-nkvr9	1/1	Running	0	4m13s	demo-green
demo-green-6f8869759f-wz5m5	1/1	Running	0	4m13s	demo-green

4. Now Expose the green service with NodePort for testing purposes only. Create the file `greenService.yaml` with the content, as shown in Example 1-31.

Example 1-31 Expose the green service as NodePort

```
apiVersion: v1
kind: Service
metadata:
  name: demo-green-svc
  labels:
    app: demo-green
spec:
  type: NodePort
  ports:
  - port: 80
    targetPort: 8080
    protocol: TCP
    name: http
  selector:
    app: demo-green
```

5. Test the application as shown in Example 1-32. As expected, we see that the version is shown as 1.1.

Example 1-32 Validate the green app's version

```
curl http://52.116.21.122:31089;echo
```

```
Hello World! from host:demo-green-6f8869759f-lnmv8 ver:1.1
```

Switching to the green deployment

Before the switch, ensure that Ingress still points to the blue deployment. Complete the following steps:

1. Verify that the blue app's version is 1.0, as shown in Example 1-33. The output shows the version is still 1.0.

Example 1-33 Verifying the blue app's version

```
curl demo.52.116.21.122.nip.io;echo
```

```
Hello World! from host:demo-59b9bffd4-xk68w ver:1.0
```

2. Prepare a new file for Ingress. Keep all of the existing settings, including the labels, name, and ingress rules the same as in the blue deployment. Only update the serviceName to the green deployment, `demo-green-svc`, as shown in Example 1-34.

Example 1-34 Update ingress for the new green service

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  annotations:
    kubernetes.io/ingress.class: nginx
    ingress.kubernetes.io/rewrite-target: /
  labels:
    app: demo
    name: demo-svc
```

```
spec:
  rules:
  - host: demo.52.116.21.122.nip.io
    http:
      paths:
      - path: /
        backend:
          serviceName: demo-green-svc
          servicePort: 80
```

3. Apply the objects by running `kubectl -n deploy-demo apply -f green_ingress.yaml`.
4. Check that the application is upgraded to v1.1 through Ingress, as shown in Example 1-35.

Example 1-35 Ingress updated to the new app

```
curl demo.52.116.21.122.nip.io;echo
Hello World! from host:demo-green-6f8869759f-b721g ver:1.1
```

Now the blue deployment can be retired.

1.6.2 Canary deployment

Canary deployment allows a small amount of traffic to be routed to the new version of the application for testing purposes. Assuming we want 20% of the requests served by our new version 1.1, we can achieve this type of canary deployment by using native Kubernetes.

Complete the following steps:

1. Create a deployment with the Kubernetes deployment object, as shown in the .yaml file in Example 1-36.

Example 1-36 New deployment Kubernetes object

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: demo-canary
  labels:
    app: demo
    version: v1.1
spec:
  replicas: 2
  selector:
    matchLabels:
      app: demo
  template:
    metadata:
      labels:
        app: demo
    spec:
      containers:
      - name: demo
        image: zhiminwen/update-demo:v1.1
        env:
        - name: LISTEN_PORT
```

```

        value: "8080"
    livenessProbe:
      httpGet:
        path: /healthz
        port: 8080
    readinessProbe:
      httpGet:
        path: /readyz
        port: 8080

```

This new deployment is different than the existing deployment but shares the selection label for the pod. We set the image version as v1.1 and the replicas as 2.

2. Apply the .yaml file. You should see two deployments now (see Example 1-37).

Example 1-37 List of deployment

```
kubectl -n deploy-demo get deploy
```

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
demo	8	8	8	8	12m
demo-canary	2	2	2	2	50s

3. If we check the pods based on the label “app”, we see the output that is shown in Example 1-38.

Example 1-38 List of pods with the same label.

```
kubectl -n deploy-demo get pods -l app=demo -L app
```

NAME	READY	STATUS	RESTARTS	AGE	APP
demo-59b9bffd4-9jx8n	1/1	Running	0	14m	demo
demo-59b9bffd4-bc5pd	1/1	Running	0	14m	demo
demo-59b9bffd4-h5kxs	1/1	Running	0	14m	demo
demo-59b9bffd4-htgb6	1/1	Running	0	14m	demo
demo-59b9bffd4-kspr4	1/1	Running	0	14m	demo
demo-59b9bffd4-rtz5x	1/1	Running	0	14m	demo
demo-59b9bffd4-s2pzv	1/1	Running	0	14m	demo
demo-59b9bffd4-tsnkx	1/1	Running	0	14m	demo
demo-59b9bffd4-v56tf	1/1	Running	0	14m	demo
demo-59b9bffd4-vfvrx	1/1	Running	0	14m	demo
demo-canary-58c7d5dc67-8gwmj	1/1	Running	0	76s	demo
demo-canary-58c7d5dc67-jxp7v	1/1	Running	0	76s	demo

As defined in the service exposure, any pod that has the label app=demo is part of the service that is serving the request. Therefore, we can bring the canary application to be part of the service by giving the same label to the pods. We set the difference by giving a version label to the new deployment.

4. To have the number of percentages easily matched, we scale down the old deployment down to 8 pods to ensure that 20% of the pods are used for the canary application, and 80% are used the existing applications. The command to run is **kubectl -n deploy-demo scale --replicas=8 deploy demo**.

5. List the pods again (see Example 1-39).

Example 1-39 Count of the app

```
kubectl -n deploy-demo get pods -l app=demo -L app
```

NAME	READY	STATUS	RESTARTS	AGE	APP
demo-59b9bffd4-9jx8n	1/1	Running	0	15m	demo
demo-59b9bffd4-h5kxs	1/1	Running	0	15m	demo
demo-59b9bffd4-htgb6	1/1	Running	0	15m	demo
demo-59b9bffd4-kspr4	1/1	Running	0	15m	demo
demo-59b9bffd4-rtz5x	1/1	Running	0	15m	demo
demo-59b9bffd4-s2pzv	1/1	Running	0	15m	demo
demo-59b9bffd4-tsnkx	1/1	Running	0	15m	demo
demo-59b9bffd4-v56tf	1/1	Running	0	15m	demo
demo-canary-58c7d5dc67-8gwmj	1/1	Running	0	2m11s	demo
demo-canary-58c7d5dc67-jxp7v	1/1	Running	0	2m11s	demo

As Ingress distributes the traffic evenly among the pod, we can have 20% (2 pods) running the new app and 80% (8 pods) running with the existing version. Therefore, only 20% of the traffic goes to the new app of version 1.1, which achieves the canary deployment.

6. Validate the canary deployment by triggering the request constantly, as shown in Example 1-40.

Example 1-40 Validating the canary deployment.

```
while true; do curl http://demo.52.116.21.122.nip.io ; echo; sleep 1; done
Hello World! from host:demo-59b9bffd4-v56tf ver:1.0
Hello World! from host:demo-59b9bffd4-s2pzv ver:1.0
Hello World! from host:demo-canary-58c7d5dc67-jxp7v ver:1.1
Hello World! from host:demo-59b9bffd4-rtz5x ver:1.0
Hello World! from host:demo-59b9bffd4-tsnkx ver:1.0
Hello World! from host:demo-canary-58c7d5dc67-8gwmj ver:1.1
Hello World! from host:demo-59b9bffd4-kspr4 ver:1.0
Hello World! from host:demo-59b9bffd4-9jx8n ver:1.0
Hello World! from host:demo-59b9bffd4-h5kxs ver:1.0
Hello World! from host:demo-59b9bffd4-htgb6 ver:1.0

Hello World! from host:demo-59b9bffd4-v56tf ver:1.0
Hello World! from host:demo-59b9bffd4-s2pzv ver:1.0
Hello World! from host:demo-canary-58c7d5dc67-jxp7v ver:1.1
Hello World! from host:demo-59b9bffd4-rtz5x ver:1.0
Hello World! from host:demo-59b9bffd4-tsnkx ver:1.0
Hello World! from host:demo-canary-58c7d5dc67-8gwmj ver:1.1
Hello World! from host:demo-59b9bffd4-kspr4 ver:1.0
Hello World! from host:demo-59b9bffd4-9jx8n ver:1.0
Hello World! from host:demo-59b9bffd4-h5kxs ver:1.0
Hello World! from host:demo-59b9bffd4-htgb6 ver:1.0
```

As you can see in the output, 20% of the requests are handled by the new version, version 1.1. Therefore, we achieved the canary deployment.

1.7 Deploying a sample stateful application

In this section, you learn how to deploy a sample stateful application on IBM Cloud Private. The sample application that is demonstrated is a Nginx web server. You store the page content on an NFS server outside IBM Cloud Private.

Note: You store the page content on an NFS server that is outside of IBM Cloud Private. Similar steps can be followed if your stateful application depends on GlusterFS or Ceph.

We perform the following high-level steps in this section:

1. Set up an NFS server in case you do not have an existing NFS server.
2. Create a persistence volume and a persistence volume claim on IBM Cloud Private to connect to the NFS server.
3. Create a Nginx application from the IBM Cloud Private catalog and bind it to the persistence volume claim.
4. Test the application.

1.7.1 Prerequisite: Setting up an NFS server

This section describes the basic steps that are used to create an NFS server on top of the virtual server instances on IBM Cloud. Skip this section if you have an NFS server. Alternatively, you can create a File Storage service on IBM Cloud to achieve the same results, or use one of your IBM Cloud Private nodes and create a File Storage service on top of an NFS server.

Complete the following steps:

1. Create a virtual server instance in IBM Cloud. You use this instance to install your File Server on top of it. Complete the following steps:
 - a. Browse to <https://cloud.ibm.com/> and log in as shown in Figure 1-12.

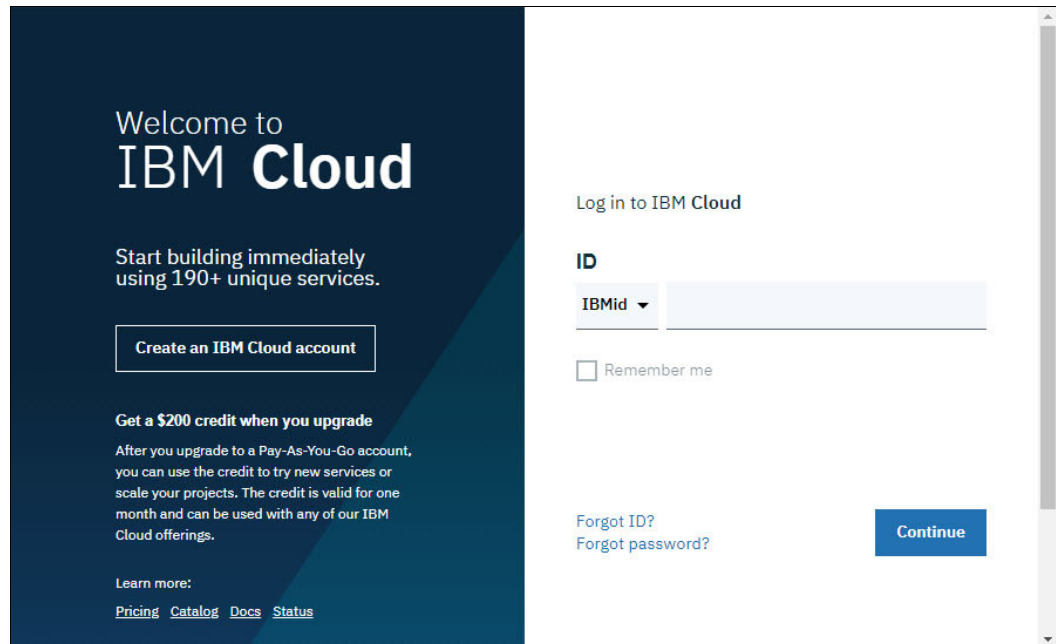


Figure 1-12 IBM Cloud console login page

- b. Click **Catalog** from the tool bar.
- c. Select **Virtual Server** from the catalog.
- d. Select **Public Virtual Server** then, click **Continue**.
- e. Keep all defaults, and choose **Ubuntu** in the Image Panel, as shown in Figure 1-13.

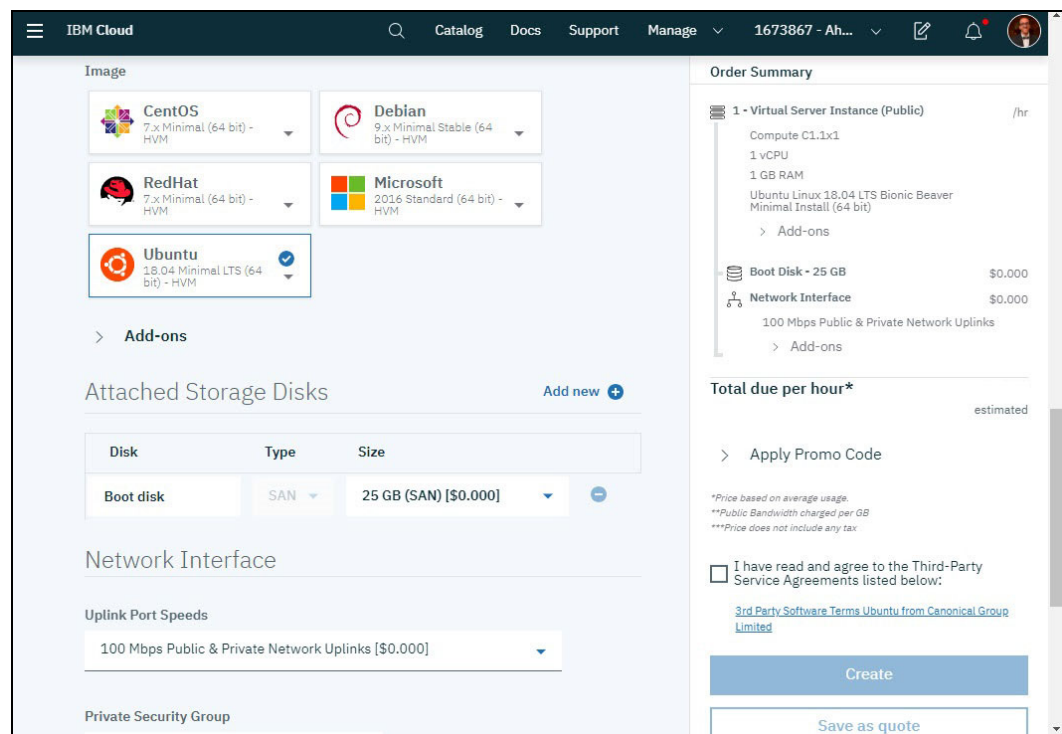


Figure 1-13 Creating IBM Cloud Virtual Server instance

- f. Read the terms and conditions. Then, click **I have read and agree to the Third-Party Service Agreements listed below**. Click **Create**.
 - g. Wait until the virtual server is provisioned.
2. Log in to the server by using SSH or PuTTY.
3. Run the script that is shown in Example 1-41 as a root user on your virtual server to create the NFS server. This a sample script is used for testing purposes and is not intended for production use.

Example 1-41 Creating NFS server script on virtual machine

```
apt update
apt install nfs-kernel-server -y
mkdir /redbooks
chown nobody:nogroup /redbooks
echo /redbooks *(rw,sync,no_root_squash,no_subtree_check) > /etc/exports
systemctl restart nfs-kernel-server
```

4. Create an `index.html` file in the `/redbooks` directory, as shown in the following example. This file is served from the Nginx application that you deploy later on IBM Cloud Private:

```
echo Redbooks Welcome Message Stored in NFS > /redbooks/index.html
```

1.7.2 Creating a persistence volume and persistence volume claim

Complete the following steps to create a persistence volume and persistence volume claim on IBM Cloud Provider to persist on the NFS server:

1. Browse to IBM Cloud Private console with your web browser and log in to it.
2. Click **Menu** → **Platform** → **Storage**, as shown in Figure 1-14.

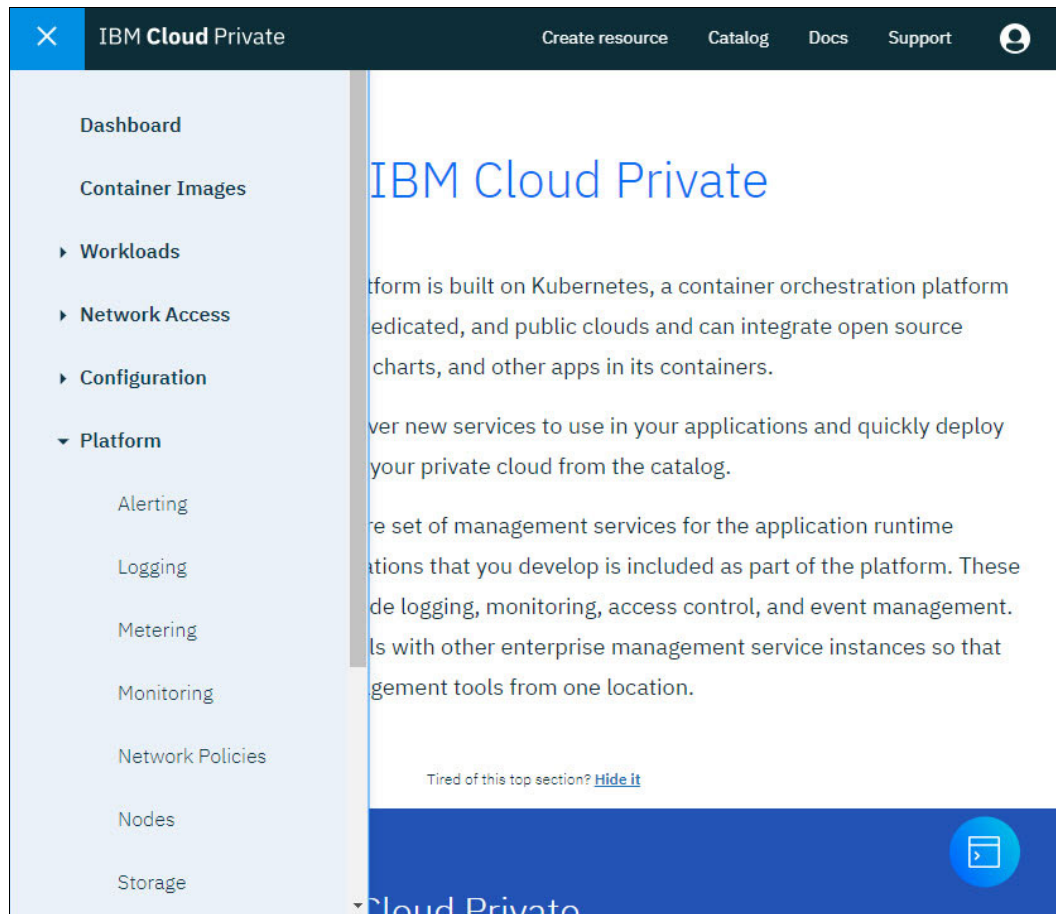


Figure 1-14 Menu

3. Click **Create PersistentVolume** to create a persistence volume, as shown in Figure 1-15 on page 37.

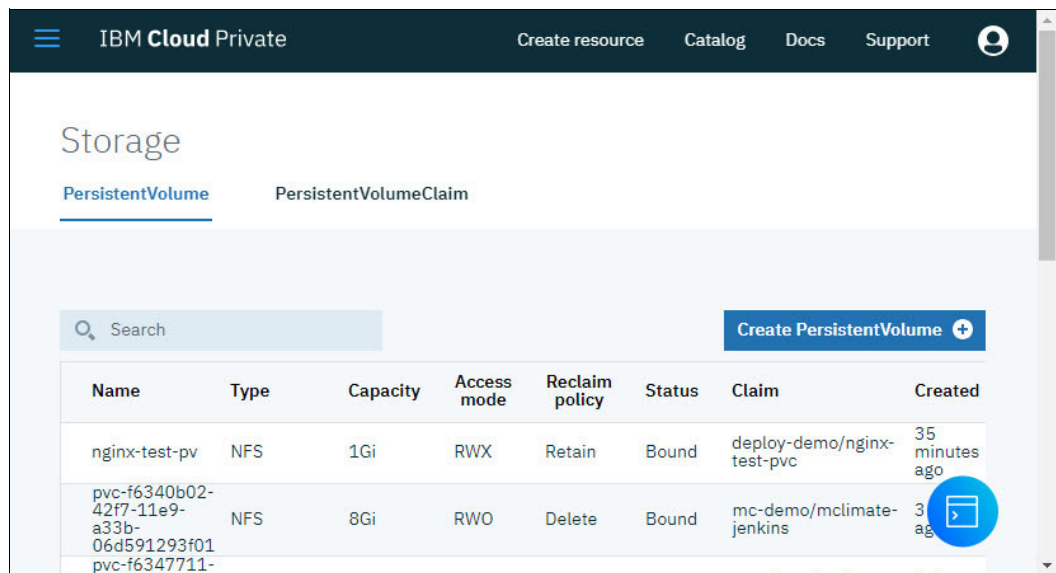


Figure 1-15 Creating PersistentVolume

4. In the General tab, complete the following information, as shown in Figure 1-16:
- Name: nginx
 - Capacity: 1
 - Access mode: Read write many

The screenshot shows a 'Create PersistentVolume' dialog box. At the top, it says 'PERSISTENTVOLUME' and 'JSON mode' with a toggle switch set to 'Off'. The title 'Create PersistentVolume' is prominently displayed. On the left, there is a sidebar with three tabs: 'General' (selected), 'Labels', and 'Parameters'. The main area contains the following fields:

- Name ***: A text input field containing 'nginx-pv'.
- Storage class name**: An empty text input field.
- Capacity ***: A numeric input field containing '1'.
- Unit**: A dropdown menu currently showing 'Gi'.
- Access mode**: A dropdown menu currently showing 'Read write many'.

At the bottom right, there are two buttons: 'Cancel' and 'Create'.

Figure 1-16 Create Persistence Volume - General tab

5. Click the **Labels** tab. Then, add Label: type, Value: nginx-nfs, as shown in Figure 1-17.

PERSISTENTVOLUME JSON mode X

Create PersistentVolume

Off ☐ On

General

Labels

Parameters

Label	Value
type	nginx-nfs

Add label +

Cancel Create

Figure 1-17 Create Persistent Volume: Labels tab

6. Click the **Parameters** tab and add the following parameters, as shown in Figure 1-18 on page 40:
- Server: {IP address of the NFS server}
 - Path: /redbooks

PERSISTENTVOLUME JSON mode Off On

Create PersistentVolume

General
Labels
Parameters

Key *	Value *	
server	174.37.18.205	⊖
Key *	Value * ⓘ	
path	/redbooks	⊖

[Add parameter](#) +

Cancel
Create

Figure 1-18 Create Persistent Volume: Parameters tab

7. Click **Create**.

Your persistence volume should appear as “Available” in the status, as shown in Figure 1-19. This process can take a few seconds. Also, notice that persistence volume claim is not bound yet to this persistent volume.

IBM Cloud Private Create resource Catalog Docs Support 👤

Storage

PersistentVolume PersistentVolumeClaim

🔍 Search Create PersistentVolume +

Name	Type	Capacity	Access mode	Reclaim policy	Status	Claim	Created
nginx-pv	NFS	1Gi	RWX	Retain	Available		1 minute ago

47

Figure 1-19 Created Persistence Volume

8. Click **PersistentVolumeClaim** and then, click **Create PersistentVolumeClaim**, as shown in Figure 1-20 on page 41.

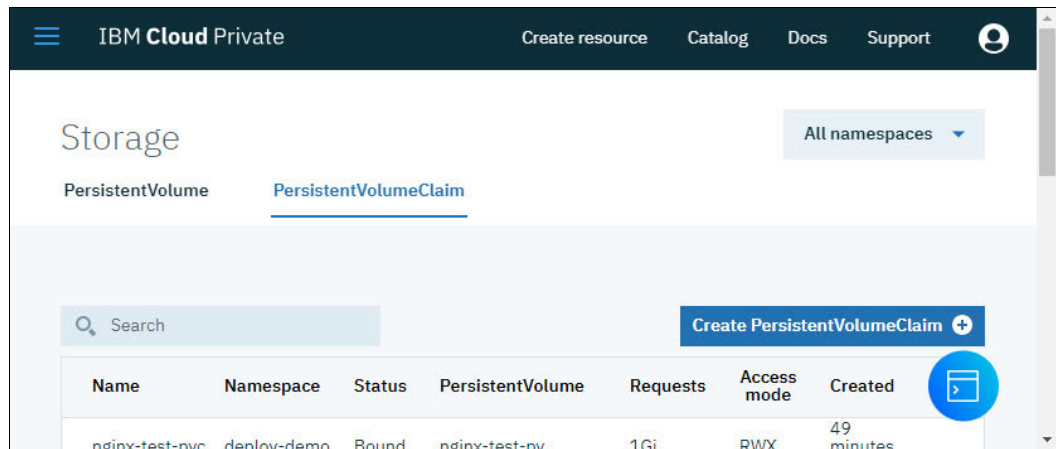


Figure 1-20 Create PersistenceVolumeClaim

9. Complete the following parameters in the Create PersistenceVolumeClaim dialog box:

- Name: nginx-pvc
- Namespace: deploy-demo
- Storage requests: 1
- Access mode: Read write many
- Volume selector label: type
- Value: nginx-nfs

10. Click **Create**, as shown in Figure 1-21.

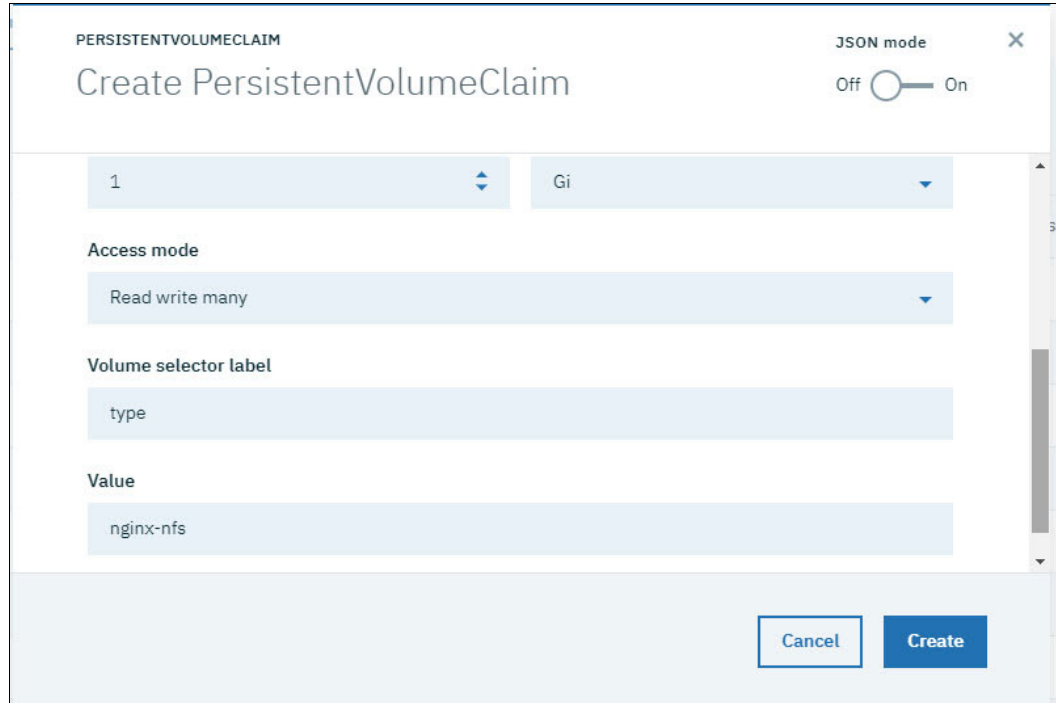
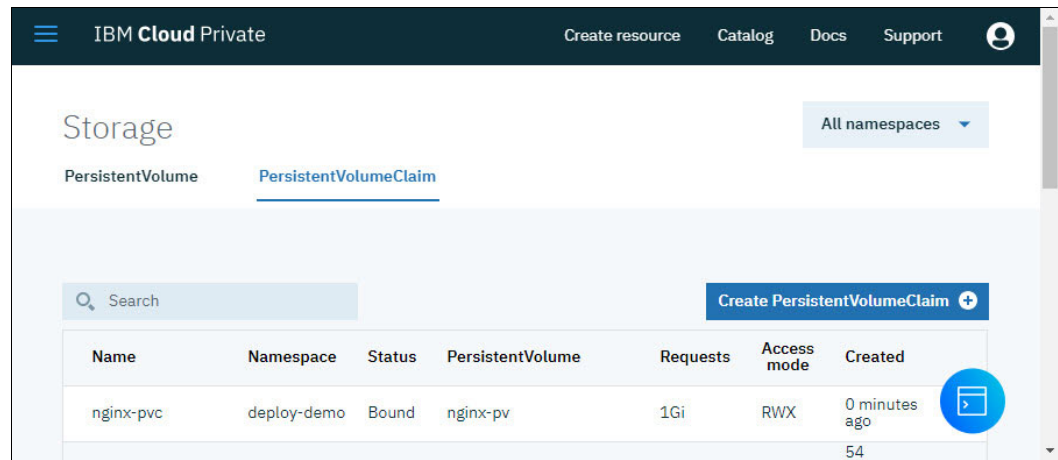


Figure 1-21 Create PersistentVolumeClaim dialog box

11. Wait until the persistent volume claim is bound to the persistent volume, as shown in Figure 1-22.



The screenshot shows the IBM Cloud Private interface for the Storage section. The 'PersistentVolumeClaim' tab is selected. A table lists the claims, with one entry 'nginx-pvc' in the 'deploy-demo' namespace, which is 'Bound' to the 'nginx-pv' persistent volume. The table also shows 'Requests' (1Gi), 'Access mode' (RWX), and 'Created' (0 minutes ago). A blue circle with a play icon highlights the 'nginx-pvc' entry.

Name	Namespace	Status	PersistentVolume	Requests	Access mode	Created
nginx-pvc	deploy-demo	Bound	nginx-pv	1Gi	RWX	0 minutes ago

Figure 1-22 Persistent volume claim list

1.7.3 Creating a Nginx app

Complete the following steps to create a Nginx web server:

1. Click **Catalog** in the tool bar.
2. Locate and click **ibm-nginx-dev**, as shown in Figure 1-23.

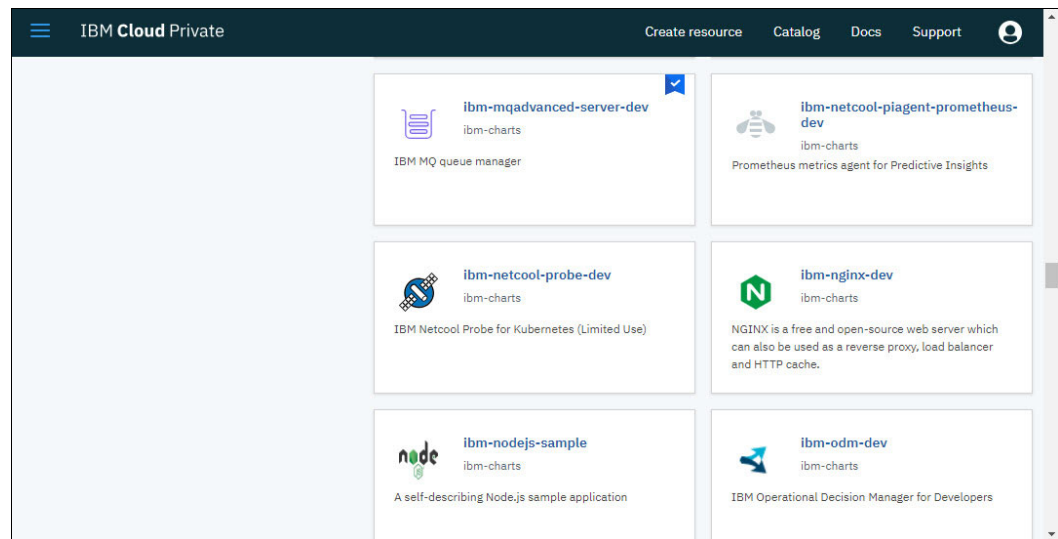


Figure 1-23 Catalog

3. Read the chart details. Then, click **Configure**, as shown in Figure 1-24.

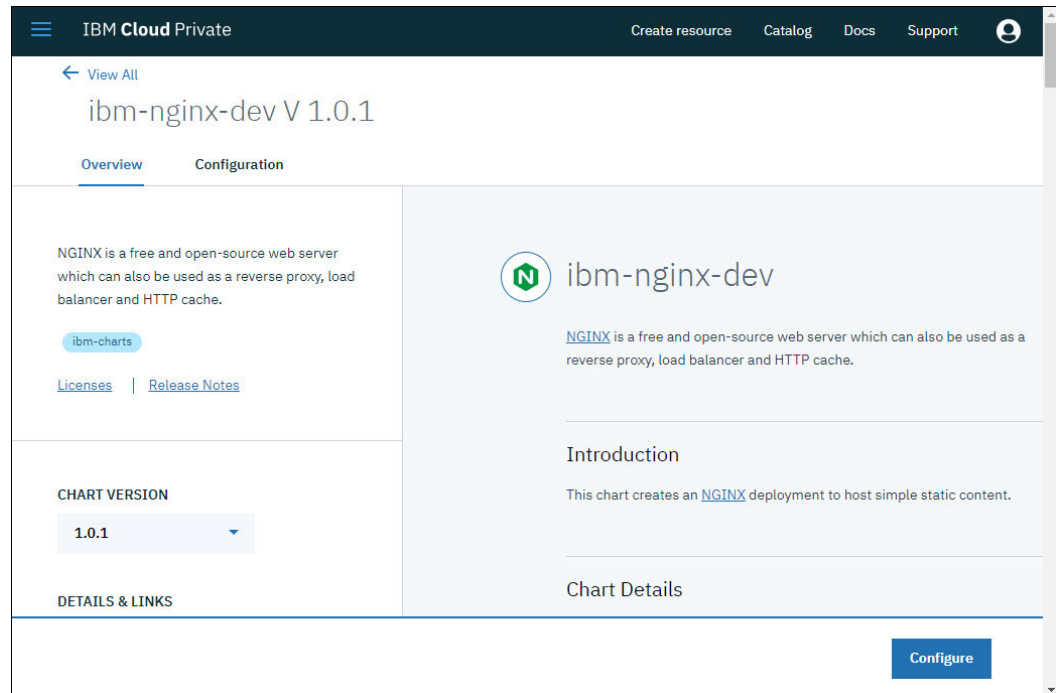


Figure 1-24 Nginx chart details

4. Add the following details to the configurations:
 - Helm release name: nginx-dev
 - Target namespace: deploy-demo
5. Read the license and click **I have read and agreed to the License agreement**.
6. Click **All parameters**.
7. Click **Enable volume mounting content** * and complete the following parameters to bind the pod with the persistence volume claim that was created earlier:
 - Access mode: ReadWriteMany
 - existingClaimName: nginx-pvc

8. Click **Install**, as shown in Figure 1-25.

The screenshot shows the 'Create resource' page for Nginx Helm release in the IBM Cloud Private console. The page has a dark blue header with the IBM Cloud Private logo and navigation links: 'Create resource', 'Catalog', 'Docs', 'Support', and a user profile icon. The main content area is light blue and contains several configuration sections. At the top, there are two input fields labeled 'Label to select the volume' and 'Value of the label to select the volume', both with 'Enter value' placeholder text. Below these is a section titled 'Content volume configuration' which includes a checked checkbox labeled 'Enable volume mounting content *' with an information icon. Underneath, there are two more input fields: 'Access mode *' with a dropdown menu showing 'ReadWriteMany' and 'existingClaimName' with the value 'nginx-pvc'. At the bottom of the configuration section, there are two more input fields labeled 'Label to select the volume' and 'Value of the label to select the volume', both with 'Enter value' placeholder text. Below these is a section titled 'Readiness probe'. At the bottom right of the page, there are two buttons: 'Cancel' and 'Install'.

Figure 1-25 Create Nginx Helm release

9. Click **View Helm Release** to view the status of deployment.

10.Wait until the pod is available, as shown in Figure 1-26.

The screenshot shows the IBM Cloud Private interface for a Helm release named 'nginx-dev'. The release is in a 'Deployed' state, indicated by a green dot. The update timestamp is 'March 13, 2019 at 3:22 PM'. A 'Launch' button is visible in the top right.

Details and Upgrades

CHART NAME	CURRENT VERSION	AVAILABLE VERSION	
ibm-nginx-dev	1.0.1	1.0.1	<button>Upgrade</button>
NAMESPACE	Installed: March 13, 2019 → Release Notes	Released: July 12, 2018 → Release Notes	<button>Rollback</button>
deploy-demo			

Deployment

NAME	DESIRED	CURRENT	UP TO DATE	AVAILABLE	AGE
nginx-dev-ibm-nginx-dev-nginx	1	1	1	1	2m

Pod

NAME	READY	STATUS	RESTARTS	AGE	
nginx-dev-ibm-nginx-dev-nginx-85f749dd5f-h6t4j	1/1	Running	0	2m	View

A blue circular button with a right arrow and a minus sign is overlaid on the bottom right of the Pod table.

Figure 1-26 Helm release status

11. Expose the service as NodePort to test it. Click **Menu** → **Network Access** → **Services**, as shown in Figure 1-27.

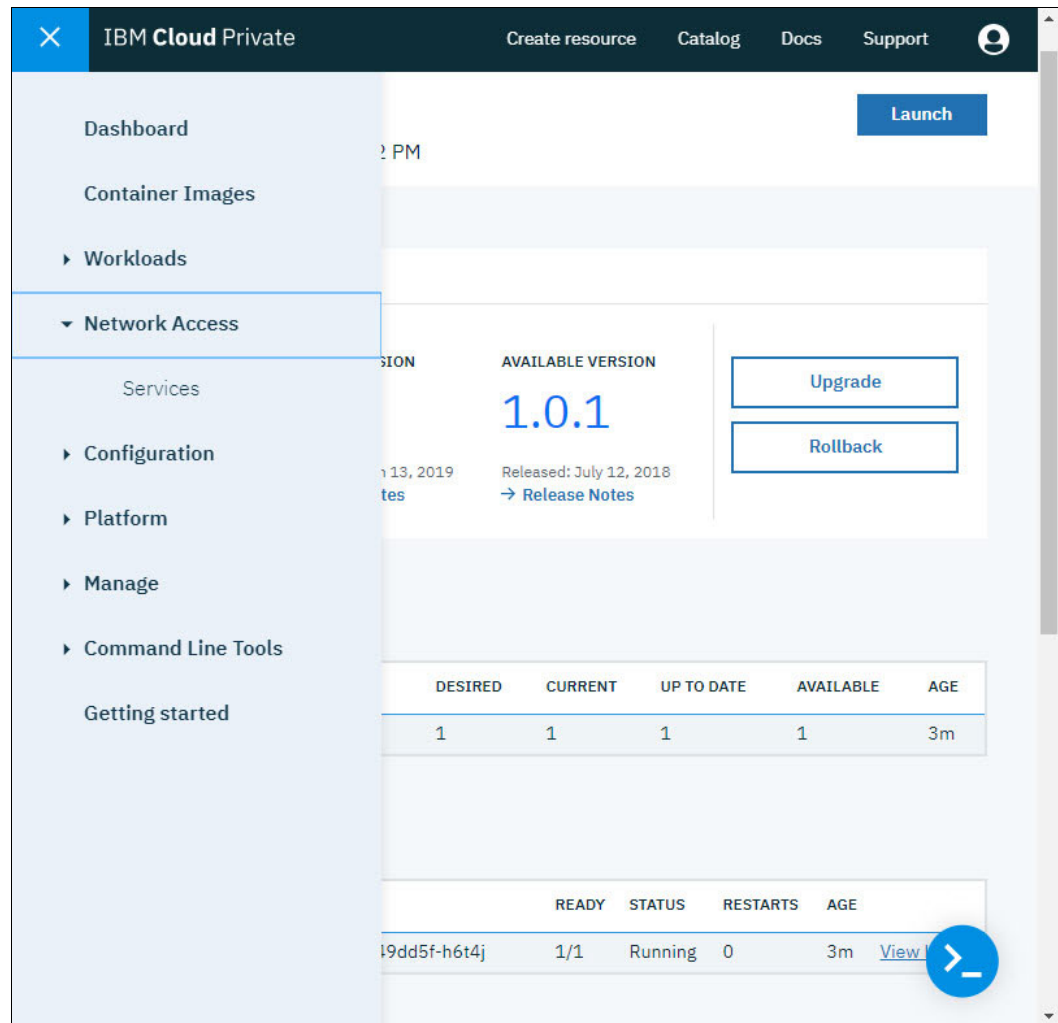


Figure 1-27 Menu -> Network Access

12. Locate `nginx-dev-ibm-nginx-dev`. Then, click and select **Edit**, as shown in Figure 1-28.

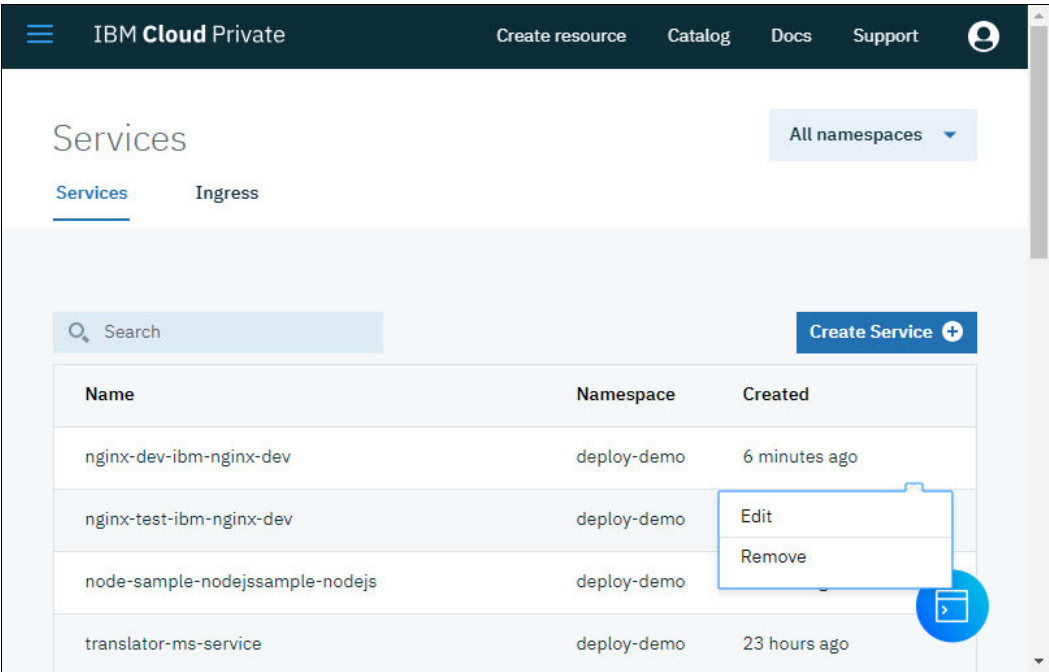


Figure 1-28 Edit menu for Services

13. Change the service type from ClusterIP to NodePort, as shown in Figure 1-29.

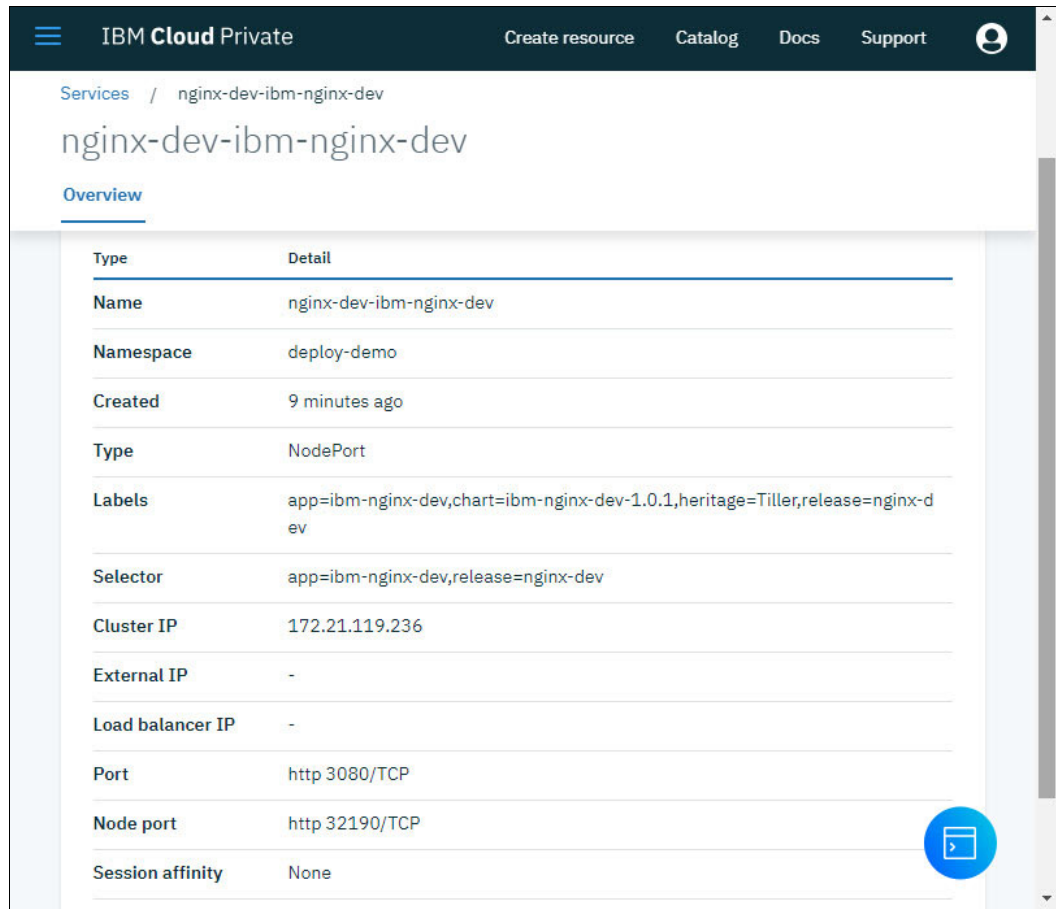


Figure 1-29 Edit Service

14. Click **Submit**.

15. Click `nginx-dev-ibm-nginx-dev` to view the details.

The Node port is displayed, as shown in Figure 1-30.



The screenshot shows the IBM Cloud Private console interface. At the top, there's a navigation bar with 'IBM Cloud Private' and links for 'Create resource', 'Catalog', 'Docs', and 'Support'. Below this, the breadcrumb 'Services / nginx-dev-ibm-nginx-dev' is visible. The main heading is 'nginx-dev-ibm-nginx-dev' with an 'Overview' tab selected. A table displays the service details:

Type	Detail
Name	nginx-dev-ibm-nginx-dev
Namespace	deploy-demo
Created	9 minutes ago
Type	NodePort
Labels	app=ibm-nginx-dev,chart=ibm-nginx-dev-1.0.1,heritage=Tiller,release=nginx-dev
Selector	app=ibm-nginx-dev,release=nginx-dev
Cluster IP	172.21.119.236
External IP	-
Load balancer IP	-
Port	http 3080/TCP
Node port	http 32190/TCP
Session affinity	None

Figure 1-30 Service details

16. Browse to {Proxy Node}:{Node Port} from your web browser. You see the content of the index.html that you specified as described in 1.7.1, “Prerequisite: Setting up an NFS server” on page 34 (see Figure 1-31).

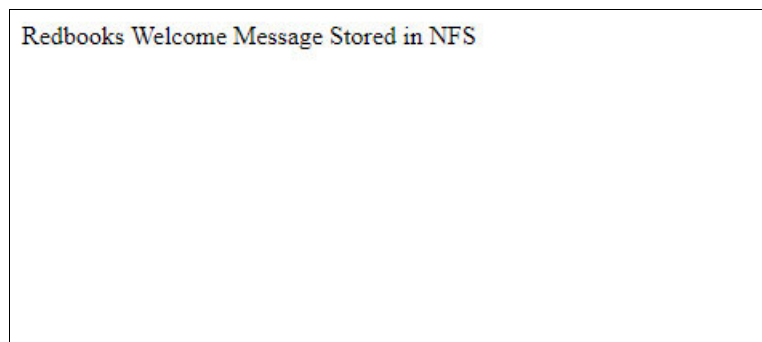


Figure 1-31 Demo application output



Helm and application packaging

IBM Cloud Private is a cloud native environment in which automation plays a significant role. In the Kubernetes world, manual deployments are not sustainable in the end, thus the requirement for proper application packaging and automated deployments.

In this chapter, we discuss the Helm charts, a *de facto standard* of application packaging for Kubernetes environments.

This chapter includes the following topics:

- ▶ 2.1, “Introduction to Helm” on page 50
- ▶ 2.2, “Helm chart structure” on page 52
- ▶ 2.3, “Creating a chart” on page 59
- ▶ 2.4, “IBM Cloud Paks” on page 74

2.1 Introduction to Helm

Helm is an open source package manager for Kubernetes that is governed by the Cloud Native Computing Foundation (CNCF). By using Helm, developers can define, version, and securely distribute their applications. Kubernetes cluster administrators can easily manage instances of applications that are running in their clusters.

The [Helm public website](#) provides many materials about the project. For the purposes of this book, we introduce the following key concepts:

Helm client	A binary program that is written in Go, which can be used to interact with and manage the Helm charts and deploy them to the Kubernetes environment.
Tiller	A server process that is installed in the Kubernetes environment that exposes API endpoints and can accept commands from the Helm client. By default, it operates with the permissions of the service account that is used to run the process. However, in IBM Cloud Private, Tiller also is secured by using Identity and Access Management (IAM) service that controls access rights.
Helm chart	A structured collection of files and directories that contain definitions of the Kubernetes objects that are needed for running an application and metadata that describes the application and helps users to manage the deployment.
Helm release	An instance of the Helm chart that runs within the Kubernetes environment.
Helm repo	A simple HTTP server that exposes the <code>index.yaml</code> file that points to the location of one or more Helm charts that are packaged in the <code>.tgz</code> format. There are several public, curated repositories that contain hundreds of ready-to-use applications; however, enterprises can run their own repositories by using a plain HTTP server or a more sophisticated solution, such as JFrog Artifactory.

The Helm Client is a command-line client for users. The client is responsible for the following domains:

- ▶ Developing local charts
- ▶ Managing repositories
- ▶ Interacting with the Tiller server
- ▶ Sending charts to be installed
- ▶ Asking for information about releases
- ▶ Requesting upgrading or uninstalling existing releases

The Tiller is an in-cluster server that interacts with the Helm client and Kubernetes API server. The server is responsible for the following tasks:

- ▶ Listening for incoming requests from the Helm client
- ▶ Combining a chart and configuration to build a release
- ▶ Installing charts into Kubernetes, and then tracking the subsequent release
- ▶ Upgrading and uninstalling charts by interacting with Kubernetes

The client is responsible for managing the charts, and the server is responsible for managing the releases.

The Helm client and Tiller are written in the Go programming language, and use the gRPC protocol suite to interact. It is recommended to use a matching versions of the Helm client and Tiller. Versions of the components can be verified, as shown in Example 2-1.

Example 2-1 Verification of Helm and Tiller versions

```
helm version --tls
```

```
Client: &version.Version{SemVer:"v2.7.3+icp",
GitCommit:"27442e4cfd324d8f82f935fe0b7b492994d4c289", GitTreeState:"dirty"}
Server: &version.Version{SemVer:"v2.9.1+icp",
GitCommit:"8ddf4db6a545dc609539ad8171400f6869c61d8d", GitTreeState:"clean"}
```

Tip: When the versions do not match, the provisioning in most cases still works, but the use of such a setup is not recommended.

IBM Cloud Private implements TLS layer between the Helm client and Tiller, because by default (at the time of this writing), the Helm project did not enforce any security. To successfully run the Helm commands in an IBM Cloud Private environment, you first must authenticate to the IBM Cloud Private cluster, as shown in Example 2-2.

Example 2-2 Authenticating to an IBM Cloud Private cluster

```
cloudctl login -a https://mycluster.icp:8443 --skip-ssl-validation
```

```
Username> admin
```

```
Password>
```

```
Authenticating...
```

```
OK
```

```
Targeted account mycluster.icp Account (id-mycluster.icp-account)
```

```
Select a namespace:
```

1. cert-manager
2. default
3. istio-system
4. kube-public
5. kube-system
6. platform
7. services

```
Enter a number> 2
```

```
Targeted namespace default
```

```
Configuring kubectl ...
```

```
Property "clusters.mycluster.icp" unset.
```

```
Property "users.mycluster.icp" unset.
```

```
Property "contexts.mycluster.icp" unset.
```

```
Cluster "mycluster.icp" set.
```

```
User "mycluster.icp-user" set.
```

```
Context "mycluster.icp-context" created.
```

```
Switched to context "mycluster.icp-context".
```

```
OK
```

```
Configuring helm: /Users/guest/.helm
```

OK

Upon successful login, the use of the `cloudctl` command creates a `.helm` subdirectory in your home directory that contains the following files:

```
~/helm
... ca.pem
... cert.pem
... key.pem
```

To run the Helm commands, it is useful to set the `HELM_HOME` environment variable by using the `export HELM_HOME=~/helm` command.

2.2 Helm chart structure

A Helm chart is a structured collection of files, typically in a YAML format. The following types of elements are used within a Helm chart:

- ▶ Metadata
- ▶ Templates
- ▶ Values

2.2.1 Helm chart metadata

Several files within a Helm chart are the metadata that describes the chart. The key file (and the only one that is required) is the `Chart.yaml` file. Sample content of `Chart.yaml` is shown in Example 2-3.

Example 2-3 Sample content of a `Chart.yaml` file

```
apiVersion: v1
appVersion: "1.0"
description: A Helm chart for Kubernetes
name: sample
version: 0.1.0
```

Although most of the tags are self-explanatory, the difference between the `version` and `appVersion` tags is important. The `version` tag refers to the Helm chart version; the `appVersion` tag refers to the application being deployed. The `version` tag value is automatically appended to a release name during the deployment time and added to the chart name during the chart packaging.

It is recommended to automatically update those tags as a part of your DevOps pipeline. Packages in repositories are identified by name plus version.

A `Chart.yaml` file must specify a version in the Semantic Versioning 2.0.0 format. As defined by Semantic Versioning 2.0.0, given a version number `MAJOR.MINOR.PATCH`, increment the following components:

- MAJOR version when you make incompatible API changes
- MINOR version when you add functionality in a backwards-compatible manner
- PATCH version when you make backwards-compatible bug fixes

Next, we provide some general guidelines for the use of version numbers to ease the process of deploying your Helm charts.

Using a MAJOR version number

MAJOR version zero (0.MINOR.PATCH) is for initial development. Version 1.0.0 defines the start of production use. MAJOR version *must* be incremented if any backwards incompatible changes are introduced, such as the following examples:

- ▶ Major chart changes.
- ▶ Changes in Helm installation variables usage or definitions in which previous Helm installation commands do not work.
- ▶ New resource prerequisites in which previous Helm installation commands do not work.

Using a MINOR and PATCH version numbers

Consider the following rules that you *must* follow in your chart development practice:

- ▶ Reset MINOR and PATCH versions to 0 when MAJOR version is incremented.
- ▶ Increment a MINOR version if new backwards compatible functionality is added.
- ▶ Reset PATCH versions to 0 when MINOR version is incremented.
- ▶ Update PATCH versions for bug fixes.

Two other pieces of metadata within a Helm chart refer to NOTES.txt and LICENSE.txt:

- ▶ NOTES.txt often is in the templates subdirectory and is used to display some contextual usage information upon successful deployment of the chart, such as Access URL.
- ▶ LICENSE.txt is an optional file that is used to include Licensing information, specifically if the Chart embeds some of the open source technologies that are governed by specific license terms.

The last element of metadata is IBM Cloud Private-specific. A file that is named `values-metadata.yaml` can contain more information that is used by the Catalog app to format the fields that are displayed to user in a browser and validate field inputs. For more information, see “Using IBM Cloud Private Catalog features (values-metadata)” on page 74.

2.2.2 Helm templates

All files that are in the `\templates` subdirectory (except NOTES.txt) are treated as Kubernetes manifests (resource templates) and are sequentially processed during the chart deployment. File names are not enforced in any way; however, the recommended practice is to keep the file names that are relevant to the content with `.yaml` suffix for the Kubernetes resources and `.tpl` for the template files that produce no formatted content. The files that include a name that starts with “_” are assumed partials that do not produce any direct output.

Tip: For more information about the current list of the Helm template recommended practices, see this [GitHub web page](#).

As an example, the default `_helpers.tpl` file is shown in Example 2-4.

Example 2-4 Sample content of `_helpers.tpl`

```
{{/* vim: set filetype=mustache: */}}
{{/*
Expand the name of the chart.
*/}}
{{- define "sample.name" -}}
{{- default .Chart.Name .Values.nameOverride | trunc 63 | trimSuffix "-" -}}
{{- end -}}
```

```

{{/*
Create a default fully qualified app name.
We truncate at 63 chars because some Kubernetes name fields are limited to this
(by the DNS naming spec).
If release name contains chart name it will be used as a full name.
*/}}
{{- define "sample.fullname" -}}
{{- if .Values.fullnameOverride -}}
{{- .Values.fullnameOverride | trunc 63 | trimSuffix "-" -}}
{{- else -}}
{{- $name := default .Chart.Name .Values.nameOverride -}}
{{- if contains $name .Release.Name -}}
{{- .Release.Name | trunc 63 | trimSuffix "-" -}}
{{- else -}}
{{- printf "%s-%s" .Release.Name $name | trunc 63 | trimSuffix "-" -}}
{{- end -}}
{{- end -}}
{{- end -}}

{{/*
Create chart name and version as used by the chart label.
*/}}
{{- define "sample.chart" -}}
{{- printf "%s-%s" .Chart.Name .Chart.Version | replace "+" "_" | trunc 63 |
trimSuffix "-" -}}
{{- end -}}

```

You can refer to the templating functions that are defined in `_helper.tpl` by using the `template` or `tpl` directive within your template files. For example, to call `sample.fullname`, you can include the following reference in your template file:

```
{{ template "sample.fullname" . }}
```

In this example, the dot (“.”) is the top-level context that is passed into the template and `sample` is the chart name.

When packaged, all of the Helm resources run through a Go templating engine that dynamically generates output that is based on the template code evaluation. The use of this templating language for each deployment can be customized with a unique set of values for the release.

2.2.3 Helm values

To allow users to customize the chart deployment, it is possible to replace static parts within a template with references to the content of the `values.yaml` file. The `values.yaml` file can define whatever names you want by following a hierarchical YAML structure. A sample `values.yaml` file is shown in Example 2-5.

Example 2-5 Sample values.yaml file

```

# Default values for sample.
# This is a YAML-formatted file.
# Declare variables to be passed into your templates.

replicaCount: 1

```

```

image:
  repository: nginx
  tag: stable
  pullPolicy: IfNotPresent

service:
  type: ClusterIP
  port: 80

ingress:
  enabled: false
  annotations: {}
    # kubernetes.io/ingress.class: nginx
    # kubernetes.io/tls-acme: "true"
  path: /
  hosts:
    - chart-example.local
  tls: []
    # - secretName: chart-example-tls
    #   hosts:
    #     - chart-example.local

resources: {}
  # We usually recommend not to specify default resources and to leave this as a
  # conscious
  # choice for the user. This also increases chances charts run on environments
  # with little
  # resources, such as Minikube. If you do want to specify resources, uncomment
  # the following
  # lines, adjust them as necessary, and remove the curly braces after
  'resources:'.
  # limits:
  #   cpu: 100m
  #   memory: 128Mi
  # requests:
  #   cpu: 100m
  #   memory: 128Mi

nodeSelector: {}

tolerations: []

affinity: {}

```

Whatever you define in the `values.yaml` file then can be referenced in the templates by using `{{ .Values.<yamlpath> }}`. The use this example `{{ .Values.image.repository }}` is resolved into “nginx”.

In addition to the entries that are directly specified in the `values.yaml` file, Helm provides a list of predefined values that refers to the release (for example, `Release.Name`) or the chart metadata (for example, `Chart.Version`). For more information about a full list of predefined values, see [this web page](#).

Tip: For more information about a current list of recommended practices for the values file, see this [GitHub web page](#).

2.2.4 Templating language hints and tips

The templating code is embedded into the Kubernetes resource file that is submitted by Tiller to the Kubernetes API, which replaces the values of fields with the content of the `values.yaml` file and some tags that are generated at runtime (for example, the release name when it is not specified up front). It is recommended to include all of the fields that you want to allow the user to customize in the `values.yaml` file.

This book is not meant to extensively cover all of the template functions available. For more information, see the Helm publication [The Chart Template Developer's Guide](#).

If you are new to Helm, this section might not be as useful until you develop a Helm chart. For more information, see 2.3, “Creating a chart” on page 59.

Next, we review several basic, but instructive, examples.

White space trimming

White space in the YAML files can be a problem, and they are not always easy to debug. When developing Helm charts, take care when template functions are used that you do not generate unwanted white spaces (and later, bad parameter mapping), as shown in Example 2-6.

Example 2-6 Space to the left of the function not trimmed

```
data:
  {{ if .Values.key1.value }}
  key1: {{ .Values.key1.value }}
  {{ end }}
```

The use of such syntax results in an error that is similar to the following example

```
“Error: YAML parse error on mychart/templates/configmap.yaml: error converting
YAML to JSON: yaml: line 9: did not find expected key”
```

It produces the following result:

```
data:
  key1: value1
```

As you can see, the `key1` parameter is incorrectly indented. You can trim the white space at the start of a template function by using a hyphen (-), as shown in Example 2-7.

Example 2-7 Space to the left of the function trimmed

```
data:
  {{- if .Values.key1.value }}
  key1: {{ .Values.key1.value }}
  {{- end }}
```

This produces the wanted result:

```
data:
  key1: value1
```


Conditional statements

By using templating language, you can define conditional blocks within a template that are used based on the result of the conditional expression. Such conditional blocks are defined by using the If/Else clause.

In Example 2-8, an “if” block prints `mug: true` if the `.Values.favorite.drink` value is 'coffee', else it will print `glass: true`.

Example 2-8 If/Else clause example

```
beverage:
  {{- if eq .Values.favorite.drink "coffee"}}
  mug: true
  {{- else }}
  glass: true
  {{- end}}
```

If the value that is passed to the template engine is “coffee”, the following output is generated when the chart is packaged:

```
beverage:
  mug: true
```

Scoping the values

When defining the template files, references to the variables that are defined in the `values.yaml` file can become long.

The use of the `with` clause sets the scope of the `.Values`, so that references to a particular variable can be shorter. Example 2-9 shows the syntax with a value scope of `.Values.favorite.drink`.

Example 2-9 Reference to the values using the ‘with’ clause

```
favorite:
  {{- with .Values.favorite.drink }}
  champagne: {{ .champagne }}
  coffee: {{ .coffee }}
  tea: {{ .tea }}
  {{- end }}
```

The outcome of this template is the same as from template that is shown in Example 2-10.

Example 2-10 Reference to the values without a scoping ‘with’ clause

```
favorite:
  champagne: {{ .Values.favorite.drink.champagne }}
  coffee: {{ .Values.favorite.drink.coffee }}
  tea: {{ .Values.favorite.drink.tea }}
```

Range

The range function allows you to loop through a list (and apply a prefix or suffix). It iterates through an array and prints the output. Example 2-11 show a sample syntax to iterate through an array of items.

Example 2-11 Sample syntax to iterate through an array of items

```
items: |-
  {{- range .Values.items }}
  - {{ . | quote }}
  {{- end }}
```

When used with the ["item1","item2","item3"] array as input, the following output is produced:

```
items:
  - item1
  - item2
  - item3
```

toYaml/toJson

These functions are useful if you want users to supply valid YAML or JSON data as a parameter. In this case, the built-in parser attempts to output correct YAML or JSON, so it is important that the data is supplied correctly. Any valid YAML or JSON formatted data is accepted.

As shown in Example 2-12, a simple JSON list is passed in the format {"key1":"value1","key2":"value2","key3":"value3"} by using a template.

Example 2-12 Sample toYaml clause

```
keys:
  {{- toYaml .Values.keys | indent 2 }}
```

The following output is produced:

```
keys:
  key1: value1
  key2: value2
  key3: value3
```

The indent function is used here to indent the output by two spaces.

Because more complex structured data can also be used, ensure that the data you supply is correct JSON. One limitation to YAML is if the supplied data is a literal string (enclosed in single quotation marks), it is still valid YAML. Therefore, it is printed as a literal string and not expanded to YAML.

Commenting

Commenting within a template can be done by using '{{/*' to open and '*/}}' to close, as shown in Example 2-13.

Example 2-13 Sample comment section

```
{{/*
This is a comment that describes the next section
```

```
* /}}}
```

Any text that is put between comment markers is ignored by the template engine and does not appear in the output.

2.3 Creating a chart

A Helm chart can be created by using several methods. In this section, we describe the simplest method: the use of the Helm CLI command. It is also possible to easily generate a draft Helm chart for your project by using [Microclimate](#) or [IBM Cloud Developer Tools](#).

To create a Helm chart, run the following command:

```
helm create <your_app_name>
```

Running this command creates a directory that is named `<your_app_name>`. Several new directories and files are included within this directory, as shown in Example 2-14. Because the name of the chart that is created is used in several places, it is best to use a suitable chart name.

Example 2-14 Content of a generic sample Helm chart

```
<your_app_name>/
... Chart.yaml
... charts
... templates
.  ... NOTES.txt
.  ... _helpers.tpl
.  ... deployment.yaml
.  ... ingress.yaml
.  ... service.yaml
... values.yaml
```

The following resources are created for you:

charts/	Directory to contain all of the embedded charts.
values.yaml	The initial values file that contains a default value of all variables that are used in template files in templates directory.
Chart.yaml	File that contains information about the chart, such as description and versioning.
templates/	Directory to contain all chart resources.
templates/NOTES.txt	Default file that is displayed postinstallation that contains instructions to the user about how to use the chart.
templates/_helpers.tpl	Initial template file that is used to provide templated data to the other files within the chart.
templates/deployment.yaml	Initial sample Kubernetes deployment.
templates/ingress.yaml	Initial sample Kubernetes ingress.

templates/service.yaml

Initial sample Kubernetes service.

We are now ready to customize the default resources to match our application.

Tip: Before continuing with development of your own Helm chart, we strongly recommend that you to review Helm's [The Chart Best Practices Guide](#).

2.3.1 Making your Helm chart flexible

Several other best practices are available that help you define a well-developed Helm template. One of the practices is to allow the chart to be installed in multiple copies. To achieve this task, avoid any hardcoded resource names; instead, use the helper function to render unique names for all the resources within a specific release.

The default helper file `_helpers.tpl` includes a set of reusable functions that helps with this task. You can customize the helper files and share them between different charts as a reusable asset.

Requirements for using namespaces with Helm charts

While developing a Helm chart, adhere to the following rules:

- ▶ Charts do not specify a namespace in any resource definitions.
The namespace is specified when the chart is installed, and the namespace should not be specified in any template resource YAML files.
- ▶ Install a chart and subcharts into only one namespace.
Helm does not support installing a chart in multiple namespaces. Helm supports creating and managing resources only in the target namespace of the release. Doing otherwise breaks the ability to upgrade or remove a Helm release and can result in orphaned resources or worse (consequences are unpredictable).
- ▶ A chart does not require installation into any specifically named namespace, including the default namespace.
- ▶ A chart is installable multiple times in a single namespace.
This rule requires the resource names to be qualified by the Helm release name to provide a unique resource name.

2.3.2 Sample application

For the rest of this chapter, we use a sample application: a Node.js microservices that implements the REST API and requires a MongoDB to operate. One way to start customizing the empty chart is to take the `.yaml` files that you use for testing the manual deployment and put them into the `templates` subdirectory. Then, you can edit the files by replacing any fixed values with references to the `values.yaml` (and adding the appropriate entries there).

Pod definition

When testing the application locally, you can run a single instance of the application container. When deploying to a Kubernetes cluster, you *must* follow the Kubernetes logic.

The smallest object that can be deployed to a Kubernetes cluster is called *pod*. The following description for a pod is provided in the Kubernetes documentation:

"A Pod is the basic building block of Kubernetes—the smallest and simplest unit in the Kubernetes object model that you create or deploy. A Pod represents a running process on your cluster.

A Pod encapsulates an application container (or, in some cases, multiple containers), storage resources, a unique network IP, and options that govern how the container(s) should run. A Pod represents a unit of deployment: a single instance of an application in Kubernetes, which might consist of either a single container or a small number of containers that are tightly coupled and that share resources."

Although it is possible to define pods directly as Kubernetes resources within a Helm chart, it is not a recommended method because this process limits your options for managing your application lifecycle.

Rather than creating pods on your own, allow the dedicated controller do this work for you. The available controllers include the following examples:

- ▶ ReplicaSet
- ▶ DaemonSet
- ▶ StatefulSet
- ▶ Deployment

Deployment definition

In our example, we use the Deployment definition, as shown in Example 2-15. This definition describes a wanted state of application resources that are running in a Kubernetes cluster.

Example 2-15 Deployment template definition

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: "{{ template "fullname" . }}-deployment"
  labels:
    chart: '{{ template "chart" . }}'
spec:
  replicas: {{ .Values.replicaCount }}
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 0
      maxSurge: 1
  revisionHistoryLimit: {{ .Values.revisionHistoryLimit }}
  template:
    metadata:
      labels:
        app: "{{ template "fullname" . }}-selector"
        version: "current"
    spec:
      containers:
        - name: "{{ .Chart.Name }}"
          image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
          imagePullPolicy: {{ .Values.image.pullPolicy }}
          livenessProbe:
            httpGet:
              path: /health
```

```

        port: {{ .Values.service.servicePort }}
        initialDelaySeconds: {{ .Values.livenessProbe.initialDelaySeconds }}
        periodSeconds: {{ .Values.livenessProbe.periodSeconds }}
    ports:
    - containerPort: {{ .Values.service.servicePort }}
    resources:
        requests:
            cpu: "{{ .Values.image.resources.requests.cpu }}"
            memory: "{{ .Values.image.resources.requests.memory }}"
    env:
    - name: PORT
      value: "{{ .Values.service.servicePort }}"
    - name: APPLICATION_NAME
      value: "{{ template \"fullname\" . }}"
      {{- $configmap := .Values.services.configMap | default (printf
"%s-configmap" (include "fullname" .)) }}
    - name: NODE_ENV
      value: "{{ default .Values.env.type \"production\" }}"
    - name: MONGO_HOST
      valueFrom:
        configMapKeyRef:
          name: "{{ $configmap }}"
          key: MONGO_HOST
    - name: MONGO_PORT
      valueFrom:
        configMapKeyRef:
          name: "{{ $configmap }}"
          key: MONGO_PORT
    - name: MONGO_DB_NAME
      valueFrom:
        configMapKeyRef:
          name: "{{ $configmap }}"
          key: MONGO_DB_NAME
    - name: MONGO_USER
      valueFrom:
        configMapKeyRef:
          name: "{{ $configmap }}"
          key: MONGO_USER
    - name: MONGO_PASS
      valueFrom:
        configMapKeyRef:
          name: "{{ $configmap }}"
          key: MONGO_PASS

```

As you can see in Example 2-15 on page 61, several values, such as `metadata.name`, `metadata.labels.app`, or `spec.template.spec.container.name`, are specified by using a `{{ template }}` keyword. This keyword instructs the templating engine to use a template that is defined in helper files (“_helper.tpl” specifically in our example).

Next, we review the `fullname` helper function (for convenience, a line-by-line breakdown of the code and the corresponding explanations are shown next):

```
{{- define "fullname" -}}
```

Definition of the function name:

```
{{- if .Values.fullNameOverride -}}
```

If during the Helm chart installation `fullNameOverride` value is provided, then:

```
{{- .Values.fullNameOverride | trunc 63 | trimSuffix "-" -}}
```

Output it, truncating to 63 characters and removing any trailing hyphens ("-"):

```
{{- else -}}
```

If not:

```
{{- $name := default .Chart.Name .Values.nameOverride -}}
```

Define `$name` temporary variable as name of the Helm chart or `nameOverride` value if provided:

```
{{- if contains $name .Release.Name -}}
```

If the newly defined `$name` variable contains a Helm release name:

```
{{- .Release.Name | trunc 63 | trimSuffix "-" -}}
```

Output release name, truncating it to 63 characters and removing any trailing hyphens ("-"):

```
{{- else -}}
```

Otherwise:

```
{{- printf "%s-%s" .Release.Name $name | trunc 63 | trimSuffix "-" -}}
```

Output release name plus hyphen ("-") plus `$name`, truncating the concatenated string to 63 characters and removing any trailing hyphens ("-"):

```
{{- end -}}
```

```
{{- end -}}
```

```
{{- end -}}
```

The last three lines are closures of the `if` statements that are used.

Tip: Value names (including `Override` in the name) are treated differently by linter and they do not have to be defined in the `values.yaml` file. They can be provided at runtime.

The goal of this function is to generate a unique name so the multiple deployments that are generated by this chart can coexist in the same target namespace.

Service definition

To expose your application to the world, an access method must be defined. Kubernetes pods feature a lifespan. They are created and expire under many circumstances, such as auto-healing by way of liveness, readiness probes, scaling up, scaling down, or performing rolling updates.

Thus, your workload must avoid working with static IP addresses because IP addresses cannot be relied upon to be stable over time. In complex applications that have pods that depend on other pods, it becomes untenable to track the changing IP addresses. Services were designed by Kubernetes to handle this problem.

A Kubernetes service defines a logical set of pods and a policy to access them, which is often called a *microservice*. It is this construct that must be mastered to take full advantage of Kubernetes.

As workloads mature, secure access becomes a critical component to providing enterprise ready workloads. It is important to use advanced concepts, such as ingress and network policies to access to Kubernetes resources.

Consider the following high-level guiding principles that immediately drive a level of security and isolation:

- ▶ Use Kubernetes service type of ClusterIP as the default:
 - Keep as much network traffic within the cluster network as possible.
 - If you do not specify a service type, ClusterIP is the default.
 - Services with type=ClusterIP handle traffic only within the cluster container network.
- ▶ Discourage the use of NodePort. Consider the use of an Ingress Controller/Ingress rule approach instead:
 - Nodeports are a simple way of exposing external access to a workload for initial development and testing. However, they expose more security concerns and are difficult to manage from an application and networking infrastructure perspective.
 - NodePort services expose “non-standard” ports in the 30000 - 32767 range externally on all the worker nodes in your cluster. All worker nodes are configured to listen on the specific NodePort that is assigned.

You now have each node externally accessible, which opens more paths that must be protected by way of fire walls, and so forth. Security products also are available that consider the use of NodePort to be something that they flag. For more information, see [this web page](#).

- It is more difficult for an application to make itself accessible when NodePorts are used in the real world. Most network ingress access is controlled by corporation fire walls that are managed by a networking team. The odds of having incoming access on a well-know port, such as 80, 443 are much greater than a dynamically generated port in the 30000 - 32767 range.

The same argument also holds for applications that want to use your service. They must be configured to communicate with “non-standard” ports and address similar firewall port requirements with their networking team.

Example 2-16 shows the service template definition of our application.

Example 2-16 Service template definition

```
apiVersion: v1
kind: Service
metadata:
  annotations:
    prometheus.io/scrape: 'true'
  name: "{{ template "fullname" . }}-api-service"
  labels:
    chart: "{{ template "chart" . }}"
spec:
  type: {{ .Values.service.type }}
  ports:
    - name: http
      port: {{ .Values.service.servicePort }}
  selector:
```



```
app: "{{ template "fullname" . }}"-selector"
```

Ingress definition

To expose the service to the outside traffic (coming from outside of an IBM Cloud Private cluster), you must specify an Ingress object. In Example 2-17, we show a sample ingress definition for the service.

Example 2-17 Ingress definition template

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: "{{ template "fullname" . }}"
  labels:
    app: "{{ template "fullname" . }}"
    chart: "{{ template "chart" . }}"
    release: "{{ .Release.Name }}"
    heritage: "{{ .Release.Service }}"
  annotations:
    ingress.kubernetes.io/backend-protocol: "HTTP"
    ingress.kubernetes.io/rewrite-target: /
    ingress.kubernetes.io/proxy-body-size: "0"
    ingress.kubernetes.io/secure-backends: "true"
    nginx.ingress.kubernetes.io/rewrite-target: /
    nginx.ingress.kubernetes.io/proxy-body-size: "0"
    nginx.ingress.kubernetes.io/secure-backends: "true"
    nginx.ingress.kubernetes.io/backend-protocol: "HTTP"
spec:
  tls:
    - secretName: "{{ .Release.Name }}-tls-secret"
      hosts:
        - "{{ .Values.service.ingresshost }}"
  rules:
    - host: "{{ .Values.service.ingresshost }}"
      http:
        paths:
          - path: /*
            backend:
              serviceName: "{{ template "fullname" . }}-api-service"
              servicePort: {{ .Values.service.servicePort }}
```

It is important to ensure that `serviceName` value in the Ingress definition matches the `metadata.name` that is provided in the service template; otherwise, it does not work. Annotations in the Ingress definition are translated into Ingress controller configuration files. In IBM Cloud Private, the default ingress is implemented by using `nginx`. For more information about the full list of possible annotations, see the [nginx ingress documentation](#).

You also might notice that the ingress definition that is shown in Example 2-17 includes the TLS section. This means that the port open for incoming traffic expects HTTPS protocol.

As of this writing, the Ingress supports only a single TLS port (443) and assumes TLS termination. If the TLS configuration section in an Ingress specifies different hosts, they are multiplexed on the same port according to the hostname that is specified through the SNI TLS extension (which is provided the Ingress controller supports SNI). The TLS secret must contain keys named `tls.crt` and `tls.key` that contain the certificate and private key to use for TLS.¹

Note: In Example 2-17 on page 65, duplicate annotations are listed. The set that begins with `ingress.kubernetes.io` is the old definition; the set that begins with `nginx.ingress.kubernetes.io` is the new definition. You can specify both, and the Ingress controller filters to use only the one it needs.

In our example, we use a dynamically generated self-signed certificate for testing convenience. It is not recommended to use such certificates for any production use.

Important: Because Helm is not secure enough to hide sensitive information in its release data, it is a recommended best practice to manually create any secrets that your chart needs before the chart is deployed. The user creates the secrets and then passes in the names of the secrets in their `values.yaml` file during the Helm chart deployment process.

Example 2-18 shows the template that generates the self-signed certificate during the Helm chart deployment.

Example 2-18 Secret definition template

```
apiVersion: v1
kind: Secret
metadata:
  name: "{{ template "fullname" . }}-tls-secret"
type: Opaque
data:
  {{ $ca := genCA "SampleApi CA" 3650 }}
  {{- $cn := .Values.service.ingresshost | quote }}
  {{- $altNames := list .Values.service.ingresshost ( printf "%s.%s" (include "fullname" .) .Release.Namespace ) ( printf "%s.%s.svc" (include "fullname" .) .Release.Namespace ) -}}
  {{- $cert := genSignedCert $cn nil $altNames 3650 $ca -}}
  tls-ca.crt: {{ print $ca.Cert | b64enc }}
  tls-ca.key: {{ print $ca.Key | b64enc }}
  tls.crt: {{ print $cert.Cert | b64enc }}
  tls.key: {{ print $cert.Key | b64enc }}
```

Attention: The `genSignedCert` function takes the following arguments:

- ▶ `cn`
- ▶ list of IP addresses
- ▶ list of alternative DNS names
- ▶ validity period in days
- ▶ CA certificate

You can skip some arguments by using 'nil' keyword, which is correctly handled by the function; however, the use of the `helm lint` command validation can complain about it.

¹ For more information, see: <https://kubernetes.io/docs/concepts/services-networking/ingress/>

Autoscaling definition

One of the foundational assumptions about an application's production readiness is that it must meet the requirements for availability. Although designing applications for high availability is out-of-scope for this book, one aspect of availability (the ability to handle dynamic demand) is particularly relevant to cloud native applications.

To address this requirement, Kubernetes provides scaling policies that allow applications to adapt to changing conditions. As traffic and workload increases for an application, scaling the application allows it to keep up with user demand. Running multiple instances of an application distributes the traffic to all of them. Additionally, after multiple instances of an application running are available, rolling updates can be performed without downtime.

To allow our sample application to scale, we define a HorizontalPodAutoscaler object that defines how scaling controller should react for changing conditions; in our case, an average CPU utilization (represented as a percentage of requested CPU) over all the pods.

Example 2-19 HorizontalPodAutoscaler template definition

```
{{ if .Values.hpa.enabled }}
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
  name: "{{ template "fullname" . }}-hpa-policy"
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1beta1
    kind: Deployment
    name: "{{ template "fullname" . }}-deployment"
  minReplicas: {{ .Values.hpa.minReplicas }}
  maxReplicas: {{ .Values.hpa.maxReplicas }}
  metrics:
  - type: Resource
    resource:
      name: cpu
      targetAverageUtilization: {{
.Values.hpa.metrics.cpu.targetAverageUtilization }}
  - type: Resource
    resource:
      name: memory
      targetAverageUtilization: {{
.Values.hpa.metrics.memory.targetAverageUtilization }}
{{ end }}
```

MongoDB component definition

The last element that is required by our sample application to run successfully is a MongoDB database. Without a working connection to the database, our Node.js app fails and Kubernetes keeps restarting the pods. The following options are available to satisfy the requirement for MongoDB:

- ▶ Install MongoDB as a part of the deployment, with or without a persistent volume.
- ▶ Reuse external MongoDB chart as requirement for our chart.
- ▶ Provide the user option to specify the connection parameters for external MongoDB service (directly or as existing ConfigMap object).

Next, we show how to include any of these options. To make it transparent to the rest of the chart, we must provide our Node.js app a consistent way to access connection information, no matter how the MongoDB gets provisioned. To achieve this, we use a ConfigMap object, as shown in Example 2-20 on page 68.

Example 2-20 ConfigMap template definition

```

{{- if not .Values.services.configMap }}
kind: ConfigMap
apiVersion: v1
metadata:
  name: {{ template "fullname" . }}-configmap
  namespace: {{ .Release.Namespace }}
  labels:
    chart: {{ template "chart" . }}
    app: {{ template "fullname" . }}
    release: "{{ .Release.Name }}"
    heritage: "{{ .Release.Service }}"
data:
  {{- if not .Values.services.mongo.install }}
  MONGO_HOST: {{ .Values.services.mongo.host }}
  MONGO_PORT: {{ .Values.services.mongo.port | default "27017" }}
  MONGO_DB_NAME: {{ .Values.services.mongo.dbname | default "TodoApp" }}
  MONGO_USER: {{ .Values.services.mongo.username }}
  MONGO_PASS: {{ .Values.services.mongo.password }}
  {{ else }}
  MONGO_HOST: {{ template "fullname" . }}-mongo-service
  MONGO_PORT: "27017"
  MONGO_DB_NAME: "TodoApp"
  MONGO_USER: null
  MONGO_PASS: null
  {{ end }}
{{- end }}

```

Installing MongoDB as a part of the deployment

For development purposes, it is convenient to include MongoDB as a part of a chart because it makes the app self-contained and allows for easy deployment in any namespace. To include MongoDB as a part of your chart, add MongoDB deployment and service templates.

The deployment of a local MongoDB instance is controlled with the `services.mongo.install` field in the `values.yaml` file.

Sample templates are shown in Example 2-21 and Example 2-22 on page 69.

Example 2-21 Definition of MongoDB local deployment definition template

```

{{- if and (not .Values.services.configMap) .Values.services.mongo.install }}
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: "{{ template "fullname" . }}-mongo-deployment"
spec:
  # this replicas value is default
  # modify it according to your case
  replicas: 1
  template:

```

```

metadata:
  labels:
    app: "{{ template "fullname" . }}-mongo-selector"
    service: mongo
spec:
  containers:
  - name: "{{ template "fullname" . }}-mongo-container"
    image: mongo:latest
    # resources:
    #   requests:
    #     cpu: 100m
    #     memory: 100Mi
{{-end}}

```

Example 2-22 Local MongoDB service definition template

```

{{- if and (not .Values.services.configMap) .Values.services.mongo.install }}
apiVersion: v1
kind: Service
metadata:
  name: "{{ template "fullname" . }}-mongo-service"
spec:
  # if your cluster supports it, uncomment the following to automatically create
  # an external load-balanced IP for the frontend service.
  #type: LoadBalancer
  type: ClusterIP
  ports:
    # the port that this service should serve on
    - name: http
      port: 27017
  selector:
    app: "{{ template "fullname" . }}-mongo-selector"
{{- end }}

```

Verifying Helm chart syntax with lint

For the YAML files, defining templates in a Helm chart formatting is crucial. Any indentation mistake generates an error during deployment. To help with the verification of the chart syntax, use the following command:

```
helm lint <chart name>
```

A sample output of the correct linting is shown in Example 2-23.

Example 2-23 Sample output of Helm verification with lint

```
helm lint ibm-nodejs-sample
```

```

==> Linting ibm-nodejs-sample
Lint OK

```

```
1 chart(s) linted, no failures
```

Another useful method is to generate a specific template file output by using the **helm template** command, as shown in Example 2-24 on page 70.

Example 2-24 Sample output of helm template command

```
helm template -x templates/service.yaml ibm-nodejs-sample
```

```
---
# Source: ibm-nodejs-sample/templates/service.yaml
apiVersion: v1
kind: Service
metadata:
  name: release-name-nodejssample-nodejs
  labels:
    app: "nodejsSample"
    chart: "ibm-nodejs-sample"
    heritage: "Tiller"
    release: "RELEASE-NAME"
    component: "nodejs"
  annotations:
    prometheus.io/scrape: 'true'
spec:
  type: NodePort
  ports:
    - port: 3000
  selector:
    app: nodejsSample
    release: RELEASE-NAME
    component: nodejs
```

2.3.3 Embedding MongoDB chart as prerequisite

While embedding a simple MongoDB is handy for development, it is not useful for serious testing and production deployments. However, rather than developing complex options for prerequisites, it is better to reuse sophisticated, production-tested Helm charts that are defined by the software vendor or a dedicated support team. Next, we discuss when and how to embed other charts as prerequisites for your chart.

Helm subcharts overview

Helm charts can have dependencies, which are called subcharts. These subcharts can have their own values and templates. There are multiple ways to manage dependencies between and share information across the charts. Careful consideration must be given to understand the global versus local scope of constructs, conditional enablement, and behavior when duplicate template defines are encountered across charts.

The following primary use cases are available for subcharts:

- ▶ Code packaged for modularity:
 - Modularity of charts, which allows multiple teams to code and test pieces of a deployment independently, yet the charts are often specialized for a particularly parent chart deployment
 - Generally, this drives the subchart source to be embedded into your chart structure as a source in the /charts directory
- ▶ Code packaged for ReUse (unchanged):
 - Inclusion of charts shared by many teams where the code is to be reused unchanged across parent charts

- Generally, this results in the subchart being a TGZ (TAR archive) that is packaged in the `/charts` directory

Helm dependencies are used to manage subchart inclusion and critical to understand when preparing your charts for integration with the content CICD build process and publication.

Generally, the **helm dependency update** command is used to verify that the required charts, as expressed in `requirements.yaml`, are present in `charts/` and are at an acceptable version. It pulls down the latest charts that satisfy the dependencies, and clean up old dependencies. On a successful update, this process generates a lock file that can be used to rebuild the requirements for an exact version.

Helm supports the following commands to manage dependencies that are expressed in `requirements.yaml`:

- ▶ **helm dependency build**: This command rebuilds the `charts/` directory based on the `requirements.lock` file.
- ▶ **helm dependency list**: This command lists the dependencies for the specific chart.
- ▶ **helm dependency update**: This command updates the `charts/` directory that is based on the contents of the `requirements.yaml` file.

The content build process expects dependencies to be managed before any push of the chart to a product team's individual or the overall consolidated charts git repository.

Helm chart dependencies

Regarding Helm chart dependencies, [Helm documentation](#) states:

In Helm, one chart may depend on any number of other charts. These dependencies can be dynamically linked through the `requirements.yaml` file or brought in to the `charts/` directory and managed manually.

Although manually managing your dependencies has a few advantages some teams need, the preferred method of declaring dependencies is by using a `requirements.yaml` file inside of your chart.

Specifying chart dependencies via `requirements.yaml`

The `requirements.yaml` file originally was an instruction to packaging (name, version, repository, and so on), but was extended to give direction during installation (enable/disable subchart inclusion based on conditions or tags and reference chart via alias).

The dual purpose of this file requires careful consideration, especially if mixing external dependency chart inclusion and conditional (or alias) logic related to internal subcharts. For example, if specifying an alias for an internal subchart, you might think that only requires chart name and alias when version + repository are required if pulling in external charts by using the **helm dependency update** command. An exact match must be found for the version that is specified in `requirements.yaml` or the alias are *not* established (without warning) and the **helm dependency update** command requires a repository for all specified entries.

Consider the following points:

- ▶ An alias is not two names for the same thing; instead, it is creating multiple instances of a specific chart.
- ▶ Condition logic is ignored if `yaml` paths that are specified do not exist in the parent's values or do not evaluate to a Boolean.
- ▶ Conditions set in `values.yaml` override tags.

- For a parent to access values of a subchart, must address directly and parameters cannot have a '-'; therefore, an alias is required for all charts with a '-' (which is a Helm standard).

A sample `requirements.yaml` file is shown in Example 2-25 on page 72.

Example 2-25 Sample content of requirements.yaml file

```
dependencies:
- name: mongodb
  version: 5.7.0
  repository: https://kubernetes-charts.storage.googleapis.com
  alias: mongo
```

To download subcharts and verify a `requirements.yaml` file, run the commands that are in Example 2-26.

Example 2-26 Sample output of helm dependency update command

helm dependency update

```
Hang tight while we grab the latest from your chart repositories...
...Unable to get an update from the "local" chart repository
(http://127.0.0.1:8879/charts):
  Get http://127.0.0.1:8879/charts/index.yaml: dial tcp 127.0.0.1:8879:
  getsockopt: connection refused
...Successfully got an update from the "stable" chart repository
Update Complete. ?Happy Helming!?
Saving 1 charts
Downloading mongodb from repo https://kubernetes-charts.storage.googleapis.com
Deleting outdated charts
```

Tip: Run the `helm init` command (as shown in Example 2-27) if you receive an error as shown in the following example:

Error: Couldn't load repositories file (.helm/repository/repositories.yaml)

Example 2-27 Sample output of helm init command

helm init --client-only

```
Creating /Users/guest/.helm/repository
Creating /Users/guest/.helm/repository/cache
Creating /Users/guest/.helm/repository/local
Creating /Users/guest/.helm/plugins
Creating /Users/guest/.helm/starters
Creating /Users/guest/.helm/repository/repositories.yaml
Adding stable repo with URL: https://kubernetes-charts.storage.googleapis.com
Adding local repo with URL: http://127.0.0.1:8879/charts
$HELM_HOME has been configured at /Users/guest/.helm.
Not installing Tiller due to 'client-only' flag having been set
Happy Helming!
```

The MongoDB Helm chart from the `@stable` repository (which is an alias for `https://kubernetes-charts.storage.googleapis.com`) defines its own sets of resources (for example, `mongodb` service). Therefore, to effectively use it, we must add `mongodb`-specific values (for subchart) to our `values.yaml` and adjust our `configmap` template.

Values that are under the name of the subchart are sent to subchart during deployment, which overrides subchart defaults.

Example 2-28 shows how such a section of the `values.yaml` file might look.

Example 2-28 Section of values.yaml that will be sent to mongodb subchart

```
mongodb:
  mongodbUsername: myusername
  mongodbPassword: mypassword
  mongodbDatabase: mydatabase
```

2.3.4 Testing Helm charts

As a recommended practice, every Helm chart should include a tests definition that can be run automatically to verify the Helm chart releases.

The Helm tests provide a level of validation that a chart is installed correctly and services are available. They are defined in the chart under the `templates/tests` directory by convention and consist of a pod definition that includes the following components:

- ▶ A hook annotation (`helm.sh/hooks: test-success` or `helm.sh/hooks: test-failure`)
- ▶ Container image
- ▶ Command to use for installation verification

Commonly, the container image that is used for testing is lightweight and publicly available with just enough functionality that is installed to run the verification commands. However, cases exist in which a more heavyweight image (such as another instance of the application) is required.

Independent of type of image, all tests pods must conditionally allow specification of an `imagePullSecret` or use a service account that contains `imagePullSecret` to access the internal IBM Cloud Private registry in isolated installations. In addition, test pods require affinity to ensure lands on architecture supported by image in a hybrid cluster (multi-architecture).

For more information about Helm tests and examples, see the [Helm documentation](#).

2.3.5 Upgrading Helm charts

As your application evolves, the application images and Helm chart changes versions. Helm provides a convenient way to upgrade existing releases with new content. The same command is used for upgrading to new version of Helm chart (for example, to define new Kubernetes resources) and to update the resources with new values (without changing the Helm chart version).

To upgrade the release, run the following command:

```
helm upgrade <release> <chart>
```

Helm upgrade can take new values that are specified with `-f <new_values.yaml>` or using `--set key=value` option. You must specify only the new values to be used for the upgrade. All of the previous values that are used before in release are carried forward. For example, run the following command to update only the container image tag that is used by the chart:

```
helm upgrade <release> <chart> --set image.tag=v1.0.1
```

To avoid this behavior and reset all the values to default values that are included in the chart, add `--reset-values`.

If any of the values change, the pod specification template within deployment definition (for example, the image tag as shown in our example), triggers creating a `ReplicaSet`. If the update strategy that is specified in deployment is `RollingUpdate`, the deployment controller gradually scales up new `ReplicaSet` and scales down old `ReplicaSet`. This process ensures that the required number of application pods is available in any point. As the result, the application should be upgraded in place, without a downtime that is visible for users. An alternative update strategy is to `Recreate`. In that case, the old `ReplicaSet` and pods are deleted and a new one is created.

Istio allows for some other application update strategies (for example, Blue/Green or Canary deployment) as described in Chapter 4, “Managing your service mesh by using Istio” on page 97.

2.4 IBM Cloud Paks

In these sections, we briefly introduce the concept of IBM Cloud Paks, which is an enterprise ready packaging of cloud native solutions.

2.4.1 What is an IBM Cloud Pak?

IBM Cloud Paks provide enterprise software container images that are pre-packaged in production-ready configurations. These configurations can be quickly and easily deployed to IBM's container platforms. They include support for resiliency, scalability, and integration with core platform services, such as monitoring or identity management.

2.4.2 Developing Helm charts for IBM Cloud Private

IBM provides a Helm chart repository that is available at [GitHub](#) that contains a stable repo with publicly available versions of many IBM software products. IBM also encourages others to publish their software charts in the community section of the repo.

To help build quality Helm charts, the repository includes detailed recommendations about how to build a proper Helm chart for IBM Cloud Private environments. Because document is a living document, we do not copy its content in this book; instead, see [this web page](#).

For more information about CloudPak certification for Helm charts, see [this web page](#).

Using IBM Cloud Private Catalog features (values-metadata)

IBM Cloud Private catalog provides some other features to make the Helm charts easier to use. One of these features is formatting and validating the values that the chart makes available for users to customize.

You can define metadata for the parameters in `values-metadata.yaml` file that is placed in root directory of the chart (same as `values.yaml`). The file `values-metadata.yaml` defines the metadata for values defined in `values.yaml`. It should mirror the same structure of `values.yaml`, except that instead of specifying a value for the leaf property, define a property named `__metadata` (see Example 2-29 on page 75).

If the `values-metadata.yaml` file is not present, the UI continues to display all parameters that are declared in `values.yaml` based on the type inference.

Example 2-29 Sample content of values-metadata.yaml

```
demonstration:
  __metadata:
    label: Demonstration
    description: I am an h2
stringField:
  __metadata:
    label: String field
    type: string
    required: true
numberField:
  __metadata:
    label: Number field (with validation)
    type: number
    required: true
checkboxField:
  __metadata:
    label: Checkbox field
    type: boolean
    required: true
selectField:
  __metadata:
    label: Select field
    type: string
    required: true
    options:
      - label: myOpt1
        value: myNotSelectedValue
      - label: myOpt2
        value: mySelectedValue
multilineField:
  __metadata:
    label: Multiline field
    type: string
    multiline: true
    required: true
immutableField:
  __metadata:
    label: Immutable field
    type: string
    required: true
    immutable: true
arrayField:
  __metadata: ### arrayField: "[]" or [] must be set in values.yaml (see below
for details).
    label: Array field
    type: string ### do to backwards-compat bug, type must be string
    description: I am an array.
    required: true
objectField: ### objectField: "{}" or {} must be set in values.yaml (see below
for details)
  __metadata:
```

```
label: Object field (new)
type: string   ### do to backwards-compat bug, type must be string
description: I am an object.
required: true
```

The following metadata fields are available:

description	Description of the parameter. Should appear in tooltip or, if group description, as subheader.
hidden	If hidden = true, the element is hidden (does the same thing as immutable, but the form field is never exposed in the UI).
immutable	If immutable = true, the user cannot modify the parameter (field = disabled).
label	Title of the parameter. If label is not specified, the key from values.yaml is used.
multiline	If type = string, display a text area field.
options	Describes a parameter that transforms into a drop-down menu.
required	Describes if the parameter is required. If so, an asterisk (*) is displayed next to the name in the UI.
type	The type of the parameter (string, boolean, number or password; array and object are also type: string bc of historical backwards-compatibility issues).
validation	Regular expression (JavaScript regex) to validate parameter value.



DevOps and application automation

This chapter introduces DevOps and application automation. It also covers several tooling options to enable DevOps with IBM Cloud Private.

The chapter includes the following topics:

- ▶ 3.1, “DevOps overview” on page 78
- ▶ 3.2, “DevOps tooling options” on page 82
- ▶ 3.3, “Introduction to Microclimate” on page 84
- ▶ 3.4, “Sample DevOps scenario with Microclimate” on page 84

3.1 DevOps overview

DevOps is short for development and operations. To speed the release of new applications and updates, the IT industry works to apply agile and lean principles to development and deployment. At the core of these principles is eliminating wasted effort, breaking down artificial barriers between related functional teams, and adopting continuous release cycles that push improvements out to users faster than ever before. This approach allows the business to more quickly seize market opportunities and reduce the time to include customer feedback.

A DevOps approach can be successful when dealing with the complexities of enterprise applications that involve multiple products and technologies.

3.1.1 Benefits of DevOps

In a DevOps shop, the business owner, developers, and operations team work cooperatively. The plan-develop-build-deploy-feedback process is continuous (see Figure 3-1). When possible, human actions are minimized in favor of *autonomous tools*, including software builds that start on their own when eligible code updates become available. Applications reach their full potential faster, often by using fewer resources than in the past.

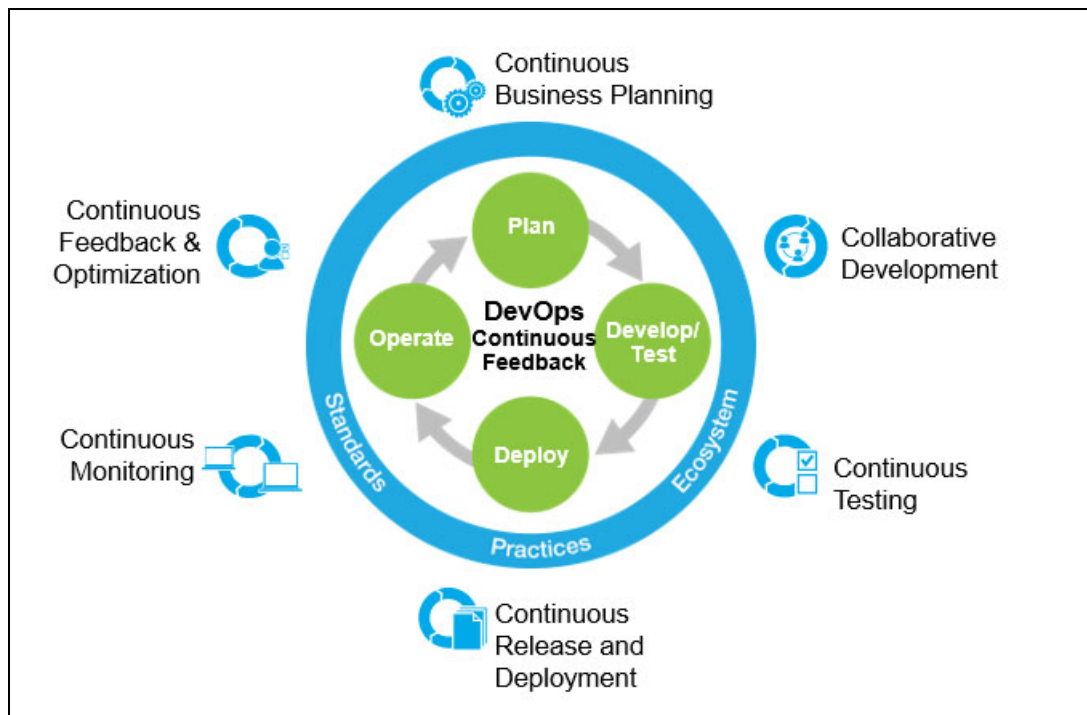


Figure 3-1 DevOps overview

3.1.2 Continuous business planning

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

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

Continuous business planning has the following benefits:

- ▶ Helps you continuously plan business needs and prioritize them
- ▶ Integrates customer feedback with business strategy
- ▶ Aligns customer feedback into the development lifecycle as business needs
- ▶ Focuses on *doing the correct things*
- ▶ Prioritizes business needs that add the most value

3.1.3 Continuous integration and collaborative development

Continuous integration refers to the leading practice of integrating the code of the entire team regularly to verify that it works well together. Continuous integration is a software development practice that requires team members to integrate their work frequently.

Integrations are verified by an automated build that runs regression tests to detect integration errors as quickly as possible. Teams find that this approach leads to significantly fewer integration problems, and enables development of cohesive software more rapidly.

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

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

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

Collaborative development has the following benefits:

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

3.1.4 Continuous testing

Continuous testing eliminates testing bottlenecks and simplifies the creation of virtualized test environments that can be easily deployed, shared, and updated as systems change. The use of these capabilities can reduce the cost of provisioning and maintaining test environments, and shorten test cycle times by enabling integration testing earlier in the lifecycle.

Continuous testing has the following benefits:

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

3.1.5 Continuous release and deployment

Continuous delivery is a set of practices that enables the code to be rapidly and safely deployed to production by delivering changes to a production-like environment. It also ensures that business applications function as expected through exhaustive automated testing. Because all changes are delivered to a staging environment by using complete automation, your applications can be deployed to production with the “push of a button” when the business is ready.

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

Continuous delivery has the following benefits:

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

3.1.6 Continuous monitoring

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

In production, the operations team manages and ensures that the application is performing as wanted, and that the environment is stable by using continuous monitoring. The Ops teams have their own tools for monitoring their environments and systems. The DevOps principles go beyond suggesting that they should also monitor the applications to ensure that the applications are performing at optimal levels.

This requires that Ops teams use tools for monitoring application performance and issues. In addition, it might require that they work with Dev to build self-monitoring or analytics gathering capabilities that are directly built into the applications. This allows for true end-to-end and continuous monitoring.

Continuous monitoring has the following benefits:

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

3.1.7 Continuous customer feedback and optimization

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

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

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

Continuous customer feedback and optimization provides the following benefits:

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

The DevOps approach is multi-platform. The principles are the same whether the application runs on a mainframe, in a distributed environment, or in the cloud. It can be applied to any development effort, from enterprise applications, such as back-end banking systems, to more customer-facing products, such as smartphone tools. It also is customer-neutral. Whether you are working with an external client or an internal line of business, DevOps can help you meet your commitments faster and with better results.

For more information about DevOps, see [*DevOps for Dummies \(IBM Edition\)*](#).

3.2 DevOps tooling options

This section provides various tooling options to enable DevOps with IBM Cloud Private.

3.2.1 Code Editors tooling options

Table 3-1 list the various code editor options that can be used by the developers.

Table 3-1 Code Editor tooling options

Tool	On-premises hosted	IBM Cloud Private
Visual Studio Code	On Developers Machine.	Integrate with Microclimate hosted on IBM Cloud Private.
Eclipse	On Developers Machine.	Integrate with Microclimate hosted on IBM Cloud Private.
Microclimate - Theia	Microclimate hosted outside IBM Cloud Private. Accessed as Online editor.	Microclimate hosted on IBM Cloud Private. Accessed as Online editor.

3.2.2 Microservice builder tooling options

Microclimate is a tool for building microservices. Microclimate is a dockerized, end-to-end development environment that enables agile development and delivery of microservices, hybrid, and Docker containerized apps in Java, Node.js, and Swift.

Microclimate offers services and tools to help you create and modernize applications in one seamless experience. You can use Microclimate for every step of the process, from writing and testing code locally to building and deployment with a pipeline. Microclimate can be hosted on IBM Cloud Private or as a stand-alone application. For more information about Microclimate, see the following resources:

- ▶ “Microservice builder tooling options” on page 82
- ▶ [Microclimate documentation](#)

3.2.3 Source code management tooling options

Table 3-2 list the various source code tooling options that can be used by the developers.

Table 3-2 Source code options

Tool	On-premises hosted	IBM Cloud Private
GitHub	Hosted on-premises	Pipelines in IBM Cloud Private can integrate GitHub hosted externally to IBM Cloud Private
Gitlab	Hosted on-premises	Hosting and Integration
IBM Rational® Team IBM Concert®	Hosted on-premises	Pipelines in IBM Cloud Private can integrate IBM Rational Team Concert™ external to IBM Cloud Private

3.2.4 Build, Test, and Continuous Integration tools

Table 3-3 list the various CI/CD tooling options that can be used by developers.

Table 3-3 CI/CD tooling options

Tool	On-premises	IBM Cloud Private
Jenkins	Hosted	Hosted
Urban Code Deploy	Hosted	Hosted and Integration
Sauce Labs	Enterprise	Integration via Jenkins

Jenkins

Jenkins is an open source continuous integration platform that runs under a Java web servlet container (such as the Liberty profile server).

Continuous integration is the practice of frequently building and testing software projects during development. The aim of this process is to discover defects and regressions early by automating the process of running unit and integration tests. These automated build and test cycles typically happen on a regular schedule (such as every night) or even after each change is delivered.

Jenkins can integrate with a large variety of frameworks and toolkits by using its extensive library of available plug-ins, including the following items:

- ▶ Source code management and version control platforms, including CVS, Subversion, Git, Mercurial, Perforce, IBM ClearCase®, and IBM Rational Team Concert.
- ▶ Build automation tools, such as Apache Ant and Maven, and standard operating system batch and script files.
- ▶ Testing frameworks, such as JUnit and TestNG.
- ▶ RSS, email, and instant messenger clients for reporting results in real time.
- ▶ Artifact uploaders and deployers for several integration platforms.

IBM UrbanCode Deploy

IBM UrbanCode® Deploy is a tool for automating application deployments through your environments. It is designed to facilitate rapid feedback and continuous delivery in agile development while providing the audit trails, versioning, and approvals that are needed in production.

IBM UrbanCode Deploy provides the following benefits:

- ▶ Automated, consistent deployments and rollbacks of applications
- ▶ Automated provisioning, updating, and de-provisioning of cloud environments
- ▶ Orchestration of changes across servers, tiers, and components
- ▶ Configuration and security differences across environments
- ▶ Clear visibility: What is deployed, where, and who changed what
- ▶ Integrated with middleware, provisioning, and service virtualization

Sauce Labs

When a Sauce Labs test suite is configured as a test job in a pipeline, the test suite can run functional and unit tests against your web or mobile app as part of your continuous delivery process. These tests can provide valuable flow control for your projects, acting as gates to prevent the deployment of bad code.

Functional testing emulates what users do when they interact with your website. Functional testing is often confused with unit testing, which tests the underlying code to ensure that it works as prescribed.

You can use Sauce Labs to automate functional testing on multiple operating systems and browsers, emulating the way that a user uses the website. With Sauce Labs, you can also run tests on various operating system and browser combinations in parallel, which reduces the amount of time to get results. Sauce Labs can be integrated with Jenkins for testing.

For more information about Sauce Labs, see [this web page](#).

3.3 Introduction to Microclimate

Microclimate is a dockerized, end-to-end development environment that enables agile development and delivery of microservices, hybrid, and Docker containerized apps in Java, Node.js, Swift, and more. Microclimate offers services and tools to help you create and modernize applications in one seamless experience. You can use Microclimate for every step of the process, from writing and testing code locally to building and deployment with a pipeline.

Teams are increasingly turning to continuous delivery, microservices, DevOps, and containers as the foundation for application architectures to enable faster innovation and business agility. To achieve agility and stability, use a microservices architecture to develop and deliver modern, lightweight, and composable workloads across public, private, and hybrid application environments.

For those users who want to maximize the pace and benefits of continuous delivery, Microclimate serves as the enabling technology for the entire software delivery lifecycle.

Microclimate documentation: For more information about the Microclimate documentation, see [this web page](#).

3.4 Sample DevOps scenario with Microclimate

In this section, we provide a step-by-step Microclimate usage example, starting from the installation, building an application with Microclimate, and deploying the application through Microclimate pipelines.

3.4.1 Installing Microclimate on IBM Cloud Private

You can install Microclimate on IBM Cloud Private by using the [IBM Helm chart](#).

Next, we describe how to install Microclimate on IBM Cloud Private.

Tip: Always refer to the latest Helm chart to get the most recent version of Microclimate.

Logging in to IBM Cloud Private with `cloudctl`

Log in to the IBM Cloud Private by using the `cloudctl` command so that the configuration for `kubectl` and `helm` are set up properly. You use some of the files that are configured for Helm during the Microclimate installation later.

Namespace setup for Microclimate

Create two namespaces in IBM Cloud Private: one for hosting the Microclimate application, (for example mc-demo), the other for the target namespace for the application that is being developed (for example mc-deploy).

Create a cluster image policy to allow the Microclimate docker images to run by preparing the following yaml file as shown in Example 3-1. Save it as image.policy.yaml.

Example 3-1 Cluster image policy for Microclimate

```
apiVersion: securityenforcement.admission.cloud.ibm.com/v1beta1
kind: ClusterImagePolicy
metadata:
  name: microclimate-image-policy
spec:
  repositories:
    - name: docker.io/maven:*
    - name: docker.io/jenkins/*
    - name: docker.io/docker:*
    - name: docker.io/ibmcom/*
```

Apply the yaml file by using the `kubectl apply -f image.policy.yaml` command.

Docker registry secret and Helm secret

To push the Docker image to the private registry after the build, you must specify the Docker registry credentials in a Kubernetes secret. Complete the following steps:

1. Determine the fully qualified name of the private registry. You can refer to the `cluster_ca_domain` field in your installation config.yaml file. You also can find it by running the command to display the cluster information that is shown in Example 3-2.

Example 3-2 Command to list cluster info

```
kubectl -n kube-public get cm ibmcloud-cluster-info -o yaml
```

```
apiVersion: v1
data:
  cluster_address: rb-icp312-ee-ubuntu-xxxxxxx.bluemix.net
  cluster_ca_domain: rb-icp312-ee-ubuntu-xxxxxxx.bluemix.net
  cluster_kube_apiserver_port: "8001"
  cluster_name: rb-icp312-ee-ubuntu
  cluster_router_http_port: "8080"
  cluster_router_https_port: "8443"
  edition: Enterprise Edition
  proxy_address: rb-icp312-ee-ubuntu-proxy-xxxxxxx.bluemix.net
  proxy_ingress_http_port: "80"
  proxy_ingress_https_port: "443"
...
```

2. Create the Docker registry secret for your namespace mc-demo and the mc-deploy namespace with the fixed name of microclimate-registry-secret, as shown in Example 3-3.

Example 3-3 Create docker registry secret

```
kubectl -n mc-demo create secret docker-registry microclimate-registry-secret
--docker-server={{ .cluster_ca_domain }}:8500 --docker-username={{ .user }}
--docker-password={{ .password }} --docker-email={{ .email }}

kubectl -n mc-deploy create secret docker-registry microclimate-registry-secret
--docker-server={{ .cluster_ca_domain }}:8500 --docker-username={{ .user }}
--docker-password={{ .password }} --docker-email={{ .email }}
```

3. Replace {{ .cluster_ca_domain }} with the findings of cluster_ca_domain. Replace the {{ .user }}, {{ .password }}, and {{ .email }} with the user name, password, and email address (can be a dummy email address) of the account you used to log in to IBM Cloud Private.
4. Patch the default service account of the namespace mc-deploy to use this registry credential for pulling the image from the private registry. Run the following command:

```
kubectl patch serviceaccount default -n mc-deploy -p "{\"imagePullSecrets\": [{\"name\": \"microclimate-pipeline-secret\"}]}"
```
5. Create a Kubernetes secret for the Microclimate by using the following command:

```
kubectl -n mc-demo create secret generic microclimate-helm-secret
--from-file=cert.pem=~/.helm/cert.pem --from-file=ca.pem=~/.helm/ca.pem
--from-file=key.pem=~/.helm/key.pem
```

We are using the certificates and certificate authority files that are created properly after a successful `cloudctl` login.

Installing the Helm chart

Perform the following steps to install the Helm chart:

1. Prepare the variable overrides yaml file as shown in Example 3-4.

Example 3-4 Variable overrides for Microclimate chart

```
persistence:
  enabled: true
  useDynamicProvisioning: true
  size: 8Gi
  storageClassName: nfs-client

global:
  ingressDomain: mc.{{ .yourProxyIp }}.xip.io

jenkins:
  Pipeline:
    TargetNamespace: mc-deploy
    Registry:
      Url: rb-icp312-ee-ubuntu-xxxxxxx.bluemix.net:8500
  Persistence:
    StorageClass: nfs-client
```

2. Enable the Microclimate persistency, which is used for storing the Microclimate project workspaces. Set `useDynamicProvisioning` as true and specify the `storageClassName` you want to use in your IBM Cloud Private environment.

Tips: Consider the following tips:

- ▶ The storage access mode that is required is *ReadWriteMany (RWX)* for the project workspace. Not all storage classes can provide the RWX capability. NFS, GlusterFS, and CephFS volumes can provide the RWX access. The Ceph Rados Block Device (RBD), enabled by [Rook](#), cannot fulfill this requirement. Here, we use the NFS dynamic storage class that is enabled by the [NFS client provisioner](#).
- ▶ The persistence volume must be turned on for Jenkins. The storage mode that is required is *ReadWriteOnce (RWO)*, which is supported by most of the persistent volumes.
- ▶ The `global.ingressDomain` field defines how you access Microclimate through Ingress. Assuming that you have internet access, you can use the `xip.io` to resolve the host name to the proxy of your cluster. Otherwise, you must register this `ingressDomain` name with the DNS server in your environment.
- ▶ In the `jenkins.pipeline.Registry.Url`, define your private registry URL by using `cluster_ca_domain`.

3. Save the yaml file as `values.override.yaml`. Install the Helm chart by using the command that is shown in Example 3-5.

Example 3-5 Helm command to install Microclimate

```
helm install --name mclimate --namespace mc-demo -f values.override.yaml
ibm/ibm-microclimate --tls
```

4. Ensure that all the pods are running, as shown in Example 3-6.

Example 3-6 Validate pods are running

```
kubectl -n mc-demo get pods
```

NAME	READY	STATUS
RESTARTS AGE		
jenkins-slave-1lxl1-bzd1w	5/5	Running 0
41s		
mclimate-ibm-microclimate-7bbd887b4c-z4lkq	1/1	Running 0
3m12s		
mclimate-ibm-microclimate-atrium-7d48b45b6c-nxrsl	1/1	Running 0
3m12s		
mclimate-ibm-microclimate-devops-7987cdffb6-c8gqm	1/1	Running 0
3m12s		
mclimate-jenkins-5bc78b8549-pp87p	1/1	Running 0
3m12s		

You now can access Microclimate `https://microclimate.mc.{ .yourProxyIp }.xip.io` and Jenkins `https://jenkins.mc.{ .yourProxyIp }.xip.io`.

5. Log in with your account in IBM Cloud Private and dismiss the introduction pages. You land on the page that is shown in Figure 3-2.

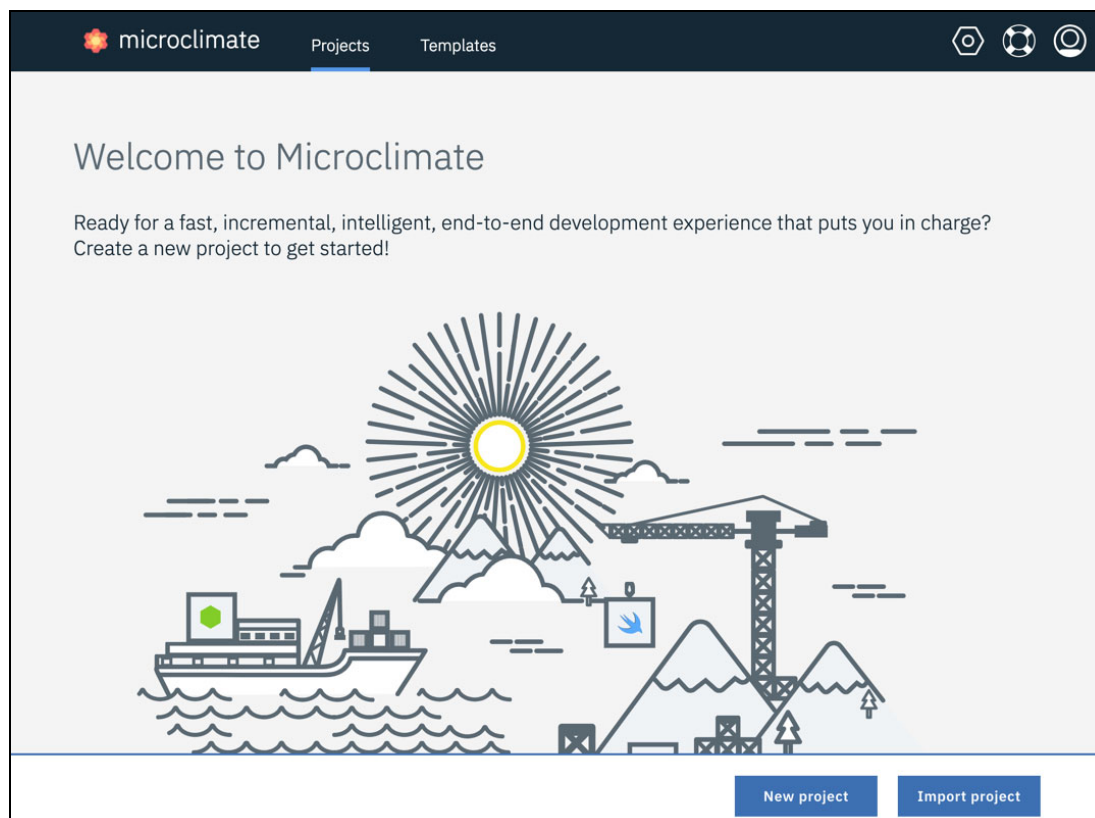


Figure 3-2 Microclimate welcome page

You are now ready to start a project with Microclimate.

3.4.2 Creating a hello world app by using Microclimate

Complete the following steps to create a hello world app by using Microclimate:

1. On the welcome page, click **New Project**. On the next page, select your language as **Go**.
2. Name your project; for example, `mcgosample`. Click **Create project** in GitHub and enter your GitHub access token, as shown in Figure 3-3 on page 89.

You can define the access token on [github.com](https://github.com/settings/tokens) by clicking **GitHub Settings** → **Developer settings** → **Personal access tokens** with the correct permissions.

The screenshot shows a web interface for creating a new project. At the top, there is a dark blue header with the 'microclimate' logo and navigation links for 'Projects' and 'Templates'. On the right side of the header are three icons: a hexagon, a network diagram, and a circular arrow. The main content area is titled 'New project' and contains a section 'Name your project'. This section includes several input fields: 'Project name' with the value 'mcgosample', 'GitHub URL' with the value 'github.com', 'Organization name' (empty), 'Description' with the value 'Microclimate Sample App', and 'Access token' (masked with dots). A checkbox labeled 'Create project in GitHub' is checked. Below the 'Organization name' field, there is a link that says 'Repository to be created in an organization'. At the bottom of the form, there are three buttons: 'Cancel', 'Previous', and 'Next'.

Figure 3-3 Creating a project on github.com

3. Click **Next** to continue. Select the project type as Go sample template. Then, click **Create**. You see that the project is created, as shown in Figure 3-4 on page 90.

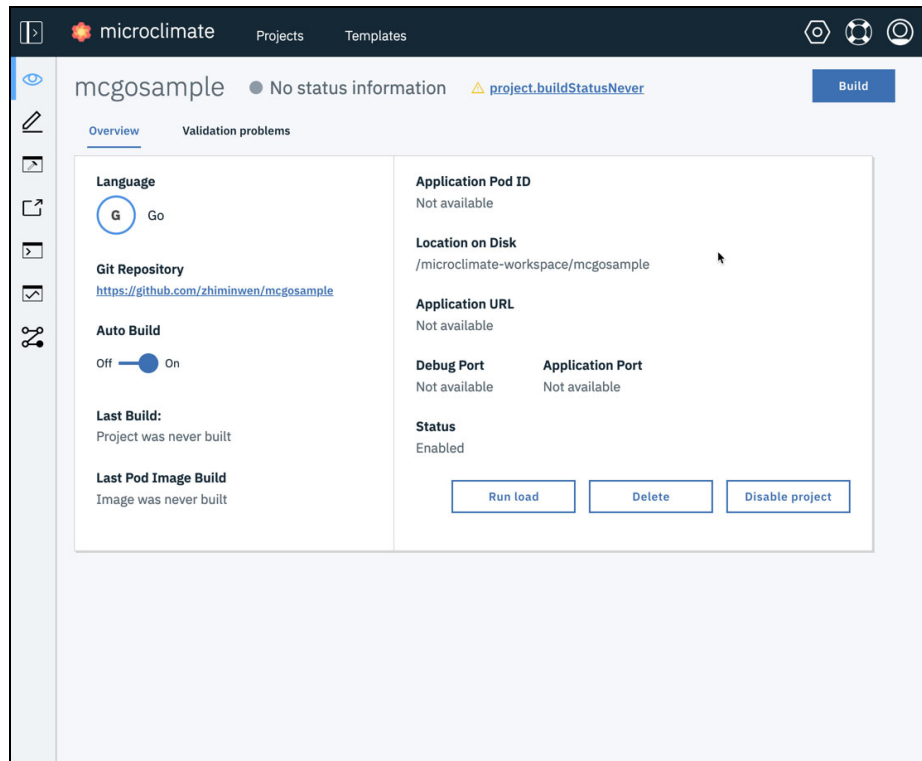


Figure 3-4 Summary page of the project

4. Go to GitHub to browse what was auto-generated for you, as shown in Figure 3-5.

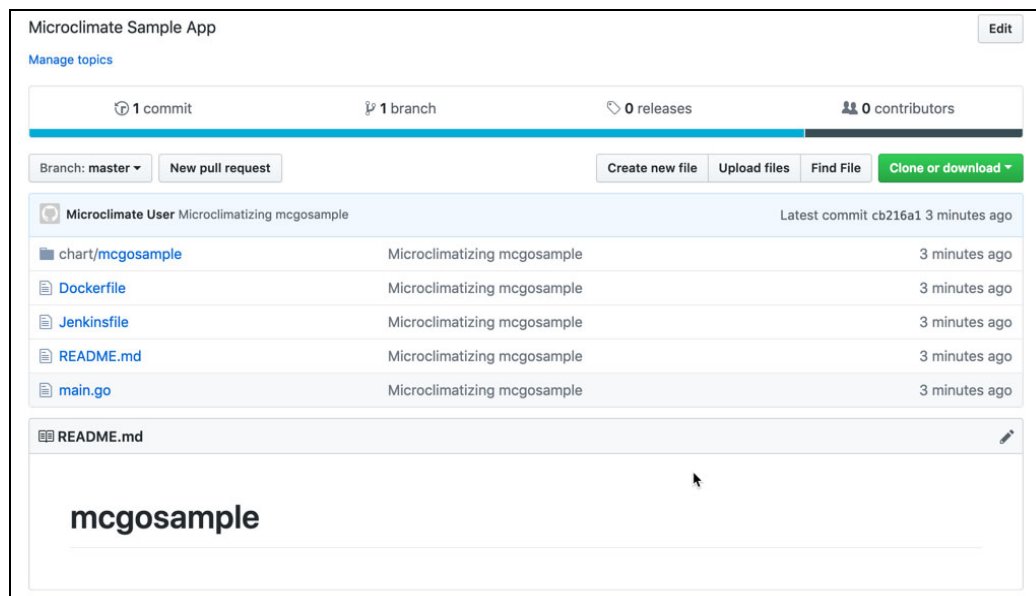


Figure 3-5 Files created on GitHub

You now have a Dockerfile, a Jenkinsfile, and a Helm chart that are created and ready to use.

5. To build the application, lick the **Build** button if the building process is not started automatically.

- Click **Build logs** on the left vertical bar to check the build logs, as shown in Figure 3-6.

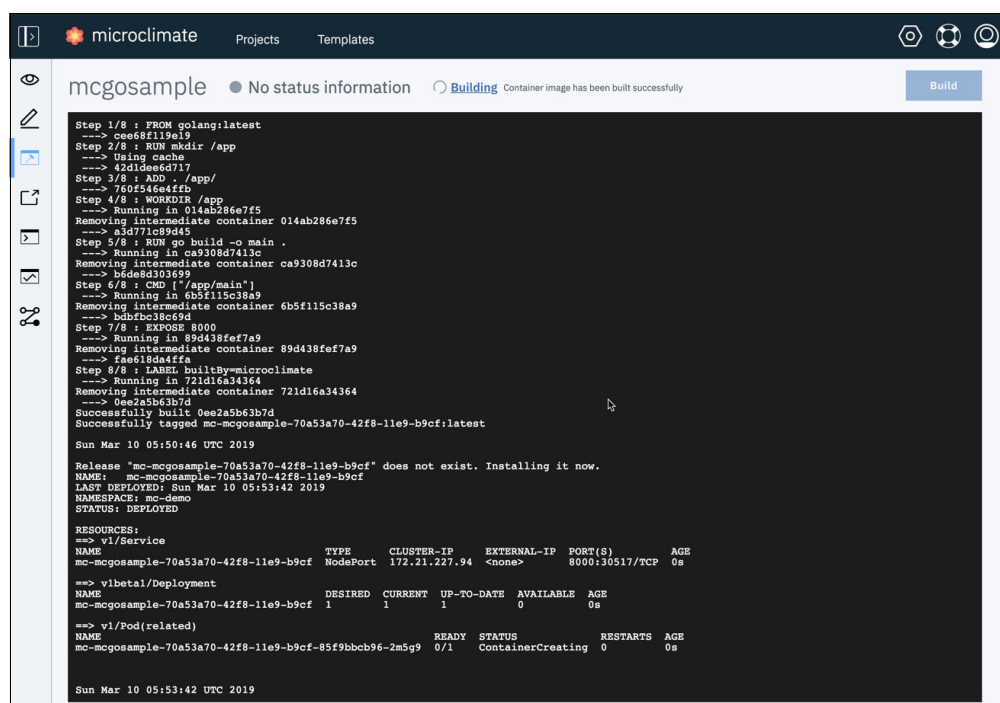


Figure 3-6 Build logs

- Now, validate the pods under the mc-demo namespace. More Microclimate application pods are running and the build application is also running, as shown in Example 3-7.

Example 3-7 Pods running under mc-demo namespace

```
kubect1 -n mc-demo get pods
```

NAME	READY	STATUS
mc-mcgosample-70a53a70-42f8-11e9-b9cf-85f9bbcb96-2m5g9	1/1	Running 0
mclimate-ibm-microclimate-7bbd887b4c-tx1j2	1/1	Running 0
mclimate-ibm-microclimate-admin-editor-66bfc5674-szqmk	2/2	Running 0
mclimate-ibm-microclimate-admin-filewatcher-bbdf7ddd7-87txh	1/1	Running 0
mclimate-ibm-microclimate-admin-loadrunner-5c866c6757-8vmxm	1/1	Running 0
mclimate-ibm-microclimate-atrium-7d48b45b6c-9fdcj	1/1	Running 0
mclimate-ibm-microclimate-devops-7987cdffb6-9dxdm	1/1	Running 0
mclimate-jenkins-5bc78b8549-pn2bf	1/1	Running 0

The build is successful, as shown in Figure 3-7.

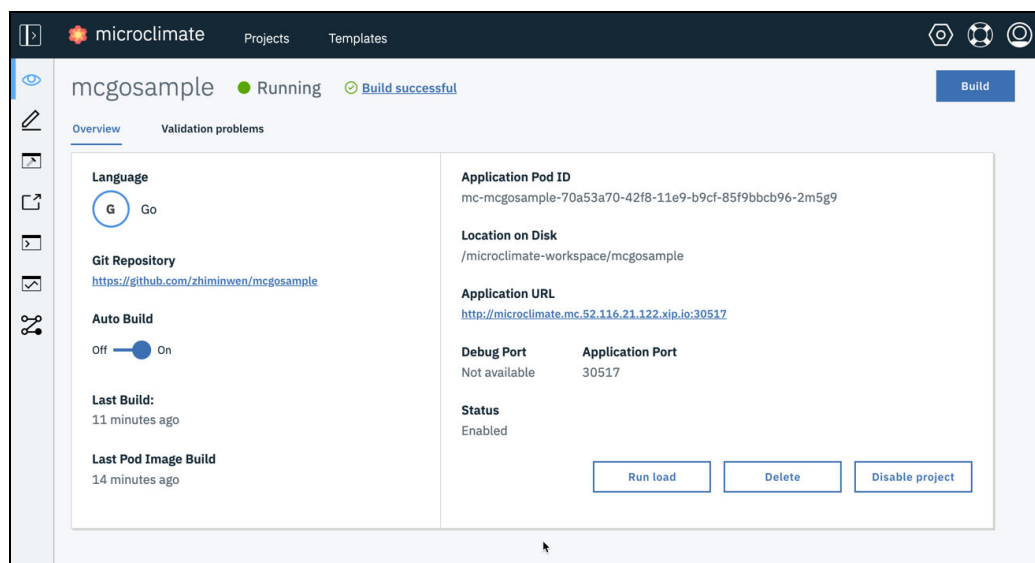


Figure 3-7 Build success

8. Click the application URL. You should see the greeting from the application as shown in the following example:

Hello from your Go sample application running in Microclimate!

3.4.3 Creating build and deployment pipelines

Now, we create build and deployment pipelines so that when a source code update is pushed, the changes are automatically deployed.

Complete the following steps:

1. Click the **Pipeline** icon on the bottom of the left vertical bar. Then, click **Create pipeline**.
2. Name the pipeline as build.
3. Click the **Credentials** drop-down list.
4. Click **Add new**.
5. Add your GitHub personal token, as shown in Figure 3-8 on page 93.

Credentials

Name

github-token

☐ User name and password

User name Password

☒ Personal access token

Cancel Save

Figure 3-8 Creating GitHub credentials

6. Select the credential that was just created. Click **Create pipeline** at the bottom of the page. The result is shown in Figure 3-9.

microclimate Projects Templates

mcgosample

build

Git repository

https://github.com/zhiminwen/mcgosample.git

Credentials

github-token Clear

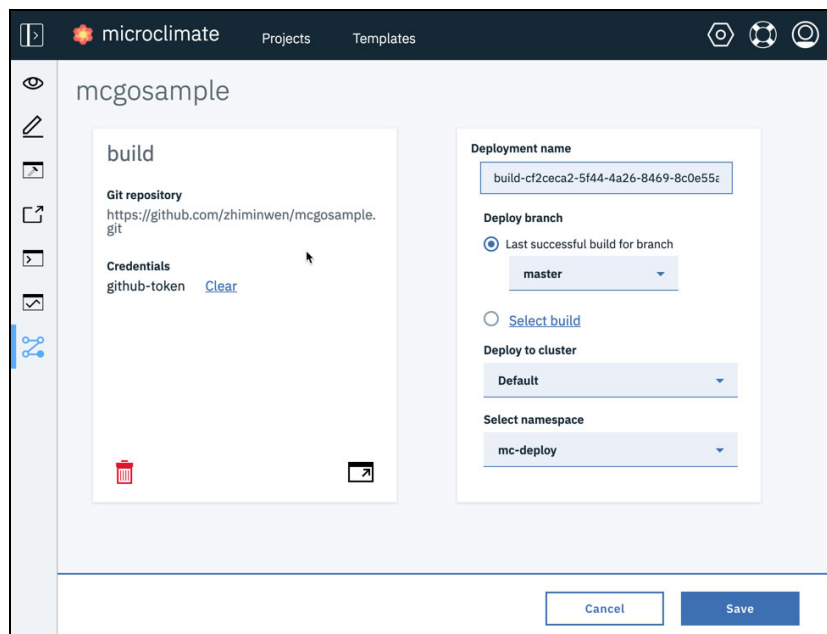
Add deployment

Open pipeline Add deployment

Figure 3-9 Creation of Build pipeline

7. Browse to github.com. Select **<your repository>** → **Settings** → **Webhooks**. You see a display that is similar to the example that is shown in Figure 3-10.

Figure 3-10 Webhook on github.com



10. Click **Save** to complete the pipeline.

Now, you are ready to test the pipeline to observe the automatic build and deployment whenever a new code checks in.

Tip: IBM Cloud Private tightens the security control of the namespace. *By default, the most restricted pod security policy (PSP) is used.* Therefore, any container that uses the root account cannot run. For this reason, a non-root account is used in our example.

Complete the following steps:

1. Update the Dockerfile in the repository, as shown in Example 3-8. As you can see, we are use a non-root account.

Example 3-8 Dockerfile with non-root

```
FROM golang:latest
RUN mkdir /app && useradd -m -u 1000 app && chown app:app /app
USER 1000
ADD . /app/
WORKDIR /app
RUN go build -o main .
CMD ["/app/main"]
EXPOSE 8000
```

2. Commit the changes and push to the GitHub repository. Start the Jenkins console with the URL `https://jenkins.mc.{ .yourProxyIp }.xip.io`.

You should see that the pipeline is started automatically, as shown in Figure 3-12.

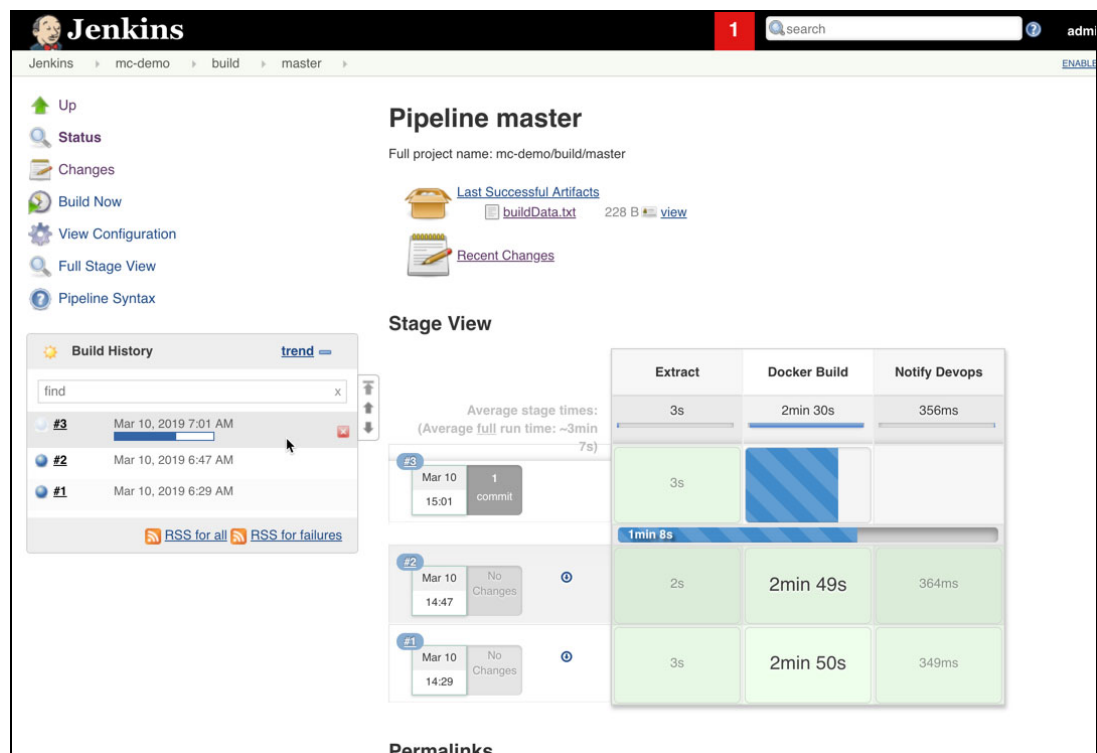


Figure 3-12 Jenkins pipeline trigger

After the Jenkins build success, you should see that the Microclimate deployment is successful, as shown in Figure 3-13 on page 96.

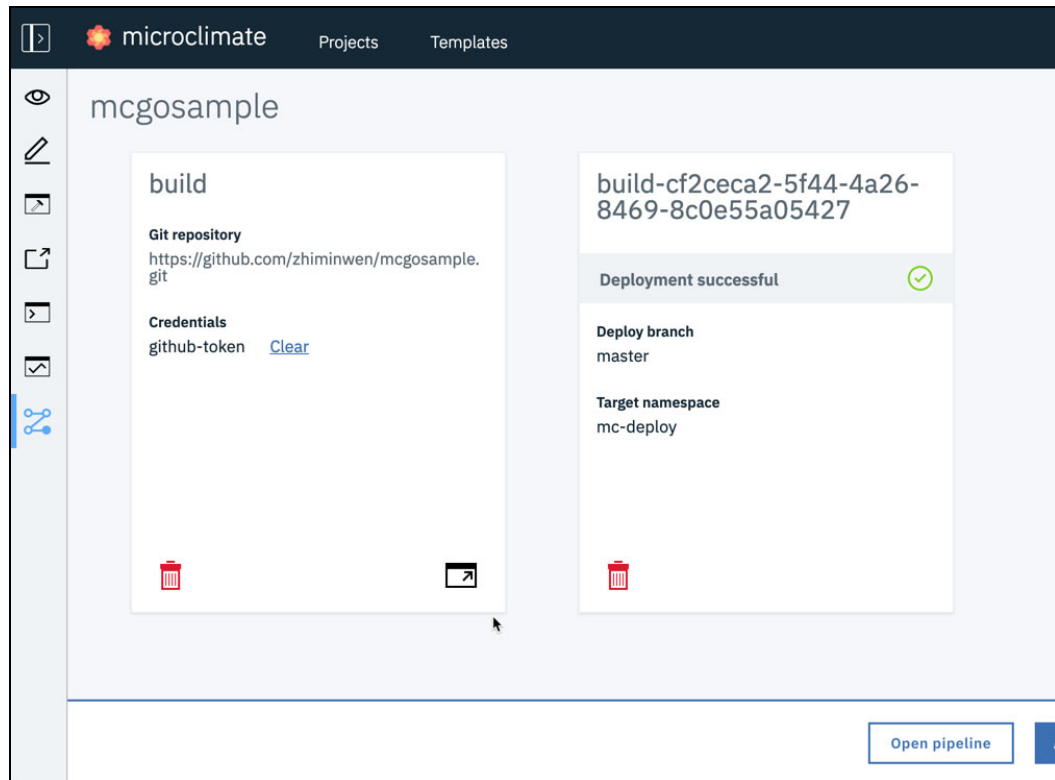


Figure 3-13 Microclimate deployment

3. Check the target namespace (mc-deploy) to see that the app is deployed and running, as shown in Example 3-9.

Example 3-9 Pods running in the target namespace

```
kubectl get pods -n mc-deploy
```

NAME	READY	STATUS	RESTARTS	AGE
mcgosample-deployment-59d8564d6d-nhnjr	1/1	Running	0	7m

4. List the service, as shown in Example 3-10.

Example 3-10 Service of the application deployed

```
kubectl get svc -n mc-deploy
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)
mcgosample-service	NodePort	172.21.200.113	<none>	8000:31325/TCP

You can then browse the port 31325 by using the proxy node IP. You should see the greeting message that is shown in Figure 3-14.

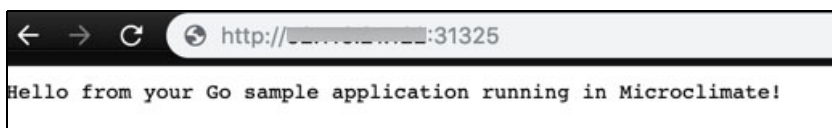


Figure 3-14 Application deployment



Managing your service mesh by using Istio

This chapter describes some practical use cases for how to use Istio to manage your service mesh.

This chapter includes the following topics:

- ▶ 4.1, “Introduction” on page 98
- ▶ 4.2, “Traffic management and application deployment” on page 98
- ▶ 4.3, “Application testing” on page 109
- ▶ 4.4, “Enforcing policy controls” on page 131

4.1 Introduction

Istio is an open platform that is used to connect, secure, control, and observe microservices:

- Connect

It allows you to connect different microservices through decoupling the traffic management by way of Istio, as described in 4.2, “Traffic management and application deployment”. It also allows you to test applications by injecting delays and faults without changing the application code, as described in 4.3, “Application testing”.

- Secure

It allows you to secure your microservices through managed authentication, authorization, and encryption of communication between services, as described in “Chapter 8- Security” in *IBM Private Cloud Systems Administrator's Guide*, SG24-8440. For more information about how to securely configure the ingress rules for your microservices and egress rules to communicate to an external HTTPS service, see 4.3, “Application testing” on page 109.

- Control

It allows enforcing policies to eliminate and decouple policy logic from the code, as described in 4.4, “Enforcing policy controls”.

- Observe

It allows observing your microservices through integrated telemetry, monitoring, tracing, and logging.

For more introduction about Istio and service mesh, see Chapter 8 of *IBM Cloud Private: System Administrator's Guide*, SG24-8441.

4.2 Traffic management and application deployment

In this section, we describe how to use the Istio techniques to achieve the same kinds of deployments, which are done with the native Kubernetes, as described in Chapter 1, “Highly available workloads and deployment on IBM Cloud Private” on page 1. We also describe the extra features that Istio brings for your application rollout.

You still use the same sample Golang application v1.0 and v1.1 as was used in Chapter 1, “Highly available workloads and deployment on IBM Cloud Private” on page 1.

4.2.1 Setup prerequisites

Complete the following steps for setting up the prerequisites:

1. Clean up the existing namespace deploy-demo by removing the deployment and services, as shown in Example 4-1.

Example 4-1 Removing the existing deployment and service

```
kubectl -n deploy-demo delete deployment <deployment names>
kubectl -n deploy-demo delete service <service names>
```

2. Assuming Istio is up and running, you must label the target namespace `deploy-demo` with `istio-injection` for Istio to manage the traffic. This process enables the automatic sidecar container injection, as shown in Example 4-2.

Example 4-2 Allowing sidecar auto injection

```
kubectl label namespace deploy-demo istio-injection=enabled
```

3. As discussed in Chapter 8 of *IBM Private Cloud Systems Administrator's Guide*, SG24-8440, we need to create an image policy to allow the envoy container to run. Apply the yaml content (see Example 4-3), which is saved as `image.policy.yaml`, by using the **`kubectl apply -f image.policy.yaml`** command.

Example 4-3 Image policy

```
apiVersion: securityenforcement.admission.cloud.ibm.com/v1beta1
kind: ImagePolicy
metadata:
  name: my-images-whitelist
  namespace: deploy-demo
spec:
  repositories:
    - name: docker.io/*
```

4.2.2 Deploying the application versions 1.0 and 1.1

Complete the following steps to deploy the application versions 1.0 and 1.1:

1. Deploy the application version 1.0 with the yaml files, as shown in Example 4-4.

Example 4-4 Version 1.0 deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: demo-v1
  labels:
    app: demo
    version: v1.0
spec:
  replicas: 5
  selector:
    matchLabels:
      app: demo
  template:
    metadata:
      labels:
        app: demo
        version: v1.0
    spec:
      containers:
        - name: demo
          image: zhiminwen/update-demo:v1.0
          env:
            - name: LISTEN_PORT
              value: "8080"
          livenessProbe:
```

```
    httpGet:
      path: /healthz
      port: 8080
  readinessProbe:
    httpGet:
      path: /readyz
      port: 8080
```

2. Deploy the application version 1.1 with yaml files, as shown in Example 4-5.

Example 4-5 Version 1.1 deployments

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: demo-v2
  labels:
    app: demo
    version: v1.1
spec:
  replicas: 5
  selector:
    matchLabels:
      app: demo
  template:
    metadata:
      labels:
        app: demo
        version: v1.1
    spec:
      containers:
      - name: demo
        image: zhiminwen/update-demo:v1.1
        env:
        - name: LISTEN_PORT
          value: "8080"
        livenessProbe:
          httpGet:
            path: /healthz
            port: 8080
        readinessProbe:
          httpGet:
            path: /readyz
            port: 8080
```

Notice that we set the two deployments independently. They share the “app” label, but have a different “version” label.

3. Create a service, as shown in Example 4-6.

Example 4-6 Service for application

```
apiVersion: v1
kind: Service
metadata:
  name: demo-svc
  labels:
```

```
    app: demo
spec:
  type: NodePort
  ports:
  - port: 80
    targetPort: 8080
    protocol: TCP
    name: http
  selector:
    app: demo
```

4. Run a **curl test** command to ensure that the app is deployed successfully, as shown in Example 4-7.

Example 4-7 Validating the application

```
while true; do curl 52.116.21.122:31942;echo; sleep 1; done
```

```
Hello World! from host:demo-v1-764c49c785-xp5jd ver:1.0
Hello World! from host:demo-v2-78bd765b9f-fs6lx ver:1.1
Hello World! from host:demo-v2-78bd765b9f-fs6lx ver:1.1
```

4.2.3 Creating an Istio gateway and destination rule

Complete the following steps to create an Istio gateway and destination rule:

1. Create a file that is named **gw.yaml** with the content that is shown in Example 4-8. Deploy it by using the **kubectl apply -f gw.yaml** command.

Example 4-8 Istio gateway

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: hello-istio-gw
  namespace: deploy-demo
spec:
  selector:
    istio: ingressgateway
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
    - demo.52.116.21.122.nip.io
```

In this example, you created a gateway service on the Istio pods that were selected by the label `istio=ingressgateway`. You applied the gateway settings on the Istio ingress pods. When the HTTP traffic with the Host field in the HTTP header is set as `demo.52.116.21.122.nip.io`, the traffic is served through this gateway.

2. Create a file and name it `destination.rule.yaml`. Deploy this file by using the `kubectl apply -f destination.rule.yaml` command (see Example 4-9).

Example 4-9 Istio destination rule

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: hello-app-destination
  namespace: deploy-demo
spec:
  host: demo-svc.deploy-demo.svc.cluster.local
  subsets:
  - name: hello-v10
    labels:
      version: v1.0
  - name: hello-v11
    labels:
      version: v1.1
```

The destination service is set as the service name of the application in Kubernetes. Use the fully qualified name `demo-svc.deploy-demo.svc.cluster.local` to avoid any cross namespace resolving issues. Define two subsets with the labels of versions that are used to select the correct pods for the request to be served.

With these settings, you are now ready to explore the different deployment options with Istio.

4.2.4 Canary testing with Istio

Complete the following steps:

1. Create an Istio virtual service object, as shown in Example 4-10.

Example 4-10 Canary virtual service

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: hello-app-vs
  namespace: deploy-demo
spec:
  hosts:
  - demo.52.116.21.122.nip.io
  gateways:
  - hello-istio-gw
  http:
  - route:
    - destination:
        host: demo-svc.deploy-demo.svc.cluster.local
        subset: hello-v10
      weight: 90
    - destination:
        host: demo-svc.deploy-demo.svc.cluster.local
        subset: hello-v11
      weight: 10
```

2. Set the host list as `demo.52.116.21.122.nip.io`, which matches what was defined in the Istio gateway. This way, for any HTTP request that includes the Host field in the request header matching this value, the traffic is served by this virtual service.
3. Set the destination host by using the Kubernetes service name and the subset, which are defined in the destination rule. Two destinations are available in this example: one for version 1.0 and the other for version 1.1. We assign the weight of the new service v1.0 as 10%, and the v1.0 as 90%. This assignment implements the canary deployment.

Tip: Compare this method with the native Kubernetes way of implementation. Kubernetes achieves the canary deployment by using different number of replicas, while Istio manages the traffic directly by assigning different weights to services. By using Istio, we can control the canary percentages more easily.

4. You can test the result by running the command that is shown in Example 4-11.

Example 4-11 Canary validation

```
while true; do curl 52.116.21.122:31380 -H "Host: demo.52.116.21.122.nip.io";  
echo; sleep 1; done
```

```
Hello World! from host:demo-v2-78bd765b9f-jc4kl ver:1.1  
Hello World! from host:demo-v1-764c49c785-gjxnb ver:1.0  
Hello World! from host:demo-v1-764c49c785-m7rqq ver:1.0  
Hello World! from host:demo-v1-764c49c785-8xrnw ver:1.0  
Hello World! from host:demo-v1-764c49c785-m7rqq ver:1.0  
Hello World! from host:demo-v1-764c49c785-8xrnw ver:1.0  
Hello World! from host:demo-v1-764c49c785-8xrnw ver:1.0  
Hello World! from host:demo-v1-764c49c785-8xrnw ver:1.0  
Hello World! from host:demo-v1-764c49c785-xp5jd ver:1.0  
Hello World! from host:demo-v1-764c49c785-gjxnb ver:1.0  
Hello World! from host:demo-v2-78bd765b9f-65k6l ver:1.1  
Hello World! from host:demo-v1-764c49c785-xp5jd ver:1.0  
Hello World! from host:demo-v1-764c49c785-8xrnw ver:1.0  
Hello World! from host:demo-v1-764c49c785-hg8k6 ver:1.0  
Hello World! from host:demo-v1-764c49c785-8xrnw ver:1.0  
Hello World! from host:demo-v1-764c49c785-xp5jd ver:1.0  
Hello World! from host:demo-v1-764c49c785-gjxnb ver:1.0  
Hello World! from host:demo-v1-764c49c785-hg8k6 ver:1.0  
Hello World! from host:demo-v1-764c49c785-xp5jd ver:1.0
```

4.2.5 Blue-green testing with Istio

Create an Istio virtual service object as defined in Example 4-12.

Example 4-12 Blue-green virtual service

```
apiVersion: networking.istio.io/v1alpha3  
kind: VirtualService  
metadata:  
  name: hello-app-vs  
  namespace: deploy-demo  
spec:  
  hosts:  
  - demo.52.116.21.122.nip.io  
  gateways:
```

```

- hello-istio-gw
http:
  - route:
    - destination:
        host: demo-svc.deploy-demo.svc.cluster.local
        subset: hello-v10
        weight: 0
    - destination:
        host: demo-svc.deploy-demo.svc.cluster.local
        subset: hello-v11
        weight: 100

```

To have the blue-green testing, you can adjust the weights of the two services. Assuming that the green application is fully tested, you can set the weight as 100. With this configuration, the service is 100% routed to the new version.

4.2.6 A/B testing with Istio

By using the dynamic routing feature of Istio, you can implement A/B testing to allow the traffic to flow to the new version of the application. Complete the following steps:

1. As shown in Example 4-13, we route the request to the new version of the application when the users are using the Chrome browser.

Example 4-13 A/B testing virtual service

```

apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: hello-app-vs
  namespace: deploy-demo
spec:
  hosts:
  - demo.52.116.21.122.nip.io
  gateways:
  - hello-istio-gw
  http:
  - match:
    - headers:
        user-agent:
          regex: "^.*Chrome.*$"
    route:
    - destination:
        host: demo-svc.deploy-demo.svc.cluster.local
        subset: hello-v11
  - route:
    - destination:
        host: demo-svc.deploy-demo.svc.cluster.local
        subset: hello-v10

```

In the HTTP ordered list, if the HTTP header that includes the field `user-agent` regex matches the Chrome, Istio routes the traffic to the `hello-v11` subset, hence the new version of the application.

2. Test this configuration by using the `curl` command, as shown in Example 4-14.

Example 4-14 Test A/B testing

```
curl 52.116.21.122:31380 -H "Host: demo.52.116.21.122.nip.io" -H "User-Agent: Chrome"; echo
```

```
Hello World! from host:demo-v2-78bd765b9f-lrwnt ver:1.1
```

As expected, you see the response from version 1.1.

3. Start your Chrome browser and modify the Header with the ModHeader extension, as shown in Figure 4-1.

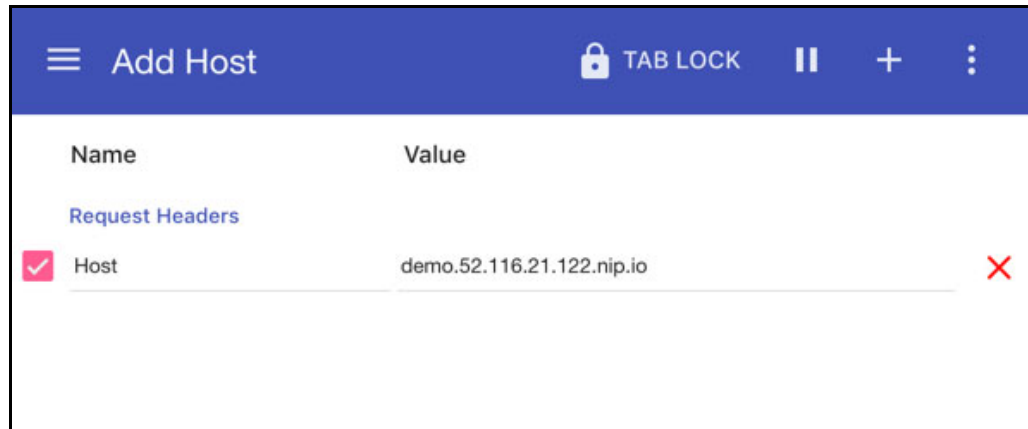


Figure 4-1 Modify the header field of Host

4. Browse to the URL, as shown in Figure 4-2.



Figure 4-2 Chrome browser result

You see that the user who uses the Chrome browser is served by the new version of the application.

If you want a new API service to follow the A/B testing approach, you can specify a header value; for example, `api-version/`. Then, create the virtual service rules to match this value and direct the traffic to the new version.

4.2.7 Mirroring the traffic by using Istio

At times, you might want to test your application with a real production workload traffic before it goes on live. You can achieve this by using Istio's traffic mirroring feature.

Complete the following steps:

1. Update the application to log the request and response, as shown in Example 4-15. Set the version to 1.2.

Example 4-15 Updated application

```
// skipped
func init() {
    version = "1.2"
}
func greet(w http.ResponseWriter, r *http.Request) {
    requestDump, err := httputil.DumpRequest(r, true)
    if err != nil {
        log.Printf("failed to dump the request:%v", err)
    }
    log.Print(string(requestDump))
    resp := fmt.Sprintf("Hello World! from host:%s ver:%s", os.Getenv("HOSTNAME"),
version)
    log.Printf("response:%s", resp)
    fmt.Fprint(w, resp)
}
//skipped
```

2. Compile the application.
3. Build the docker image and tag it with v1.2.
4. Push the image to the Docker Hub.
5. Deploy the application that is named demo-v3 by using the yaml file, as shown in Example 4-16.

Example 4-16 Deploying yaml file of the new application

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: demo-v3
  labels:
    app: demo
    version: v1.2
spec:
  replicas: 1
  selector:
    matchLabels:
      app: demo
  template:
    metadata:
      labels:
        app: demo
        version: v1.2
    spec:
      containers:
```

```
- name: demo
  image: zhiminwen/update-demo:v1.2
  env:
    - name: LISTEN_PORT
      value: "8080"
  livenessProbe:
    httpGet:
      path: /healthz
      port: 8080
  readinessProbe:
    httpGet:
      path: /readyz
      port: 8080
```

6. Deploy only one replica so that you can track the logs more easily.
7. Update the Istio destination rule, as shown in Example 4-17. Name the new subset hello-v12 and define the label of version to v1.2.

Example 4-17 Updated Istio destination rule

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: hello-app-destination
  namespace: deploy-demo
spec:
  host: demo-svc.deploy-demo.svc.cluster.local
  subsets:
    - name: hello-v10
      labels:
        version: v1.0
    - name: hello-v11
      labels:
        version: v1.1
    - name: hello-v12
      labels:
        version: v1.2
```

8. Update the virtual service definition, as shown in Example 4-18.

Example 4-18 Updated Istio virtual service

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: hello-app-vs
  namespace: deploy-demo
spec:
  hosts:
    - demo.52.116.21.122.nip.io
  gateways:
    - hello-istio-gw
  http:
    - route:
        - destination:
            host: demo-svc.deploy-demo.svc.cluster.local
```

```

subset: hello-v10
mirror:
  host: demo-svc.deploy-demo.svc.cluster.local
  subset: hello-v12

```

9. Set the first route rule (default rule) to v1.0 application by using the subset hello-v10. Add the mirror to the hello-v12 subset so that the same traffic is routed to the v1.2 pod. The new version of the application can now be tested with the production request.
10. Validate the setup. Determine the v1.2 pod by using the command that is shown in Example 4-19.

Example 4-19 Find out the new version application pod

```

kubectl -n deploy-demo get pods

```

NAME	READY	STATUS	RESTARTS	AGE
demo-v1-64465b486c-c4j58	2/2	Running	0	28m
demo-v1-64465b486c-j57jw	2/2	Running	0	28m
demo-v1-64465b486c-nwb9h	2/2	Running	0	28m
demo-v1-64465b486c-tgpb7	2/2	Running	0	28m
demo-v1-64465b486c-w7fjp	2/2	Running	0	28m
demo-v2-5bb7c4f45c-c55q5	2/2	Running	0	28m
demo-v2-5bb7c4f45c-v9h7r	2/2	Running	0	28m
demo-v2-5bb7c4f45c-wwmc4	2/2	Running	0	28m
demo-v2-5bb7c4f45c-xqghm	2/2	Running	0	28m
demo-v2-5bb7c4f45c-z4nq6	2/2	Running	0	28m
demo-v3-7c8d5d7ff9-stvls	2/2	Running	0	28m

11. Copy the name of the only pod whose name starts with demo-v3 and monitor its log by running the `kubectl -n deploy-demo logs demo-v3-7c8d5d7ff9-stvls -c demo -f` command.
12. Open another shell and trigger a request to the v1.0 application, as shown in Example 4-20.

Example 4-20 Request to application

```

curl 52.116.21.122:31380 -H "Host: demo.52.116.21.122.nip.io";echo
Hello World! from host:demo-v1-64465b486c-w7fjp ver:1.0

```

You should see the log output of the v1.2 pods that were started to receive the real traffic, as shown in Example 4-21.

Example 4-21 Mirrored request to the application of v1.2

```

kubectl -n deploy-demo logs demo-v3-7c8d5d7ff9-stvls -c demo -f
2019/02/27 20:19:09 starting server on port:8080

2019/02/27 20:21:09 GET / HTTP/1.1
Host: demo.52.116.21.122.nip.io-shadow
Accept: */*
Content-Length: 0
User-Agent: curl/7.54.0
X-B3-Sampled: 1
X-B3-Spanid: fb35f52aeace793f
X-B3-Traceid: fb35f52aeace793f
X-Envoy-Decorator-Operation: demo-svc.deploy-demo.svc.cluster.local:80/*
X-Envoy-Internal: true
X-Forwarded-For: 10.73.147.239, 172.20.74.12

```

```
X-Forwarded-Proto: http
X-Istio-Attributes:
Ck8KCNvdXJjZS51aWQSQRI/a3ViZXJuZXR1czovL21zdG1vLW1uZ3Jlc3NnYXR1d2F5LTU2NzU4YmY5Nj
gtNGI2cXouaXN0aW8tc3lzdGVtCj8KE2Rlc3RpbmF0aW9uLnN1cnZpY2USKBIWZGVtby1zdmMuZGVwbG95
LWR1bW8uc3ZjLmNsdXNOZXIubG9jYWwKQgoXZGVzdG1uYXRpb24uc2VydmljZS51aWQSQjxI1aXN0aW86Ly
9kZXBs3ktZGVtby9zZXJ2aWN1cy9kZW1vLXN2YwpEChhkZXN0aW5hdG1vbi5zZXJ2aWN1Lmhvc3QSKBIm
ZGVtby1zdmMuZGVwbG95LWR1bW8uc3ZjLmNsdXNOZXIubG9jYWwKLgodZGVzdG1uYXRpb24uc2VydmljZS
5uYW1lc3BhY2USDRIILZGVwbG95LWR1bW8KJgoYZGVzdG1uYXRpb24uc2VydmljZS5uYW1lEgoSCGR1bW8t
c3Zj
X-Request-Id: albeac06-7a23-9285-85ec-ba5fc5e1db0e

2019/02/27 20:21:09 response:Hello World! from host:demo-v3-7c8d5d7ff9-stv1s
ver:1.2
```

4.3 Application testing

In this section, we describe how to perform the following tasks within the context of a scenario in which your microservice is integrated with an external public service over the internet:

1. Create an instance of the Watson Language Translator service.
2. Clone the GitHub repository that includes the sample code for the microservice.
3. Deploy and test the microservice on IBM Cloud Private.
4. Define an Istio egress rule to integrate with the external service.
5. Expose services to be used externally by defining ingress rules.
6. Handle the unreliable external service by simulating different failures through Istio Abort Injection and Delay Injection.
7. Enhance the application resiliency by adding automatic retry attempts and setting request timeouts.

No change to your code: You can perform all of these steps *without* changing your application code.

4.3.1 Creating an instance of the Watson Language Translator service

In this section, an instance of the Watson Language Translator service is created on IBM Cloud. It is used to integrate with your sample microservice. Later on, you configure Istio egress rules to connect to it.

Note: This exercise shows a sample integration scenario. The same logic applies to any other integrations and are not tied to the services on IBM Cloud.

Complete the following steps to create an instance of the Watson Language Translator service:

1. Log in to IBM Cloud console by using <https://cloud.ibm.com>, as shown in Figure 4-3 on page 110.

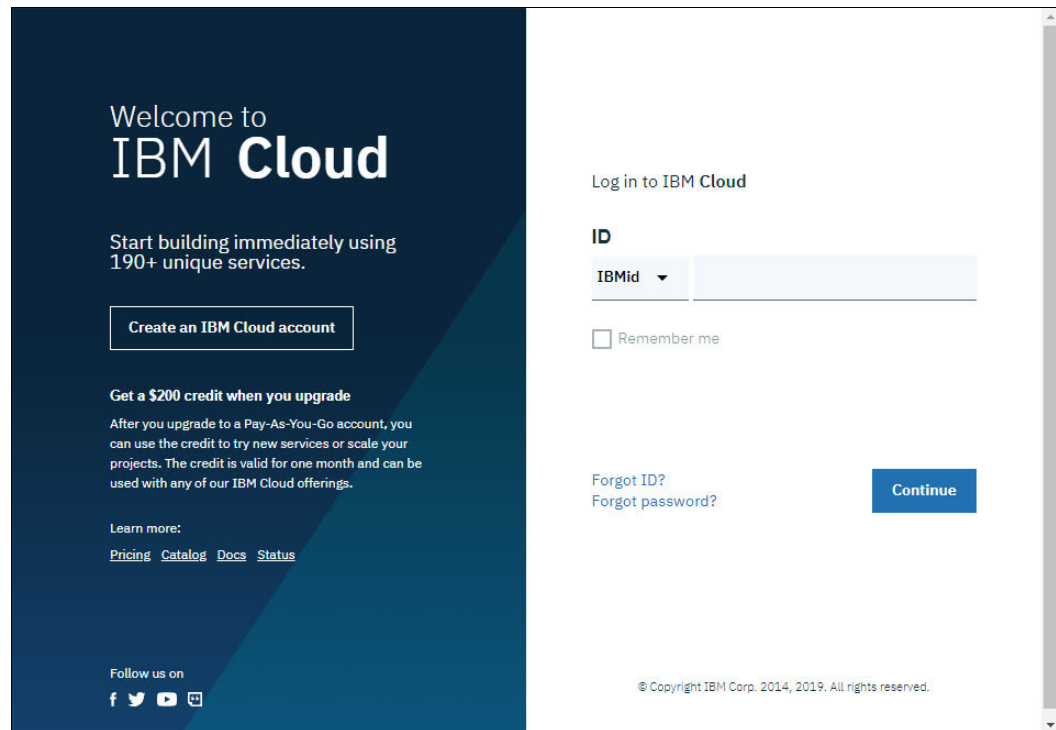


Figure 4-3 IBM Cloud login page

2. Click **Catalog** in the top toolbar.
3. Locate and select the **Language Translator** service, as shown in Figure 4-4.

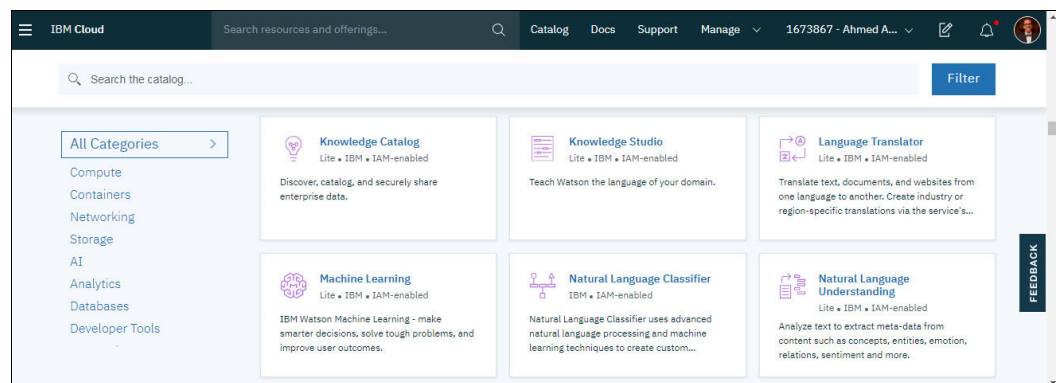


Figure 4-4 IBM Cloud catalog

4. Select the region where you want to deploy the service; then, click **Create**.
5. Complete the following steps to copy the credentials to authenticate to your service instance:
 - a. Click the **Manage** tab in the left bar.
 - b. Copy the API Key and the URL, as shown in Figure 4-5 on page 111.

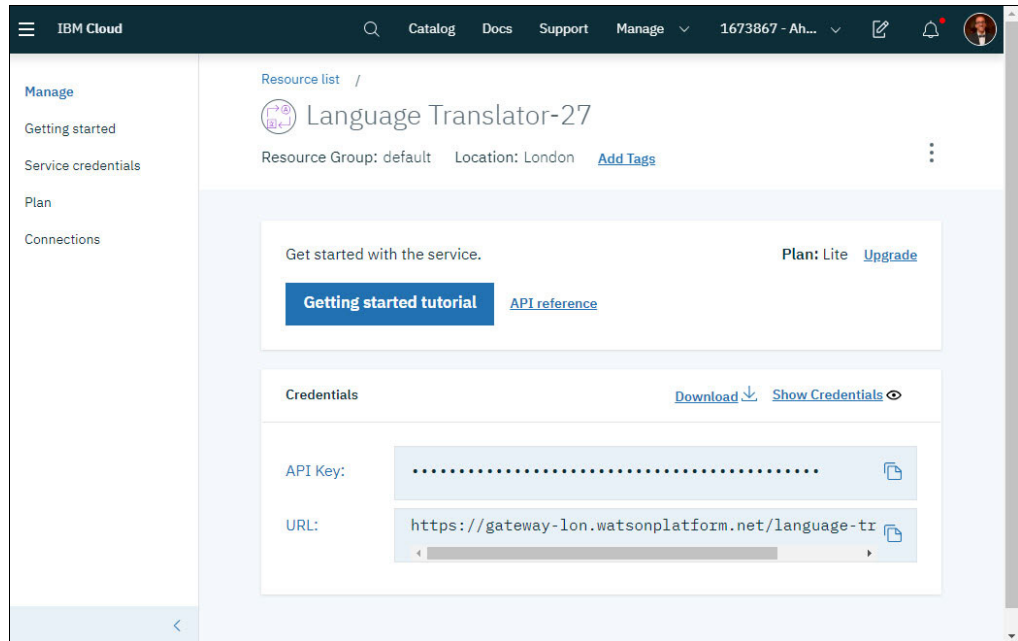


Figure 4-5 Language Translator credentials

6. Language Translator provides an REST API that translates a text. Download and use **curl** (<https://curl.haxx.se/download.html>) to test it. You can also use Postman (<https://www.getpostman.com/downloads/>) or any other REST API testing tool for testing purposes.
7. If you are using **curl**, write the following command in your terminal. This command calls the Language Translator API to translate How are you? from English to French. Replace {apikey} and {url} with the values that were retrieved in step 5.

```
curl -X POST -u "apikey:{apikey}" --header "Content-Type: application/json"
--data "{\"text\": \"How are you?\", \"model_id\": \"en-fr\"}"
"{url}/v3/translate?version=2018-05-01"
```

Your output should look like the output that is shown in Example 4-22.

Example 4-22 Language Translator sample output

```
{
  "translations" : [ {
    "translation" : "Comment es-tu?"
  } ],
  "word_count" : 3,
  "character_count" : 12
}
```

4.3.2 Cloning GitHub repository including microservice sample code

In this section, you clone the sample code of the microservices that integrates with the Language Translator API. This simple Node.js code passes a text to the Language Translator API in English, then the service translates it into French. That is, the same scenario that you tried by using **curl** in 4.3.1, “Creating an instance of the Watson Language Translator service” on page 109.

Review the code that is found at this [GitHub web page](#).

Open `/services/watsonService.js` to view the procedure for calling the external service. As an environment variable, you provide the `API_KEY` and `WATSON_TRANSLATOR_API` that was retrieved in step 5 in 4.3.1, “Creating an instance of the Watson Language Translator service” on page 109. Later, you pass the text to the service to be translated.

Complete the following steps to clone the code:

1. Download and Install Git client (if not already installed) from [this web page](#).
2. Run the following command to clone the GitHub repository:

```
git clone
https://github.com/IBMRedbooks/SG248441-IBM-Cloud-Private-Application-Developer
-s-Guide.git
```

4.3.3 Deploying and testing the microservice on IBM Cloud Private

In this section, you deploy and test the microservice on IBM Cloud Private without the use of Istio.

Building the container image

Complete the following steps to build the container image:

1. Browse to the following translator-microservice directory:

```
cd
SG248441-IBM-Cloud-Private-Application-Developer-s-Guide\Ch4-Manage-your-Servic
e-Mesh-with-Istio\application-testing\translator-microservice
```

2. View the `DockerFile` that includes instructions to build the container image, as shown in Example 4-23. The instructions retrieve the base image (a Node.js image) that is based on the Linux Alpine project. The instructions later copy the files to the container image, install the app dependencies, and expose the app on a specific port.

Example 4-23 Dockerfile content

```
FROM node:6.12.0-alpine
```

```
# Install the application
ADD package.json /app/package.json
RUN cd /app && npm install
COPY /routes/* /app/routes/
COPY /services/* /app/services/
COPY /views/* /app/views/
ADD app.js /app/app.js
ENV WEB_PORT 80
EXPOSE 80
```

```
# Define command to run the application when the container starts
CMD ["node", "/app/app.js"]
```

Tip: The use of the `npm install` command installs the dependencies for the application that is defined in `package.json`. Notice that `npm install` is run early in the Dockerfile.

Docker works in a layer concept, which means that when you change an upper layer, Docker does not rebuild the lower layer. In this example, when you change `app.js`, Docker does not reinstall the dependencies, which results in a more efficient build.

As a recommended practice, order your instructions so that the less frequently changed instructions are placed first (to ensure that the build cache is reusable) and the more frequently changed instructions are placed last.

3. Build the image as shown in Example 4-24.

Example 4-24 Build the image

```
docker image build -t <Registry Repository>/translator-microservice:2.0 .
```

```
Sending build context to Docker daemon 73.73kB
Step 1/10 : FROM node:6.12.0-alpine
---> fa74995b3b27
Step 2/10 : ADD package.json /app/package.json
---> 1f825a242533
Step 3/10 : RUN cd /app && npm install
---> Running in 37bbcd81496c
translator-microservice@1.0.0 /app
+-- body-parser@1.18.3
| +-- bytes@3.0.0
...
OUTPUT OMITTED FOR CLARITY
...
---> fc16e71128c8
Step 4/10 : COPY /routes/* /app/routes/
---> 42d1e3715243
Step 5/10 : COPY /services/* /app/services/
---> 0dd2433461ff
Step 6/10 : COPY /views/* /app/views/
---> b07f2852b78c
Step 7/10 : ADD app.js /app/app.js
---> 8ab805d703dd
Step 8/10 : ENV WEB_PORT 80
---> Running in 22a0cde8772e
Removing intermediate container 22a0cde8772e
---> 71378f99b5fd
Step 9/10 : EXPOSE 80
---> Running in 784b2baca744
Removing intermediate container 784b2baca744
---> e85c47a9b975
Step 10/10 : CMD ["node", "/app/app.js"]
---> Running in 392069a8903e
Removing intermediate container 392069a8903e
---> 7e71bd05d560
Successfully built 7e71bd05d560
Successfully tagged aazraq/translator-microservice:2.0
```

4. Log in to your Docker Registry:
 - **docker login** in case you are using the DockerHub.
 - **docker login <IBM Cloud Private Domain>:8500** in case you are using the IBM Cloud Private Docker Registry.
5. Push your image to the Docker Registry by running the command that is shown in Example 4-25.

Example 4-25 Pushing the image

docker push <Registry Repository>/translator-microservice:2.0

```
docker push aazraq/translator-microservice:2.0
The push refers to repository [docker.io/aazraq/translator-microservice]
d35a671ac3a1: Pushed
65f4377b6e65: Pushed
79aef7c2c8a2: Pushed
0af0c62a1337: Pushed
f36b26657f2e: Pushed
a36b93489ff7: Pushed
31c270e8fe7e: Layer already exists
86a4d08dfa94: Layer already exists
52a5560f4ca0: Layer already exists
2.0: digest:
sha256:f54a05207e8a7486da1349c4d2fd8e696887ca6c64b801452fb80def27e3935b size:
2197
```

Deploying the microservice on IBM Cloud Private

Complete the following steps to deploy the microservice on IBM Cloud Private:

1. Log in to your IBM Cloud Private cluster from your local workstation. This process makes your kubectl commands point to the master node of your cluster. You use the same namespace that is used in 4.2, “Traffic management and application deployment” on page 98, which is deploy-demo. Run the following command:

```
cloudctl login -a CLUSTER_URL -n deploy-demo --skip-ssl-validation
```
2. Disable the Istio Injection on the deploy-demo namespace by running the following command. In the first scenario, you test it without using istio:

```
kubectl label namespace deploy-demo istio-injection=disabled --overwrite
```
3. Browse to the yaml directory. Then, edit the line that begins with “image:” in 01translator-ms-deployment.yaml (see Example 4-26) to point to the container image repository that you pushed in the previous step.

Example 4-26 Edit the line starting with image: in 01translator-ms-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: translator-ms-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: translator-ms
  template:
```

```

metadata:
  labels:
    app: translator-ms
spec:
  containers:
  - name: translator-ms
    image: <Registry Repository>/translator-microservice:2.0
    env:
      - name: WATSON_TRANSLATOR_API
        valueFrom:
          secretKeyRef:
            name: watson-translator
            key: url
      - name: API_KEY
        valueFrom:
          secretKeyRef:
            name: watson-translator
            key: api-key
    resources:
      limits:
        memory: "200Mi"
        cpu: "250m"
      requests:
        memory: "100Mi"
        cpu: "125m"
    ports:
      - containerPort: 80

```

4. This deployment expects a Kubernetes secret object that is named `watson-translator` with two keys: `url` and `api-key`.

Complete the following steps to create the secret:

- a. Browse to and log in to the IBM Cloud Private console by using a web browser.
- b. Click the **Menu** icon in the toolbar. Then, click **Configuration** and **Secrets**, as shown in Figure 4-6 on page 116.

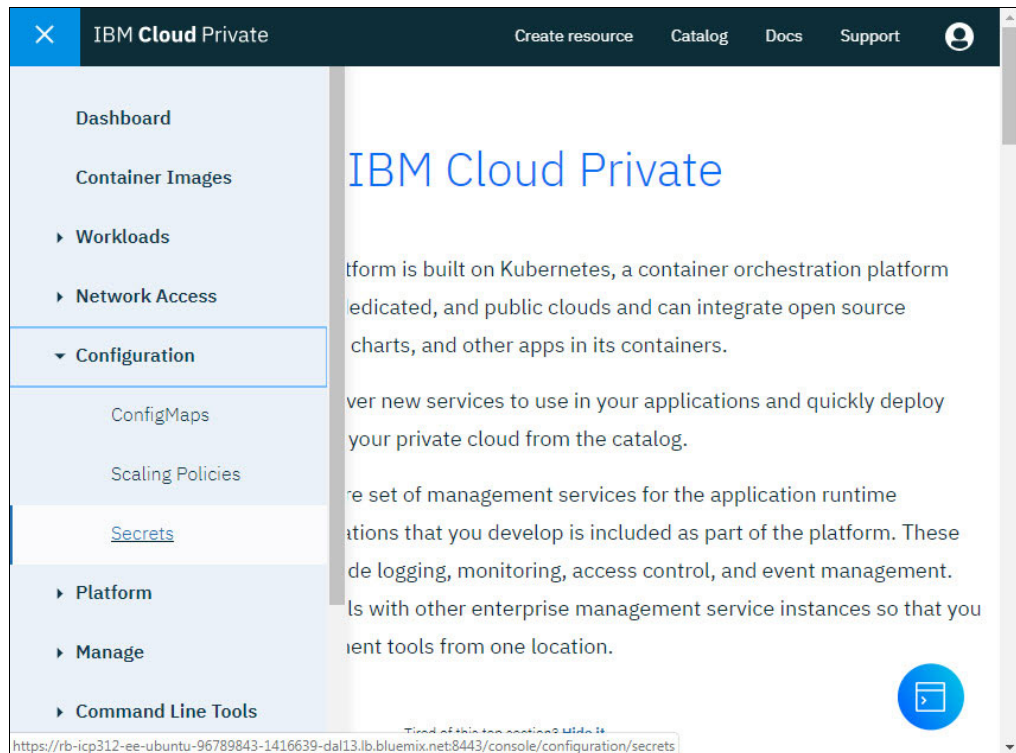


Figure 4-6 IBM Cloud Private menu

- c. Click **Create Secret**, as shown in Figure 4-7.

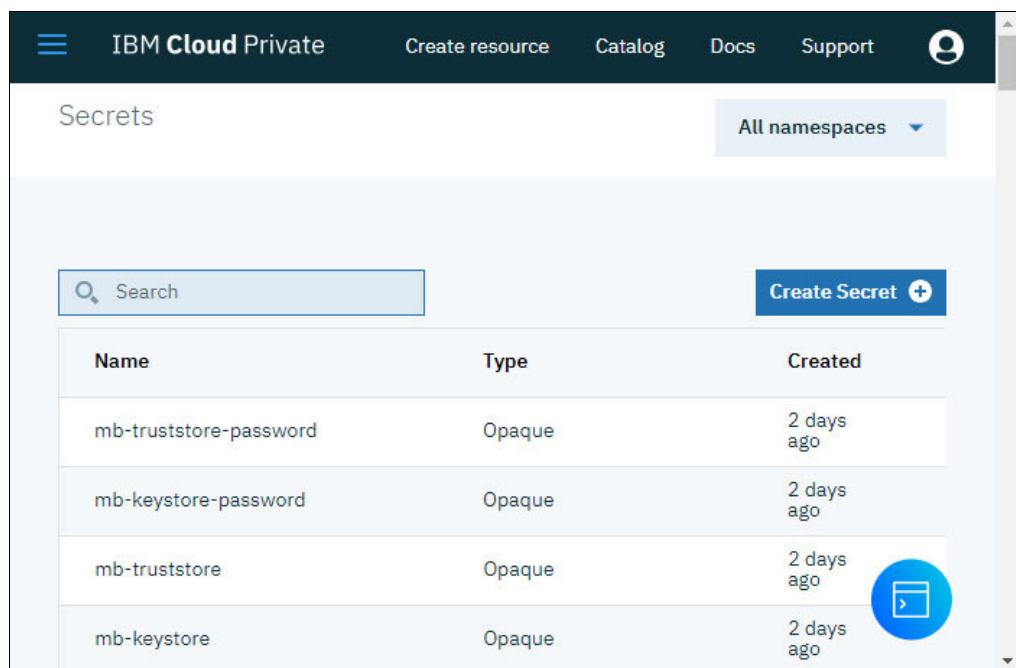


Figure 4-7 Secrets

- d. In the General tab, add the name of the Secret as `watson-translator`, and select **deploy-demo** for the namespace, as shown in Figure 4-8.

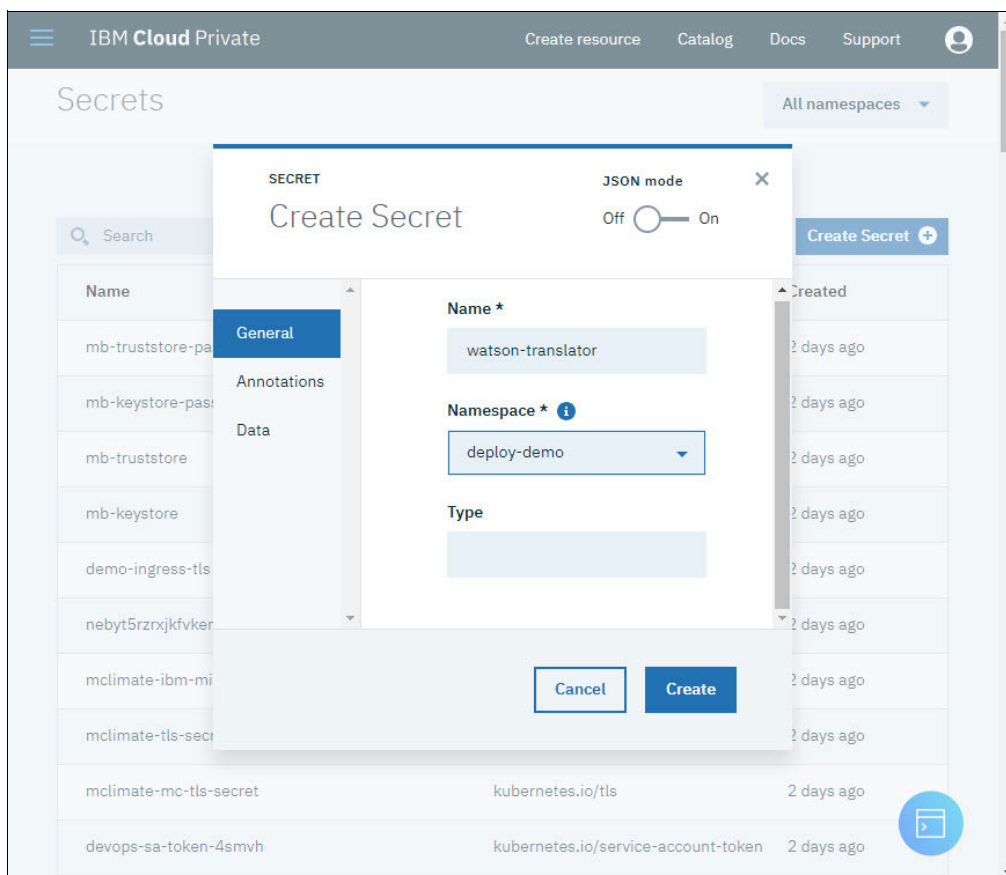


Figure 4-8 Create Secret - General tab

- e. Click **Data** to add the keys and values properties of this secret.
- f. Retrieve the value of `url` and `api-key` from step 5 in 4.3.1, “Creating an instance of the Watson Language Translator service” on page 109.
- g. Run the following command to decode the values of `url` and `api-key` to Base64 by using the command line or any other tool:

```
echo -n <api-key>|base64
echo -n <url>|base64
```

- h. Add the following key value pairs to store within the secret: url, api-key with their encoded Base64 as value (see Figure 4-9).

Figure 4-9 Create Secret - Data tab

- i. Click **Create** to create the secret.
5. Return to the terminal. Apply the deployment to deploy the application to IBM Cloud Private by running the following command:


```
kubectl apply -f 01translator-ms-deployment.yaml
deployment.apps/translator-ms-deployment created
```
6. To ensure that the pods are created successfully, run the command that is shown in Example 4-27.

Example 4-27 *get pods command*

```
kubectl get pods --selector=app=translator-ms
```

NAME	READY	STATUS
translator-ms-deployment-854c849cbd-6ssdr	0/1	CreateContainerConfigError
0 49m		
translator-ms-deployment-854c849cbd-bsfzb	0/1	CreateContainerConfigError
0 49m		
translator-ms-deployment-854c849cbd-k4vtr	0/1	CreateContainerConfigError
0 49m		

7. The results show that pods failed with an error CreateContainerConfigError. To understand the reason for failure, run the **kubectl describe pod {pod-name}** command.

Notice that the error is “Error: container has runAsNonRoot and image will run as root on machine”. The reason for this error is that the container you are creating runs as root, which is not a recommended practice. It is stopped by default by the IBM Cloud Private pod security policy, as described in Chapter 8 of *IBM Private Cloud Systems Administrator’s Guide*, SG24-8440.

8. For illustration purposes, enable the privileged container to run inside the deploy-demo namespace by applying the RoleBinding, as shown in Example 4-28.

Example 4-28 RoleBinding to allow privileged containers access

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: ibm-privileged-clusterrole-rolebinding
  namespace: deploy-demo
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: ibm-privileged-clusterrole
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: Group
  name: system:serviceaccounts:deploy-demo
```

9. Apply the RoleBinding by running the following command:

```
kubectl apply -f role-binding.yaml
rolebinding.rbac.authorization.k8s.io/ibm-privileged-clusterrole-rolebinding
configured
```

10. Delete and apply the deployment again by running the following command:

```
kubectl delete -f 01translator-ms-deployment.yaml
deployment.apps "translator-ms-deployment" deleted

kubectl apply -f 01translator-ms-deployment.yaml
deployment.apps/translator-ms-deployment created
```

11. Run the **get pods** command again to ensure that the deployment is successful, as shown in Example 4-29.

Example 4-29 Successful deployment

```
kubectl get pods --selector=app=translator-ms
```

NAME	READY	STATUS	RESTARTS	AGE
translator-ms-deployment-854c849cbd-6xswp	1/1	Running	0	5m55s
translator-ms-deployment-854c849cbd-7rnb8	1/1	Running	0	5m55s
translator-ms-deployment-854c849cbd-ht2fz	1/1	Running	0	5m55s

12. Expose the service as NodePort, as shown in Example 4-30.

Example 4-30 Expose the service as NodePort

```
apiVersion: v1
kind: Service
metadata:
  name: translator-ms-service
spec:
  selector:
```

```
    app: translator-ms
  ports:
  - name: http
    protocol: TCP
    port: 80
  type: NodePort
```

13. Apply the service by running the following command:

```
kubectl apply -f 02translator-ms-service.yaml
service/translator-ms-service created
```

Testing the microservice

In this section, you test the microservice to ensure that it works as-is without introducing Istio.

Complete the following steps to test the microservice:

1. Get the NodePort for the service, as shown in Example 4-31.

Example 4-31 Get NodePort of the service

```
kubectl get services translator-ms-service
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
translator-ms-service	NodePort	172.21.119.12	<none>	80:30321/TCP	7h37m

2. As shown in Example 4-31, the NodePort is 30321. Open your browser and browse to <Proxy IP>:30321. You receive a similar window as shown in Figure 4-10.



Figure 4-10 Microservice UI

3. Enter any English-based text and then, click **Submit**. You should receive its French translation, as shown in Figure 4-11.

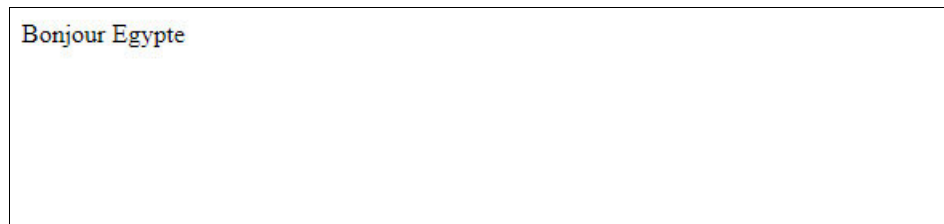


Figure 4-11 Result of translation of a sample text “Hello Egypt” to French through the microservice

You now successfully deployed and tested your microservice on IBM Cloud Private.

4.3.4 Defining an Istio egress rule to integrate with an external service

This scenario features the following high-level steps:

1. Enable Istio for the namespace.
2. Redeploy the application to inject Istio sidecar and notice that Istio blocks all external traffic.
3. Allow external outbound traffic and retest.

Enabling Istio for the namespace

Assuming Istio is up and running, for Istio to manage the traffic, you must label the target namespace `deploy-demo` with `istio-injection` to allow the automatic sidecar container injection, as shown in Example 4-32.

Example 4-32 Allowing sidecar auto injection

```
kubectl label namespace deploy-demo istio-injection=enabled --overwrite
namespace/deploy-demo labeled
```

Redeploying the application

To configure the sidecar injection of Istio at creation time, you must redeploy the app. Complete the following steps to redeploy and test the application:

1. Delete all pods that are related to the deployment, as shown in Example 4-33.

Example 4-33 Deleting all pods that are related to the deployment

```
kubectl delete pods --selector=app=translator-ms

pod "translator-ms-deployment-6998df47b7-6z7sq" deleted
pod "translator-ms-deployment-6998df47b7-jscn2" deleted
pod "translator-ms-deployment-6998df47b7-vns2z" deleted
```

2. Kubernetes self-healing capability ensures that the current state of replicaset matches the target state of replicaset that is defined in the deployment. It automatically self-heals the pods and creates pods. When it creates pods, Istio automatically injects its side car container into each pod. Notice that two containers are displayed for each pod, as shown in Example 4-34.

Example 4-34 Each pod now has two containers

```
kubectl get pods --selector=app=translator-ms
```

NAME	READY	STATUS	RESTARTS	AGE
translator-ms-deployment-649b855c6b-4n5t4	2/2	Running	0	2m38s
translator-ms-deployment-649b855c6b-ht6x2	2/2	Running	0	2m27s
translator-ms-deployment-649b855c6b-rjnnq6	2/2	Running	0	2m32s

3. Access the microservice again with your web-browser through its NodePort. Enter any text then, click **Submit**. Now, all of the external traffic is blocked by Istio. Complete the steps that are presented in “Allowing external outbound traffic” on page 122 to allow outbound (egress) traffic.

Allowing external outbound traffic

As described in “Redeploying the application” on page 121, the external traffic is blocked by default by Istio. This capability of Istio secures your service mesh.

Complete the following steps to allow outbound (egress) traffic to a specific endpoint, which is Language Translator API in this scenario:

1. In your terminal, go to the following istio directory:

```
cd istio
```
2. When you define Istio Egress traffic, you communicate within your service mesh as HTTP even with external services and Istio handles the HTTPS external communication, as shown in Figure 4-12.

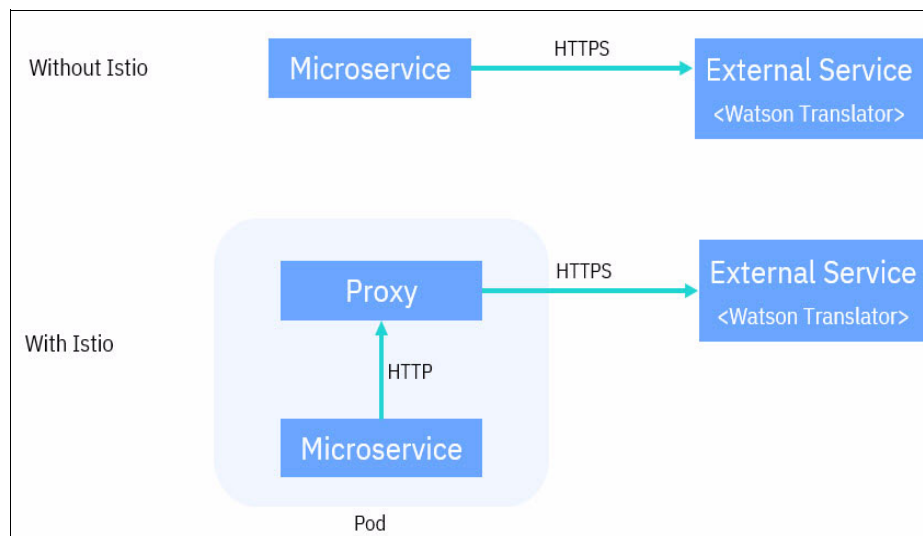


Figure 4-12 Communicating with external services in Istio

3. Figure 4-13 on page 123 shows how to define egress rules in Istio for enabling outbound traffic from the service mesh to a specific HTTPS host. Egress Rules are defined in Istio Pilot then Pilot converts high level routing rules that control traffic behavior into Envoy-specific configurations, and propagates them to the sidecars at runtime.
 - ServiceEntry
Enables services within the mesh to access a service that is not necessarily managed by Istio. The rule describes the endpoints, ports, and protocols of a white-listed set of mesh-external domains and IP blocks that services in the mesh can access. In this example, you add a ServiceEntry to define the Watson Language Translator APIs by specifying the port and stating that the port is external to the mesh.
 - VirtualService
Defines a set of traffic routing rules to apply when a host is addressed. Each routing rule defines a matching criteria for the traffic of a specific protocol. If the traffic is matched, it is sent to a named destination service (or a subset or a version of it) that is defined in the DestinationRule. In this example, notice that it matches with HTTP only and redirects the traffic to the 443 HTTPS port that was defined in the DestinationRule. Therefore, your applications used HTTP within your service mesh for communicating with external services and Istio handles the HTTPS external communication.

- DestinationRule

Defines policies that apply to the traffic that is intended for a service after routing occurred. Any destination host and subset reference in a VirtualService rule must be defined in the corresponding DestinationRule. In this example, the DestinationRule starts HTTPS when accessing gateway-lon.watsonplatform.net.

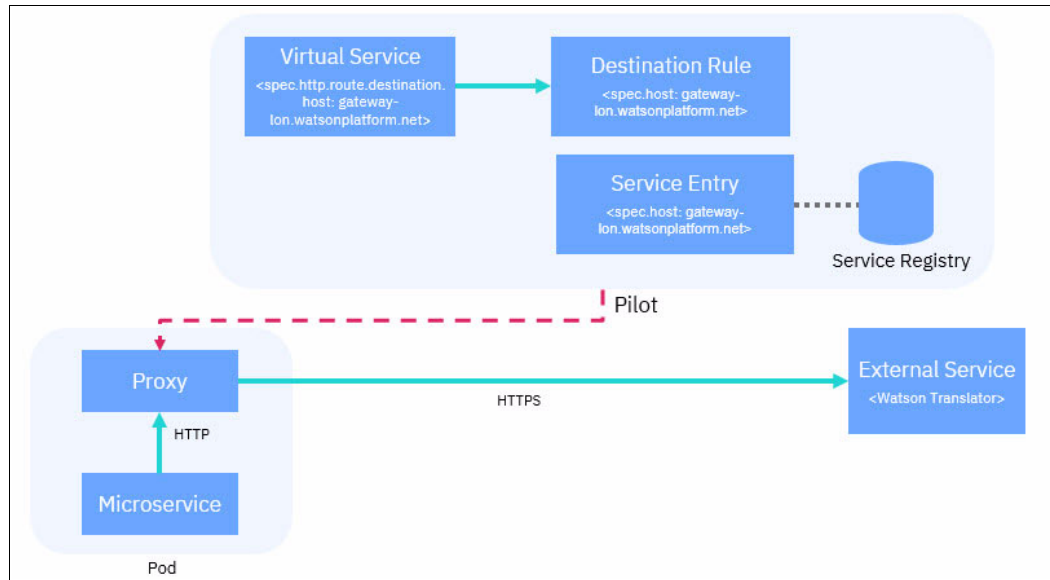


Figure 4-13 Defining egress rules in Istio for enabling outbound traffic

4. Example 4-35 shows the istio configurations that are needed to define the egress rules in Istio for enabling outbound traffic from the service mesh to a specific HTTPS host.

Example 4-35 Istio configuration

```
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: watson
spec:
  hosts:
  - gateway-lon.watsonplatform.net
  ports:
  - number: 443
    name: http-port-for-tls-origination
    protocol: HTTP
  resolution: DNS
  location: MESH_EXTERNAL
---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: watson
spec:
  hosts:
  - gateway-lon.watsonplatform.net
  http:
  - match:
    - port: 80
```

```

    route:
    - destination:
        host: gateway-lon.watsonplatform.net
        port:
            number: 443
        weight: 100
---
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: watson-tls-originate
spec:
  host: gateway-lon.watsonplatform.net
  trafficPolicy:
    loadBalancer:
      simple: PASSTHROUGH
    portLevelSettings:
    - port:
        number: 443
      tls:
        mode: SIMPLE # initiates HTTPS when accessing
gateway-lon.watsonplatform.net

```

5. As explained in step 2, from the microservice perspective, the microservice should communicate with the external service as HTTP; then, Istio handles the HTTPS communication. Change the secret to change `watson-translator.url` from <https://gateway-lon.watsonplatform.net/language-translator/api> to <http://gateway-lon.watsonplatform.net/language-translator/api>.
6. Run the following command to decode the values of the updated url to Base64 by using the command line or any other tool:


```
echo -n http://gateway-lon.watsonplatform.net/language-translator/api|base64
```
7. Update the secret `watson-translator.url` by completing the following steps:
 - a. Browse to and log in to IBM Cloud Private console by using your web browser.
 - b. Click **Menu** → **Configuration** → **Secrets**.
 - c. Locate the `watson-translator` secret then, click Then, click **Edit**, as shown in Figure 4-14 on page 125.

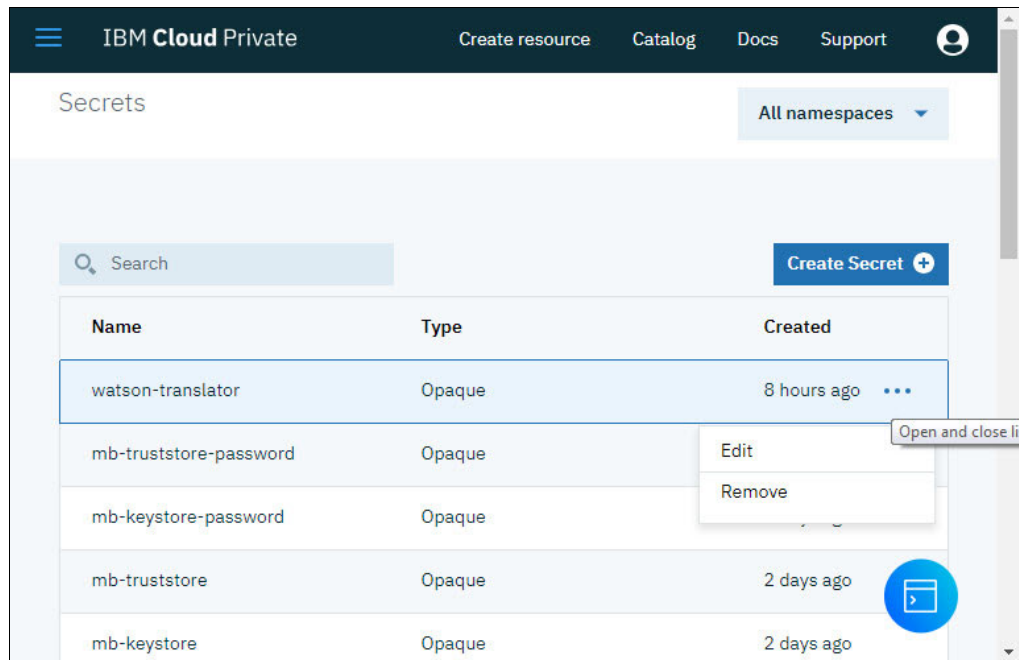


Figure 4-14 Secret options

- d. Edit the url with the updated Base64 of URL and then, click **Submit**, as shown in Figure 4-15.

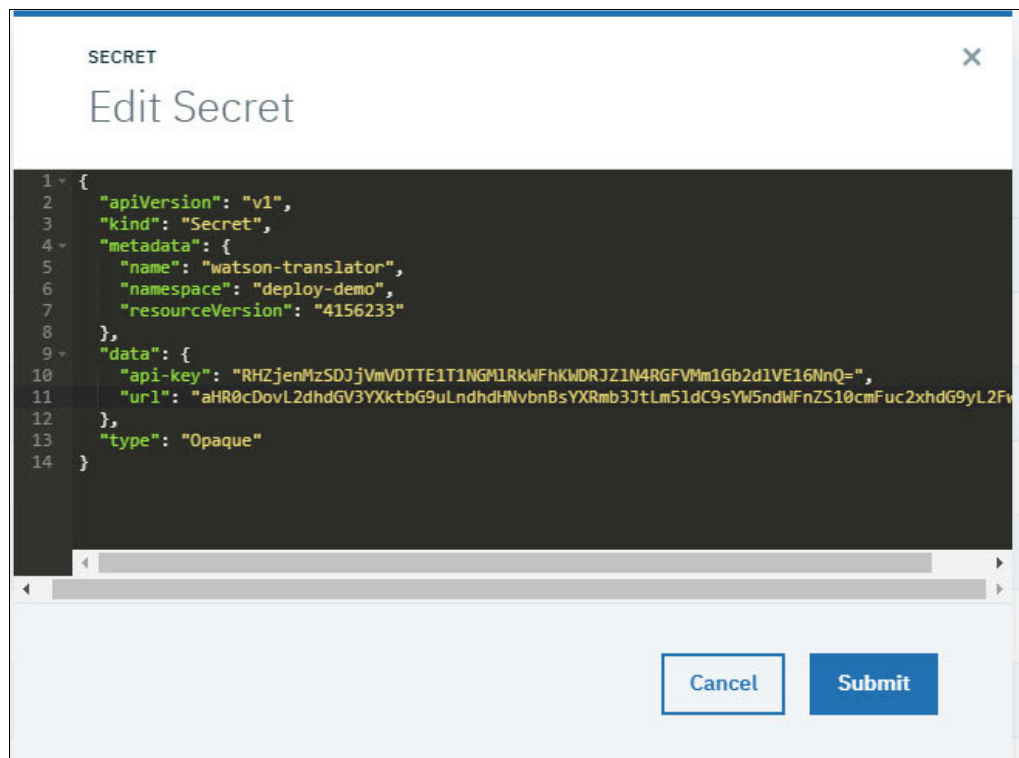


Figure 4-15 Edit Secret

8. Delete all the pods that are related to the deployment so that Node.js picks up the updated URL by running the following command:

```
kubectl delete pods --selector=app=translator-ms
```

9. Access the microservice again by using your web-browser through its NodePort. Enter some text then, click **Submit**. You should now have a successful result that displays the translated text.

4.3.5 Exposing services to be used externally through defining ingress rules

In this section, you define ingress rules to expose your service to be used externally.

Complete the following steps to define ingress rules for your service:

1. Example 4-36 shows how to define ingress rules in Istio. The following components are available:
 - Gateway
Describes a load balancer that is operating at the edge of the mesh that is receiving incoming or outgoing HTTP/TCP connections. In this example, you use the default Istio gateway implementation.
 - VirtualService
Defines a set of traffic routing rules to apply when a host is addressed. Each routing rule defines matching criteria for traffic of a specific protocol. If the user browses to the prefix /translator-ms, Istio redirects requests to the service translator-ms-service. The output page is run through a POST to /watson-translator. Another route is defined for it for Istio to redirect the output page to the translator-ms-service.

Example 4-36 Defining ingress rules

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: translator-ms-gateway
spec:
  selector:
    istio: ingressgateway # use Istio default gateway implementation
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
    - "*"
---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: translator-ms
spec:
  hosts:
  - "*"
  gateways:
  - translator-ms-gateway
  http:
  - match:
```

```
- uri:
  prefix: "/translator-ms"
rewrite:
  uri: "/"
route:
- destination:
  port:
    number: 80
    host: translator-ms-service
- match:
  - uri:
    prefix: "/watson-translator"
  route:
  - destination:
    port:
      number: 80
      host: translator-ms-service
```

2. Apply the Istio rule, as shown in Example 4-37.

Example 4-37 Apply Istio ingress rule

```
istioctl create -f 02trans-ms-ingress.yaml
Created config gateway/deploy-demo/translator-ms-gateway at revision 4233161
Created config virtual-service/deploy-demo/translator-ms at revision 4233163
```

3. You access the service through the Istio ingress gateway. For illustration purposes, you can access it through the Istio service NodePort. Get the NodePort of the istio-ingressgateway by running the following command:

```
kubectl get svc istio-ingressgateway -n=istio-system
```
4. Browse to the service through your web-browser by using {Proxy Node IP}:{Istio Ingress Gateway NodePort}/translator-ms-service. The application should be up and running, as shown in Figure 4-16.



Figure 4-16 Accessing the application through ingress

4.3.6 Handling an unreliable external service by simulating different failures through Istio abort and delay injection

In this section, you explore how to simulate different failures for the external service through injecting Abort in the service mesh and delays.

Istio fault injection

Example 4-38 shows how to inject fault with Istio. In this example, it injects the status code “500”, which means “Internal Server Error” with a frequency of 50% of the time. This injection simulates that Language Translator API returns this error. You use this during the testing of your application to ensure that it can handle different failures.

Example 4-38 Istio fault injection

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: watson
spec:
  hosts:
  - gateway-lon.watsonplatform.net
  http:
  - match:
    - port: 80
    route:
    - destination:
        host: gateway-lon.watsonplatform.net
        port:
          number: 443
      weight: 100
    fault:
      abort:
        httpStatus: 500
        percent: 50
```

Apply the rule to your Istio by running the **istioctl** command, as shown in the following example:

```
istioctl replace -f 03watson-fault.yaml
Updated config virtual-service/deploy-demo/watson to revision 4242847
```

Notice that now 50% of the time, you receive “error: 500” when you try to translate a text. This result indicates that you need a better error handling in your application to display a more explanatory message in case the service responds with statusCode 500.

Istio delay injection

Example 4-39 shows how to delay fault. In this example, it injects a delay of 50 seconds 100% of the time. This result simulates that the Language Translator API response is received after this delay. You use this during testing your application to ensure that it can handle different delays.

Example 4-39 Istio delay injection

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
```

```
name: watson
spec:
  hosts:
  - gateway-lon.watsonplatform.net
  http:
  - match:
    - port: 80
    route:
    - destination:
        host: gateway-lon.watsonplatform.net
        port:
          number: 443
        weight: 100
    fault:
      delay:
        percent: 100
        fixedDelay: 50s
```

Apply the rule to your istio through `istioctl` as shown in the following example:

```
istioctl replace -f 04watson-delay.yaml
```

Updated config virtual-service/deploy-demo/watson to revision 4242847

Notice that after you click **Submit**, it takes at least 50 seconds to see a response.

4.3.7 Enhancing the application resiliency by setting request timeouts and adding automatic retry attempts

In this section, you explore how to set request timeout and how to handle automatic retry attempts without any modification to the code.

Setting request timeouts

Example 4-40 shows how to set a request timeout. You just added a timeout to the ingress rule (see the last line in bold in Example 4-40). In this example, `/watson-translator` ingress times out after 5 seconds. Because you injected delay of 50 seconds for `watson` egress in the previous section, this ingress should time out.

Example 4-40 Istio: Define a timeout for ingress

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: translator-ms
spec:
  hosts:
  - "*"
  gateways:
  - translator-ms-gateway
  http:
  - match:
    - uri:
        prefix: "/translator-ms"
    rewrite:
      uri: "/"
    route:
```

```

- destination:
  port:
    number: 80
  host: translator-ms-service
- match:
  - uri:
    prefix: "/watson-translator"
  route:
  - destination:
    port:
      number: 80
    host: translator-ms-service
timeout: 5s

```

Apply the rule to your Istio through `istioctl` as shown in the following example:

```
istioctl replace -f 05trans-ms-timeout.yaml
```

Updated config virtual-service/deploy-demo/translator-ms to revision 4245924

Notice that after you click **Submit**, the request times out after 5 seconds with the message “upstream request timeout”. During the development, you must enhance your application to handle the timeout in a more explanatory way. By using Istio, you can know your application’s behavior in these scenarios so that you can enhance it.

Adding automatic retry attempts

Example 4-41 shows how to set automatic retry attempts. You just added a retries to the egress rule (last three lines in bold in Example 4-41). In this example, Istio automatically retries the endpoint if it fails 10 times and times out in each retry after 2 seconds.

Example 4-41 Retries

```

apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: translator-ms
spec:
  hosts:
  - "*"
  gateways:
  - translator-ms-gateway
  http:
  - match:
    - uri:
      prefix: "/translator-ms"
    rewrite:
      uri: "/"
    route:
    - destination:
      port:
        number: 80
      host: translator-ms-service
  - match:
    - uri:
      prefix: "/watson-translator"
    route:
    - destination:

```

```
port:
  number: 80
  host: translator-ms-service
timeout: 5s
retries:
  attempts: 10
  perTryTimeout: 2s
```

Apply the rule to your Istio through `istioctl`, as shown in the following example:

```
istioctl replace -f 06trans-ms-retries.yaml
Updated config virtual-service/deploy-demo/translator-ms to revision 4250902
```

Apply the 50% fault injection rule to be able to test the retries, as shown in the following example:

```
istioctl replace -f 03watson-fault.yaml
Updated config virtual-service/deploy-demo/watson to revision 4251462
```

Notice that after you click **Submit**, the request succeeds even if we have an abort failure injected 50% of the time. This result occurs because Istio keeps retrying the service 10 times. Because the fault is injected only 50% of the time, the service should pass in one of the 10 attempts.

4.4 Enforcing policy controls

Istio provides a flexible model to enforce authorization policies and collect telemetry for the services in a mesh.

Infrastructure backends are designed to provide support functionality that is used to build services. They include such things as access control systems, telemetry capturing systems, quota enforcement systems, and billing systems. Services are traditionally directly integrated with these backend systems, which creates a hard coupling and bakes-in specific semantics and usage options.

Istio provides a uniform abstraction that makes it possible for Istio to interface with an open-ended set of infrastructure backends. This process is done in such a way to provide rich and deep controls to the operator, while imposing no burden on service developers.

Istio also is designed to change the boundaries between layers to reduce systemic complexity, eliminate policy logic from service code, and give control to operators.

Figure 4-17 on page 132 shows enforcing policy control using Mixer.

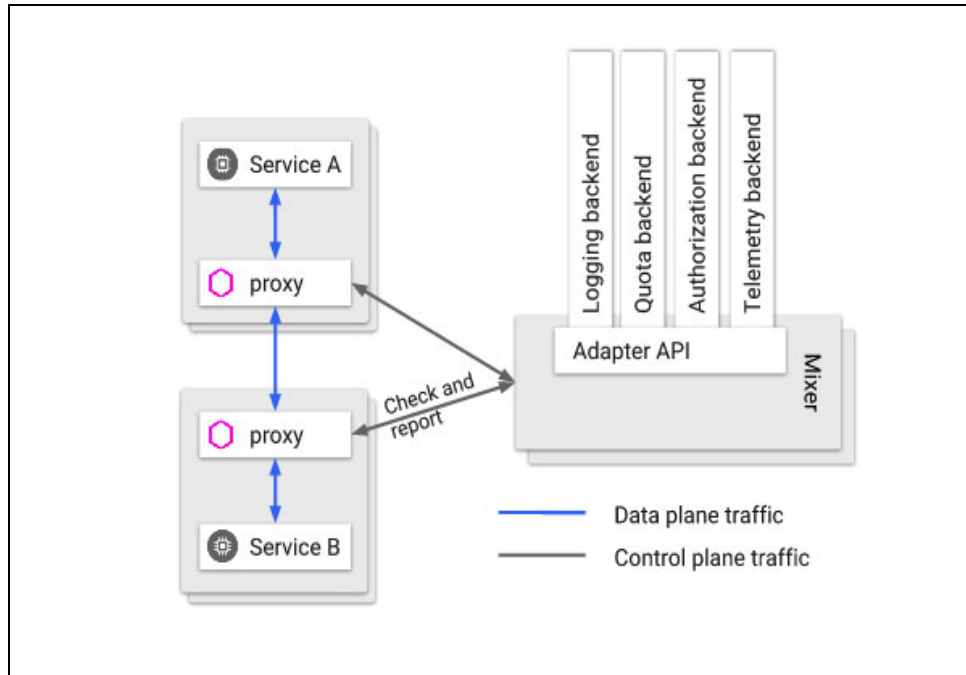


Figure 4-17 Enforcing policy control using Mixer¹

The Envoy sidecar logically calls Mixer before each request to perform precondition checks, and after each request to report telemetry. The sidecar has local caching such that most precondition checks can be performed from cache. Additionally, the sidecar buffers outgoing telemetry such that it calls Mixer only infrequently.

At a high level, Mixer provides the following benefits:

- **Backend abstraction:** Mixer insulates the rest of Istio from the implementation details of individual infrastructure backends.
- **Intermediation:** Mixer allows operators to have fine-grained control over all interactions between the mesh and infrastructure backends.

Policy enforcement and telemetry collection are entirely driven from configuration. These features can be disabled, which avoids the need to run the Mixer component in an Istio deployment.

For more information, see this [Istio web page](https://istio.io/docs/concepts/policies-and-telemetry/).

¹ Image source: <https://istio.io/docs/concepts/policies-and-telemetry/>



Application development with Cloud Foundry

This chapter discusses the application development aspect of using Cloud Foundry and includes the following topics:

- ▶ 5.1, “Introduction” on page 134
- ▶ 5.2, “What is Cloud Foundry?” on page 134
- ▶ 5.3, “Application high availability” on page 134
- ▶ 5.4, “Autoscale” on page 135
- ▶ 5.5, “Using external services ” on page 135
- ▶ 5.6, “Application packaging” on page 135
- ▶ 5.7, “IBM buildpacks versus community” on page 136
- ▶ 5.8, “Zero downtime deployment” on page 136
- ▶ 5.9, “Command lines” on page 137
- ▶ 5.10, “Sample application to use” on page 137

5.1 Introduction

Cloud Foundry is an application service that allows developers to focus on code and not application hosting, routing, or scale. Cloud Foundry makes it easy to add and remove routes for your applications. It manages load balancing inbound routes when multiple instances of an application occur.

When Twelve-Factor Application development is used, instances can be created or destroyed on-demand to meet the needs of the enterprise. All the routing that is required to divert and shape traffic to different instances is handled automatically by Cloud Foundry.

Cloud Foundry with IBM Cloud Private Kubernetes also brings stateful services to your enterprise applications. These local services can be provisioned by the CLI or console and attached to your applications. Kubernetes offers accelerated deployment and tear down, which allows developers access data services on-demand at any time.

5.2 What is Cloud Foundry?

Cloud Foundry is a platform for delivering cloud applications. It builds on the Twelve-Factor Application concept, where the application is stateless and services are bound to the application to provide state, processing, and transformations.

It allows developers to focus on developing and lowers the bar of entry to anyone with a basic knowledge of a programming language. With that, developers can get up and running quickly, whether they are new or experienced. It also manages all the aspects of application hosting and routing, which means experienced developers can switch languages, and push applications even faster.

5.3 Application high availability

Cloud Foundry offers application high availability by way of stateless instances. With this tenant for cloud applications, Cloud Foundry can scale the number of instances by adding or removing instances of the application. This process occurs quickly because all of the artifacts that are needed to host the application are readily available and can be combined at a moments notice.

IBM Cloud Private Cloud Foundry usually operates with more than one Diego cell host. With this fact in mind, Cloud Foundry instantiates new instances on different cells to improve the high availability of the application. Then, if an outage of the instance is detected, it is restarted in a working Diego cell, with an attempt that is made by the placement engine not to colocate instances of the same application when possible.

5.4 Autoscale

IBM Cloud Private Cloud Foundry offers an autoscale service for automatically managing the instance counts of your applications. It can be used to increase or decrease an application by using memory that is used, response time, and throughput. Several other factors can be used to better constrain the autoscaler so that the best application performance versus resource utilization can be maintained.

For more information about the autoscaler implementation that is available for use with IBM Cloud Private Cloud Foundry, see [IBM Knowledge Center](#).

5.5 Using external services

Consider the following points regarding the use of external services:

- ▶ Cloud Application runtimes are inherently stateless. Therefore, multiple instances can be started or stopped, and the application continues to function. This ability often is achieved by using stateful services.

Cloud Foundry provides a catalog of services in the Cloud Foundry marketplace. The marketplace displays services that are available to the user that can be local, dedicated, or public in nature. In the default configuration, IBM public services and Kubernetes-based data services are available (Postgres, MongoDB, IBM Db2®, Rabbit MQ, and MariaDB).

- ▶ With local services running on Kubernetes, the provisioning speed of the service is quick. No more waiting around for VMs to spin up and software to be deployed. The service containers are pre-built to run the application in a lean, expandable format. Creating and removing occurs in seconds.

The fact that containers also manage resources well (because of their streamlined nature), means more services fit on a host than when VMs are used. They also easily grow with the service (a VM requires much more intervention or to be reprovisioned).

5.6 Application packaging

IBM Cloud Private Cloud Foundry comes includes several community and IBM buildpacks:

- ▶ IBM Buildpacks:
 - SDK for NodeJS
 - Liberty
 - Swift
 - Linux DotNet
- ▶ Community Buildpacks:
 - Golang
 - PHP
 - Java
 - Ruby
 - Python
 - Binary buildpack
 - Staticfile buildpack
 - Hostable Web Core (HWC) Windows legacy DotNet

This wide breath of languages means that new developers can start hosting their applications on immediately. It also provides experienced developers an easy way to move between languages without having to deal with the varying hosting and publishing requirements.

5.7 IBM buildpacks versus community

IBM-provided buildpacks are instrumented, reviewed, and cleared open source, and security scanned. IBM maintains and applies security patches and application environment-level maintenance and enhancement. IBM buildpacks are fully supported by our Cloud Application Support organization through IBM's standard support processes.

5.8 Zero downtime deployment

Blue/Green deployments are the idea that you deploy an update to your application on a different host URL, validate that the application is working, then seamlessly transfer traffic from the old application instance to the new one. This process allows for zero downtime during a production upgrade. It also provides a window to test the application in production before transferring all traffic.

Example 5-1 shows a basic Blue/Green deployment example.

Example 5-1 Blue/Green deployment example

```
#Push the production application
cf push prod1-green -n my-production-app
#Now I make updates to the application and push a new version, but to a temporary
URL.
cf push prod1-blue -n my-production-app-blue
#I now validate the new version of the application at the URL prefix
my-production-app-blue...
#When I am satisfied that the blue version of the application is working, I map
the existing route used by the green app, to the blue app
cf map-route prod1-blue APP_DOMAIN --hostname my-production-app
#now all traffic to my-production-app.APP_DOMAIN will be sent to either
prod1-green or prod1-blue
# Finally we remove the old route mapping from prod1-green
cf unmap-route prod1-green APP_DOMAIN --hostname my-production-app
#For good measure, we should also remove the old blue route
cf unmap-route prod1-blue APP_DOMAIN --hostname my-production-app-blue
#now remove the old production application
cf delete prod1-green
#Finally rename the prod1-blue to prod1-green
cf rename prod1-blue prod1-green
```

At this point, the update is complete, and no outage should be experienced by the users.

5.9 Command lines

IBM Cloud Private Cloud Foundry uses the command lines from the community. As such, the developer can use the same sets of command lines in an IBM Cloud Private Cloud Foundry environment.

5.10 Sample application to use

For more information about a sample MongoDB Java application that you can use to test the Cloud Foundry functionality in your environment, see [this web page](#).

This sample application provides you with a sample workflow for working with any Liberty app on IBM Cloud or in IBM Cloud Private. You set up a development environment, deploy an app locally and on the cloud, and then, integrate an IBM Cloud database service in your app.



Additional material

This book refers to additional material that can be downloaded from the internet as described in the following sections.

Locating the GitHub material

The web material that is associated with this book is available in softcopy on the internet from the [IBM Redbooks GitHub repository](#).

Cloning the GitHub material

Complete the following steps to clone the GitHub repository for this book:

1. Download and install Git client if not installed from [this web page](#).
2. Run the following command to clone the GitHub repository:

```
git clone  
https://github.com/IBMRedbooks/SG248441-IBM-Cloud-Private-Application-Developer  
-s-Guide.git.
```


Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this book.

IBM Redbooks

The IBM Redbooks publication *IBM Cloud Private: System Administrator's Guide*, SG24-8440, provides more information about the topic in this document. Note that this publication might be available in softcopy only.

You can search for, view, download or order this document and other Redbooks, Redpapers, Web Docs, draft and additional materials, at the following website:

ibm.com/redbooks

Online resources

The following websites are also relevant as further information sources:

- ▶ IBM Cloud Private v3.1.2 documentation:
https://www.ibm.com/support/knowledgecenter/en/SSBS6K_3.1.2/kc_welcome_containers.html
- ▶ IBM Cloud Private v3.1.2 supported platforms:
https://www.ibm.com/support/knowledgecenter/en/SSBS6K_3.1.2/supported_system_config/supported_os.html
- ▶ Spring Boot project website:
<https://start.spring.io>
- ▶ Full list of predefined values for Helm:
https://helm.sh/docs/developing_charts/#predefined-values
- ▶ Current list of Helm template recommended practices:
https://github.com/helm/helm/blob/master/docs/chart_best_practices/templates.md
- ▶ Helm Developer's Guide:
https://helm.sh/docs/chart_template_guide/#the-chart-template-developer-s-guide
- ▶ Helm Chart dependencies:
<https://github.com/helm/helm/blob/master/docs/charts.md#chart-dependencies>
- ▶ Microclimate website:
(<https://microclimate-dev2ops.github.io/creatingaproject#doc>)
- ▶ IBM Cloud Developer Tools:
<https://console.bluemix.net/docs/cli/idx/index.html#developing>
- ▶ Helm public website:
(<https://helm.sh>)

- ▶ IBM Helm Charts repository:
<https://github.com/IBM/charts>
- ▶ CloudPak certification guidance:
<https://github.com/IBM/cloud-pak/blob/master/community/cloud-pak.md>
- ▶ Helm Charts Best Practices Guide:
https://github.com/helm/helm/tree/master/docs/chart_best_practices
- ▶ Istio concepts:
<https://istio.io/docs/concepts/what-is-istio/>
- ▶ Knowledge Center link for installing Istio:
https://www.ibm.com/support/knowledgecenter/SSBS6K_3.1.2/manage_cluster/istio.html
- ▶ Ingress documentation:
<https://github.com/kubernetes/ingress-nginx/blob/master/docs/user-guide/nginx-configuration/annotations.md>
- ▶ DevOps for Dummies (IBM Edition):
http://www.ibm.com/common/ssi/cgi-bin/ssialias?subtype=BK&infotype=PM&appname=SWGGE_RA_VF_USEN&htmlfid=RAM14026USEN&attachment=RAM14026USEN.PDF#1oa
- ▶ Sauce Labs information:
<https://wiki.saucelabs.com/display/public/DOCS/The+Sauce+Labs+Cookbook+Home>
- ▶ Installing Microclimate on IBM Cloud Private:
<https://github.com/IBM/charts/tree/master/stable/ibm-microclimate>
- ▶ Istio telemetry website:
<https://istio.io/docs/tasks/telemetry/>

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services



SG24-8441-00

ISBN 0738457620

Printed in U.S.A.

Get connected

